T-RDF: A Task-Based Parallel System for Efficiently Answering SPARQL Queries

Lyuheng Yuan*, Da Yan*, Saugat Adhikari*, Jiao Han*, Lei Zou‡, Yang Zhou+
*Indiana University Bloomington {lyyuan, yanda, adhiksa, jiaohan}@iu.edu
‡Peking University zoulei@pku.edu.cn
+Auburn University yangzhou@auburn.edu

***Abstract*—The RDF data model and SPARQL query language are two standards recommended by W3C for representing and querying linked data on the Web. They have been widely used in commercial knowledge graphs and public knowledge bases. Existing SPARQL query engines rely intensively on join operations which usually generate huge redundant intermediate data. This is inefficient especially when answering "heavy" queries with many conditions (and hence many joins) and low selectivity. Moreover, intra-query parallelism is under-explored for SPARQL queries.**

Two technology breakthroughs emerge that provide new opportunities to speed up SPARQL query processing: (1) faster algorithms for subgraph matching which is essential to SPARQL querying, and (2) the think-like-a-task (TLAT) parallel computing model that enables full utilization of CPU cores. To utilize these new breakthroughs, we propose T-RDF, a new SPARQL query engine equipped with the latest subgraph matching algorithms as well as TLAT-based parallelization techniques to fully utilize all CPU cores. T-RDF utilizes both inter- and intra-query parallelism, and avoids straggler tasks by using a task timeout mechanism. Experiments show that T-RDF is orders of magnitude faster than existing SPARQL query engines, and achieves ideal speedup as the number of CPU cores increases.

I. Introduction

Resource Description Framework (RDF) is a standard data model recommended by W3C for the Semantic Web. RDF represents linked data as a set of ⟨*subject, predicate, object*⟩ triplets, forming a directed graph with edge labels (predicates).

SPARQL is the standard query language for RDF recommended by W3C. Its most common query form is "SELECT RD where GP", where GP is a graph pattern and RD is a result description. The most basic form of GP is a set of triple patterns, called BGP (Basic Graph Pattern). BGPs can be combined to create advanced GP such as Group Graph Pattern (AND of BGPs) or Union Graph Pattern (OR of BGPs), so this paper focuses on answering BGPs efficiently.

Figure 1 shows a simplified RDF graph following the university domain ontology of the Lehigh University Benchmark (LUBM) [9], where the rdfs:type predicate for specifying node classes are represented with node colors instead, and the labels of other predicates are in shorthand notation. Each node has a label that is either a class, an entity, or a literal. An entity or a class is usually labeled by a unique Uniform Resource Identifier (URI), while a literal is a constant value such as strings, numbers, and dates. However, for querying purpose, we can simply view all of them as strings.

Figure 2 shows a SPARQL query to find all professor(X)–course(Y)–student(Z) tuples where X is both Z's advisor and

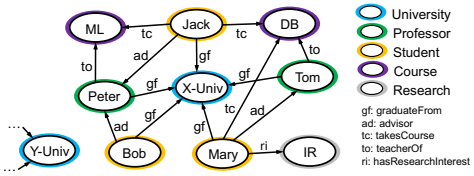


Fig. 1. An RDF Graph for Illustration

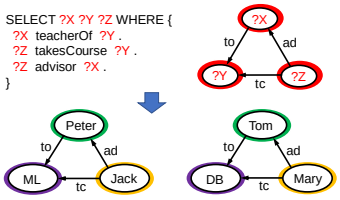


Fig. 2. An Example SPARQL Query

the instructor of the course Y that Z is also taking. When applying this query to the data in Figure 1, we obtain two subgraph matches as shown at the bottom of Figure 2, from which we can extract RD. Clearly, GP can be regarded as a query graph for subgraph matching.

RDF and SPARQL have been widely used in many public knowledge bases including DBpedia [19], Probase [63], PubChemRDF [28] and Bio2RDF [21]. Besides human-formulated SPARQL queries, many SPARQL queries are generated from natural language questions [20], [57], [59], [64] so that they can be executed on knowledge graphs. Since such queries can have many triple patterns in GP, evaluating them by join operations (as done by most existing systems) is too costly, and a faster SPARQL query engine supporting load-balanced intra-query parallelism is in urgent need.

Two recent technology breakthroughs have brought new opportunities to speed up SPARQL query processing. The first breakthrough is a series of new algorithms and systems that significantly advanced the efficiency of subgraph matching [38], [49]–[55], [76], the techniques of which can benefit GP matching in SPARQL query evaluation. The second breakthrough is a new parallel framework called T-thinker [8], [67], or think-like-a-task (TLAT), that is able to speed up subgraph finding problems (including subgraph matching) ideally with the number of CPU cores used. Many systems have been developed on top of T-thinker including G-thinker [29], [30], [39], [66], [68], G-thinkerQ [71], PrefixFPM [25], [46], [69], [70], T-FSM [73], T-DFS [72] and TreeServer [65].

Utilizing both breakthroughs, we propose T-RDF, a new SPARQL query engine equipped with the latest subgraph

matching algorithms as well as TLAT-based parallelization techniques to fully utilize CPU cores. Our contributions are:

- T-RDF adopts a task-based query engine design that fully utilizes CPU cores by both inter- and intra-query parallelism, while keeping memory cost bounded. Also, tasks of queries received earlier are prioritized for scheduling to respect fairness while keeping CPU cores busy.
- Different SPARQL queries can have drastically different evaluation workloads due to different number of triple patterns in GP and different selectivity of constants in each triple pattern; even for the same query, the selectivity of different triple patterns in different regions of an RDF graph can also be very different. For load balancing, T-RDF adopts a timeout mechanism to allow tasks to be automatically decomposed into subtasks after running beyond a time threshold, which eliminates straggler tasks.
- T-RDF devises novel candidate size estimation techniques for the variables in a SPARQL query, which are faster and more accurate than existing methods such as VS-tree [77], [78]. In particular, (1) T-RDF utilizes a collection of indices are carefully selected to be the minimal collection tailored for our new subgraph matching algorithm; (2) T-RDF treats both constants and variables in the query in a uniform manner, and accurately estimates their selectivity to pick the next most selective node to explore, which can significantly reduce the running time.
- T-RDF adopts the newest search-space pruning techniques to speed up GP matching, including using a good matching order and SIMD set intersection algorithms.
- T-RDF adopts a hash-based pruning method to effectively detect invalid variable matches early during backtracking search to avoid wasted search on the RDF graph.

While a lot of SPARQL queries are "light", the performance bottleneck of a query engine is often caused by "heavy" queries, so T-RDF is designed to allow "heavy" queries to run much faster than in existing systems while retaining the efficiency of "light" queries. Experiments show that T-RDF is orders of magnitude faster than existing query engines, and scales up well with the number of CPU cores.

In the sequel, Section III describes our GP matching algorithm. Due to space limit, we present how the algorithm is parallelized as independent tasks and how the tasks are managed by our T-RDF query engine in Appendix C [6]. Finally, Section IV reports experiments and Section V concludes.

II. Related Work

Due to page limit, we keep this section brief, and provide a more detailed review in Appendix A accessible online at [6].

RDF Query Engines. Most existing SPARQL engines rely on join-based approaches, where triple patterns are scanned to produce intermediate tables that are joined for the final results. Systems such as RDF-3X [43] and Hexastore [62] index all SPO permutations to avoid full scans, while BitMat [18] and TripleBit [74] use compressed bit-matrix representations. To mitigate large intermediate results, Trinity.RDF [75] employs graph exploration with one-step pruning, and Wukong [48] uses full-history pruning to eliminate joins. gStore [77] adopts subgraph matching with signature-based pruning and a VS-tree index. SPARTex [13] implements a vertex-centric model supporting user-defined graph algorithms. MAGiQ [36] instead reformulates SPARQL into sparse matrix algebra, though its edge-driven pruning is costly. Recent work also explores more user-friendly query formulation [20], [59]. With advances in subgraph matching algorithms and the think-like-a-task (TLAT) parallel model, it is timely to revisit SPARQL query engine design via (parallel) subgraph matching. Notably, $\text{Turbo}_{\text{HOM}}$++ [40] is based on the older and less efficient $\text{Turbo}_{\text{ISO}}$ [33] for subgraph matching, which supports NUMA-aware parallelism but is not open-sourced for public use.

Subgraph Matching. Recent advances in subgraph matching, particularly the study by Sun and Luo [51], unify eight algorithms into a three-stage framework: candidate filtering, query vertex ordering, and Ullmann-style graph exploration. Their work highlights the benefits of auxiliary indices and efficient set intersection, but since many SPARQL queries are highly selective, we find the index construction cost outweighs its benefit. Thus, in T-RDF we adopt their recommended vertex ordering and set intersection techniques, while relying on lightweight preprocessing indices to estimate candidate sizes and guide efficient query vertex ordering.

Think-Like-a-Task (TLAT) Parallelism. The TLAT model [8], [67] was introduced as an abstract framework for parallelizing compute-intensive problems, with G-thinker [66], [68] demonstrating CPU-scalable distributed subgraph finding that outperforms prior systems by up to two orders of magnitude. Examples of other TLAT systems include PrefixFPM for frequent pattern mining [69], [70] and T-FSM for frequent subgraph mining [73]. Unlike these offline analytics systems, T-RDF targets online querying with multiple concurrent queries.

III. Algorithm for GP Matching

To avoid storing and processing long strings during query answering, each string in S, P or O is first converted into a unique ID, with the bidirectional mapping maintained for string-to-ID conversion during query parsing, and ID-to-string recovery during answer delivery. This is following the convention such as the mapping dictionary of RDF-3X [43], and the string server in Trinity.RDF [75] and Wukong [48]. Even though the original gStore directly hashed strings into signatures [77], its latest version now hashes IDs [2], [3]. Note that these mappings can be efficiently implemented by prefix compression as illustrated in Figure 1 of TripleBit [74].

We describe our approach to GP matching in three subsections: (1) indexing, (2) query variable ordering, and (3) graph exploration.

A. Indexing for the RDF Graph

To quickly locate the matching candidates for each triple pattern, existing works create indices on (S, P, O) permuta-

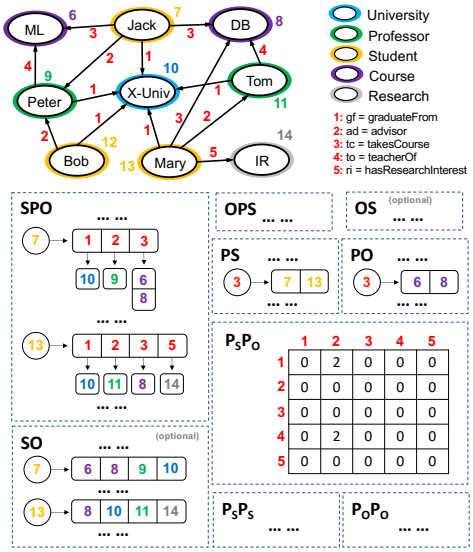


Fig. 3. Indices Maintained by T-RDF

tions. We also create a few indices to be used by our new matching algorithm (see Figure 3):

- **SPO & OPS:** These indices are used to determine local candidates for the next triple pattern to match. For example, for the query in Figure 2 on Page 1, suppose that ?Z is already matched with node 7 (Jack), and we are now finding candidates for ?Y following the query triple pattern (?Z tc ?Y), then we can use the SPO index as shown in Figure 3 to obtain candidates SPO[7][3] = $\{6, 8\}$ for further graph exploration. Similar indices have been created by existing works such as the spo index of Hexastore illustrated in Figure 2 of [62].
- **SO & OS (optional):** These indices are for candidate finding for triple patterns like (?Z ?P ?Y). In our previous example where ?Z is matched with node 7 (Jack), we can find the candidates for ?Y as SO[7] = $\{6, 8, 9, 10\}$. These indices are optional since, for example, we can also get the candidates of ?Y from SPO[7] but additional redundancy removal of O may be needed since an object may be pointed to by multiple predicates, which incurs additional computing overheads.
- **PS & PO:** These indices are used to find initial node candidates when no variables are bound (i.e. matched). For example, to find the initial candidates of ?Z for the query in Figure 2, we notice that ?Z has two adjacent edges in the query graph (?Z tc ?Y) and (?Z ad ?X). Since ?Z is the subject for both adjacent predicates 'tc' (3) and 'ad' (2), we get candidate lists PS[3] = $\{7, 13\}$ and PS[2] = $\{7, 12, 13\}$ and intersect them to obtain the final candidates Jack (7) and Mary (13) for ?Z. Finding initial candidates from predicates is effective since some predicates can be very selective; a similar idea called predicate-based store has been explored by Wukong as illustrated in Figure 7 of [48].
- **P_SP_O, P_SP_S & P_OP_O:** These indices are for looking up or estimating the initial candidate size of any node variable in a query, which is needed to determine a good matching order for the query vertices. Figure 3 illustrates P_SP_O, where $P_SP_O[4][2] = 2$ since there are two candidates, Peter (9) and Tom (11), that are both subjects (S) for 'to' (4) and objects for 'ad' (2), which can be computed as PS[4]$\cap$ PO[2]. In our previous query, ?Z is the subject for both adjacent predicates 'tc' (3) and 'ad' (2), so we can obtain the exact candidate # as $P_SP_S[2][3] = |$PS[2]$\cap$ PS[3]$|$.

Note that P_SP_S and P_OP_O are symmetric matrices so we only need to keep the upper triangular matrix. Also, these predicate-predicate (PP) matrices are small since the number of predicates (similar to table attributes/columns in relational DB) is small as compared with the number of subjects and objects (similar to table rows). These PP matrices are pre-computed using PS and PO for direct access during querying. Here, we do not keep the joined lists between any two predicates but rather the list cardinality to be space efficient. Similar 'predicate $\times$ predicate' matrices are maintained by Trinity.RDF as mentioned in Section 5.6 of [75].

To be space-efficient, our collection of indices are carefully selected to be the minimal collection tailored for our new matching algorithm.

For each node v in the RDF graph, we also maintain a 250-bit signature using the current hashing scheme of gStore [2], where the first 150 bits are the bloom filter obtained by hashing all v's one-hop neighbors (S or O), and the other 100 bits are by hashing all v's adjacent predicates. We denote this signature by $sig(v)$. For each query node u, we can also compute a signature $sig(u)$ from its adjacent predicates and neighbors that are not variables in the query graph. In this way, we determine that v cannot match u if the bitwise OR of $sig(u)$ and $sig(v)$ is not equal to $sig(v)$, which is a quick pruning technique to avoid useless graph exploration.

B. Query Variable Ordering

Query Vertex Ordering. As indicated by [51], selecting a good query vertex order is critical for the performance of subgraph matching, and [51] empirically compared various vertex ordering methods and recommended that of GraphQL as the default. We, therefore, adopt the same method. Specifically, let us denote the candidate set of a query vertex u by $C(u)$, and the currently selected vertex order be $\pi_k = [u_1, u_2, \ldots, u_k]$. Then, the first vertex is selected as $u_1 = \arg\min_{u \in V(Q)} |C(u)|$, where $V(Q)$ is the set of all query vertices that are variables (rather than constants). After that, we iteratively select u_{k+1}= $\arg\min_{u \in N(\pi_k) - \pi_k} |C(u)|$, where $N(\pi_k)$ denotes the set of variables that are neighbors of at least one node in π_k.

Intuitively, we select the next node variable to match to be (1) connected to the set of query nodes already selected (as required by Ullmann's backtracking algorithm), and to be (2) such a query node that is the most selective. For example, for the query shown in Figure 4, we select u_1 as ?R since only node IR (14) in Figure 3 is the possible candidate, which is the most selective. To select u_2, however, we have to select it as ?Z since it is the only vertex connected to π_1 = [?R],

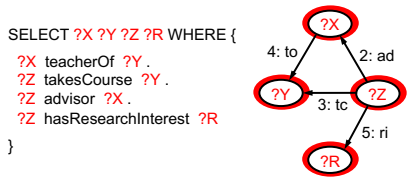


Fig. 4. A Sample Query with 4 Node Variables

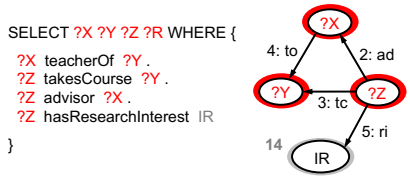


Fig. 5. A Sample Query with 3 Node Variables

even though it matches to 3 yellow vertices in Figure 4 which is more than the two green vertices (?X) and the two purple ones (?Y).

The next question is, how to obtain $C(u)$ for each query node variable u to be as tight as possible? We want to prevent, as much as possible, those candidates v from being included into $C(u)$ if graph exploration from v cannot ultimately lead to a valid subgraph match in the RDF graph. We answer this question next.

Candidate Size Estimation. As indicated by Wukong [48], if there exists a constant query vertex, it is often the most selective to start graph exploration from a constant vertex; while if all query vertices are variables, it is favorable to start graph exploration from the most selective predicate. We propose to treat both constant and variable query nodes in a uniform manner, and evaluate their selectivity with the help of adjacent predicates and neighbor nodes.

Specifically, for a query vertex u, let us denote its adjacent PO and PS pairs by $N(u) = \{(p^{(1)}, u^{(1)}), (p^{(2)}, u^{(2)}), \ldots, (p^{(n)}, u^{(n)})\}$. Then, **Case 1:** if there exist some $u^{(i)}$ that is a constant node, then we estimate $|C(u)|$ as follows since constant nodes are selective:

$$\min_{u^{(i)} \in N(u) \ \wedge \ u^{(i)} \text{ is a constant}} \left| \Pi(p^{(i)}, u^{(i)}) \right|, \tag{1}$$

where

$$\Pi\,(p^{(i)},u^{(i)}) = \begin{cases} \text{SPO}[u^{(i)}][p^{(i)}], & \text{if} \\ \text{SO}[u^{(i)}], & \text{if} \\ \text{OPS}[u^{(i)}][p^{(i)}], & \text{if} \\ \text{OS}[u^{(i)}], & \text{if} \end{cases}$$

	$p^{(i)}$	$u^{(i)}$
if	constant	subject
if	variable	subject
if	constant	object
if	variable	object

Note that $C(u)$ can be computed by the following intersection

$$\bigcap_{u^{(i)} \in N(u) \ \wedge \ u^{(i)} \text{ is a constant}} \Pi(p^{(i)}, u^{(i)}), \tag{2}$$

so Eq (1) gives a proper upper bound of $|C(u)|$.

To illustrate, consider Figure 5. Since ?Z has and only has one constant neighbor 'IR' (14), based on Eq (1), we have $|C(?\text{Z})| = |\Pi(5, 14)| = \text{OPS}[14][5] = \{13\,(\text{Mary})\}$ for the graph in Figure 3.

If all $u^{(i)} \in N(u)$ are variables, then **Case 2.1:** if there is only one $p^{(i)} \in N(u)$ that is a constant, then we estimate $|C(u)|$ as $|\text{P}(p^{(i)})|$, where:

$$\text{P}(p^{(i)}) = \begin{cases} \text{PO}[p^{(i)}], & \text{if } u^{(i)} \xrightarrow{p^{(i)}} u \\ \text{PS}[p^{(i)}], & \text{if } u^{(i)} \xleftarrow{p^{(i)}} u \end{cases}$$

Otherwise, **Case 2.2:** we estimate $|C(u)|$ as

$$\min_{p^{(i)},\ p^{(j)} \in N(u) \ \wedge \ p^{(i)} \text{ and } p^{(i)} \text{ are constants}} \left| \text{P}(p^{(i)}) \ \cap \ \text{P}(p^{(j)}) \right|. \tag{3}$$

where each term can be obtained directly from our PP matrices:

$$|\text{P}(p^{(i)}) \cap \text{P}(p^{(j)})| = \begin{cases} \text{P}_\text{O}\text{P}_\text{O}[p^{(i)}][p^{(j)}], & \text{if } u^{(i)} \xrightarrow{p^{(i)}} u \xleftarrow{p^{(j)}} u^{(j)} \\ \text{P}_\text{S}\text{P}_\text{O}[p^{(j)}][p^{(i)}], & \text{if } u^{(i)} \xrightarrow{p^{(i)}} u \xrightarrow{p^{(j)}} u^{(j)} \\ \text{P}_\text{S}\text{P}_\text{O}[p^{(i)}][p^{(j)}], & \text{if } u^{(i)} \xleftarrow{p^{(i)}} u \xleftarrow{p^{(j)}} u^{(j)} \\ \text{P}_\text{S}\text{P}_\text{S}[p^{(i)}][p^{(j)}], & \text{if } u^{(i)} \xleftarrow{p^{(i)}} u \xrightarrow{p^{(j)}} u^{(j)} \end{cases}$$

Note that $C(u)$ can be computed by the following intersection

$$\bigcap_{p^{(i)} \in N(u) \ \wedge \ p^{(i)} \text{ is a constant}} \text{P}(p^{(i)}), \tag{4}$$

so both $|\text{P}(p^{(i)})|$ and Eq (3) give a tight upper bound of $|C(u)|$.

To illustrate, consider Figure 4. Since ?Z has no constant neighbors, and is the subject of three triple patterns with constant predicates 2 (ad), 3 (tc) and 5 (ri), respectively, it belongs to Case 2.2. Based on Eq (3), for the graph in Figure 3, we have

$$\begin{aligned} |C(?\text{Z})| &= \min\{|PS[2] \cap PS[3]|, |PS[2] \cap PS[5]|, \\ &\qquad |PS[3] \cap PS[5]|\} \\ &= \min\left\{|\{\text{Jack}, \text{Mary}\}|, |\{\text{Mary}\}|, |\{\text{Mary}\}|\right\} \\ &= \min\left\{P_S P_S[2][3], P_S P_S[2][5], P_S P_S[3][5]\right\} \ = \ 1. \end{aligned}$$

Since estimating $|C(u)|$ in Case 1 and Case 2.1 only involves list cardinality lookups from S-P-O permutation indices, and estimating $|C(u)|$ in Case 2.2 only involves number lookups from the precomputed PP matrices, the computation of $|C(u)|$ for all variables u in a SPARQL query is fast, so is the subsequent query vertex ordering utilizing these candidate-size estimates. Once the query vertex u_1 with the smallest candidate-size estimate is determined, however, we need to compute $C(u_1)$ using Eq (4) which requires intersecting candidate lists from PS and/or PO, since our PP matrices only keep list lengths rather than the lists themselves to be space efficient.

Query Variable Ordering. So far, we only ordered node variables in GP of the query, but there may exist predicates in GP that are also variables. Let π_Q be the sequence of variables (including both nodes and predicates) currently selected. The goal of query variable ordering is to determine which variable in GP should be appended to π_Q among those that are not already in π_Q, and this step is repeated until all variables in GP are added to π_Q.

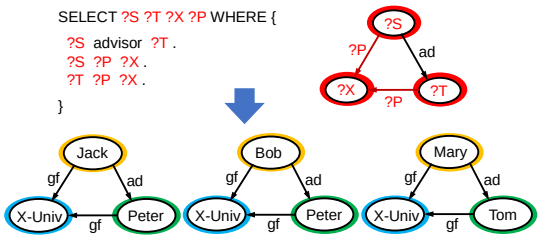


Fig. 6. A Sample Query with Predicate Variables

Recall that we denote the selected node variable order for matching by $\pi_k = [u_1, u_2, \ldots, u_k]$. To obtain π_Q from π_k, we revise our query vertex ordering approach so that as soon as we select the i^{th} node variable u_i and append it to variable sequence π_Q, we immediately append all its adjacent predicate variables that are not already in π_Q to π_Q. As an illustration, consider the query in Figure 6, where a predicate variable ?P appears in two triple patterns in GP. We select the vertex matching order as [?T, ?S, ?X] since $C(?T) = \text{PO}[\text{'ad'}] = \{\text{'Peter'}, \text{'Tom'}\}$, $C(?S) = \text{PS}[\text{'ad'}] = \{\text{'Jack'}, \text{'Bob'}, \text{'Mary'}\}$, and $?X$ is unrestricted and can match any RDF data node during graph exploration. Right after ?T is appended to π_Q, we append its adjacent predicate variable ?P to π_Q; we then append ?S, and since its adjacent predicate variable ?P is already in π_Q (though from a different triple pattern), we do not append ?P again, and finally append ?X to obtain $\pi_Q = [?T, ?P, ?S, ?X]$.

Our approach guarantees that when we match a node variable u_k, all the predicate variables adjacent to u_i, $i = 1, 2. \cdots, k-1$ have been matched, which is important to effectively narrow down the search space during graph exploration, especially for queries where a predicate variable appears in multiple triple patterns such as the one in Figure 6. To illustrate, let us denote the current match during graph exploration by $\mathcal{M}$. Assume that we matched 3 node variables in Figure 6 so $\mathcal{M} = [\langle ?T, \text{'Peter'}\rangle, \langle ?P, \text{'gf'}\rangle, \langle ?S, \text{'Bob'}\rangle]$. Then when matching the next node variable ?X using adjacent triple patterns (?S, ?P, ?X) and (?T, ?P, ?X), since ?T, ?P, ?S $\in \mathcal{M}$, we can compute local candidates of $?X$ as SPO['Bob']['gf'] $\cap$ SPO['Peter']['gf']. In contrast, if ?P is not already matched (right after ?T is matched), we cannot utilize these two triple patterns for pruning.

So far, we have assumed that matching begins from a node variable u_1, meaning that we need to compute $C(u_1)$, which requires set intersections (c.f. Eq (2) and Eq (4)). We can actually avoid computing $C(u_1)$ when there exist a node in GP that is a constant and has adjacent predicate variables. In this case, we first add to $\mathcal{M}$ all those predicate variables with S or O being a constant in GP, and then select u_1 only among their S and O with the minimum $|C(u)|$.

To illustrate, let us revise the query in Figure 5 by changing the edge '5: ri' to be a predicate variable ?P, then ?P will be the first element of π_Q, and even when $|C(?X)| < |C(?Z)|$, we will select $u_1 = ?Z$ to be the second element of π_Q since we can avoid computing $C(?X)$, and during graph exploration when ?P has been matched, e.g., to 'ri,' we can directly get ?Z's candidates as OPS['IR']['ri'].

Algorithm 1 Graph Exploration: Recursion Version

```
1:  Denote π_Q = [q_1, q_2, ..., q_k], initialize M ← []
2:  recursive_match(M, 1, π_Q, t_cur)
    Procedure recursive_match(M, i, π_Q, t_0)
3:      if i = 1 and q_1 is a node then LC(q_i) ← C(u_1)
    //q_1 = u_1
4:      else compute local candidates of q_i as LC(q_i)
5:      for each v ∈ LC(q_i)
6:          append M with ⟨q_i, v⟩
7:          if |M| = k then emit M
8:          else if t_cur − t_0 < τ_time then
9:              recursive_match(M, i + 1, π_Q, t_0)
10:         else create task ⟨M, i + 1, π_Q⟩ and add it to task
    queue
11:         pop (q_i, v) from M
    Task ⟨M, i, π_Q⟩
12:     recursive_match(M, i, π_Q, t_cur)
```

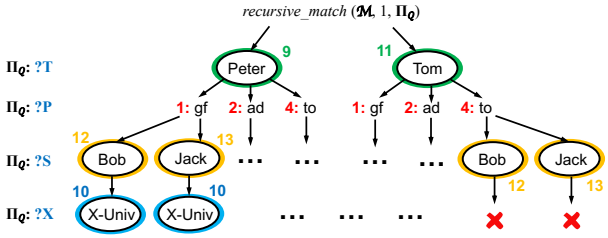


Fig. 7. Illustration of the Recursion Tree of Algorithm 1

Query Startup Stage. We call the 3 steps of (1) candidate-size estimation, (2) query variable ordering and (3) initial candidate computation (i.e., computing $C(u_1)$ when u_1 is the first element of π_Q) as the startup stage, after which graph exploration begins.

C. Graph Exploration

After the query startup stage finishes, we obtain a variable matching order $\pi_Q = \{q_1, q_2, \ldots, q_k\}$ and start matching by calling *recursive_match* $(\mathcal{M}, 1, \pi_Q, t_{cur})$ as shown in Line 2 of Algorithm 1. Let us ignore the contents in red in Algorithm 1 for now; they are needed for load balancing in parallel execution (see Appendix C [6]).

Here, procedure *recursive_match* $(\mathcal{M}, i, \pi_Q)$ (Lines 3–11) is an Ullmann-style recursive algorithm that assumes $q_1, q_2, \ldots, q_{i-1}$ have been matched as tracked by $\mathcal{M}$, and continues graph exploration from the next variable q_i to return all subgraph matches.

Figure 7 shows the recursion tree of Algorithm 1 for the query in Figure 6. We illustrate by first considering the leftmost matching path of the recursion tree starting from $q_1 =$?T, and steps (i)–(iv) below are illustrated in Figure 8.

Initially, **(i)** the recursion root obtains $LC(?T) = \{\text{'Peter'}, \text{'Tom'}\}$ in Line 3. Line 5 selects the first candidate 'Peter' as v, and Line 6 sets $\mathcal{M} = [\langle ?T, \text{'Peter'}\rangle]$. Then, Line 9 recurses to check $q_2 =$?P.

(ii) In this recursive call, Line 4 computes $LC(?P) = \text{SPO}[\text{'Peter'}] = \{1, 2, 4\}$ (see Figure 3). Here, we use SPO

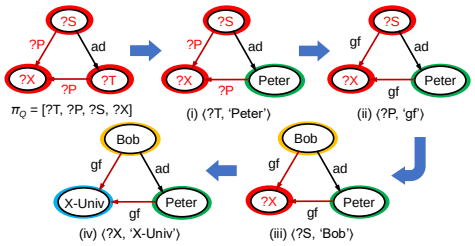


Fig. 8. Recursion Process for the Leftmost Matching Path

to obtain only the predicates, which is straightforward as illustrated in Figure 3: for example, we can obtain the outbound predicates of 7 from SPO as $\{1, 2, 3\}$. Line 5 then selects the first candidate 1 ('gf') as v, and Line 6 sets $\mathcal{M} = [\langle$?T, 'Peter'$\rangle, \langle$?P, 'gf'$\rangle]$. Subsequently, Line 9 recurses to check $q_3 =$?S.

(iii) In this recursive call, Line 4 computes $LC(?S) =$ OPS['Peter']['ad'] $=$ {'Bob', 'Jack'}, Line 5 selects 'Bob' as v, and Line 6 sets $\mathcal{M} = [\langle$?T, 'Peter'$\rangle, \langle$?P, 'gf'$\rangle, \langle$?S, 'Bob'$\rangle]$. Subsequently, Line 9 recurses to check $q_4 =$?X.

(iv) In this recursive call, Line 4 computes $LC(?X) =$ SPO['Peter']['gf'] $\cap$ SPO['Bob']['gf'] $=$ {'X-Univ'}, Line 5 selects 'X-Univ' as v, and Line 6 sets $\mathcal{M} = [\langle$?T, 'Peter'$\rangle, \langle$?P, 'gf'$\rangle, \langle$?S, 'Bob'$\rangle, \langle$?X, 'X-Univ'$\rangle]$. Since $|\mathcal{M}| = 4 = |\pi_Q|$, Line 7 outputs $\mathcal{M}$ as a valid match, and pops $\langle$?X, 'X-Univ'$\rangle$ from $\mathcal{M}$.

(v) The recursion then backtracks upwards for two layers, and then continue to check the next candidate 'Jack' $\in LC(?S)$ in Line 5.

(vi) On the rightmost matching path, the partial match $\mathcal{M} = [\langle$?T, 'Tom'$\rangle, \langle$?P, 'to'$\rangle, \langle$?S, 'Jack'$\rangle]$ is finally pruned when matching ?X, since SPO['Tom']['to'] $\cap$ SPO['Jack']['to'] $= \emptyset$.

By default, $LC(q_i)$ needs to exclude those vertices already matched to $q_1, q_2, \ldots, q_{i-1}$, to make sure that the same node in the RDF graph is not matched to more than one variable in the query. This isomorphism-based matching semantic is the most widely recognized. If we relax this constraint by not filtering previously matched candidates from $LC(q_i)$, then we obtain the homomorphism-based matching semantic, a variant described in [17].

Local Candidate Computation. So far, we have not explained how the local candidate set $LC(q_i)$ is computed. Unlike an initial candidate set $C(u)$ of a node variable u, local candidate set $LC(q_i)$ can be much tighter since Algorithm 1 ensures that all adjacent variables q_j with $j < i$ have been matched in $\mathcal{M}$, which allows additional pruning. Specifically, when q_i is a node variable u_j, we compute

$$LC(u_j) = \left(\bigcap_{(u_{j'},\ p) \in N(u_j)\ \wedge\ j' < j} \Pi\left(\mathcal{M}[p], \mathcal{M}[u_{j'}]\right) \right) \cap \left(\bigcap_{(u,\ p) \in N(u_j)\ \wedge\ u \text{ is constant}} \Pi\left(\mathcal{M}[p], u\right) \right), \quad (5)$$

where $N(u_j)$ is the set of PO and PS pairs adjacent to u_j, $\Pi(p, u)$ was defined in Eq (1) on Page 4, and $\mathcal{M}[q_i]$ denotes the match to q_i in $\mathcal{M}$ (i.e., $\langle q_i, \mathcal{M}[q_i]\rangle \in \mathcal{M}$). Intuitively, Eq (5) intersects candidate lists obtained by looking up indices SPO or OPS with two groups of adjacent query nodes: (1) matched node variables $u_{j'}$ ($\mathcal{M}[p]$ must exist since p gets matched right after $u_{j'}$); (2) constants u ($\mathcal{M}[p]$ must exist since such p are first added to π_Q before u_1).

When q_i is a predicate variable p, and let its immediate predecessor node variable in π_Q be u, then if u is a subject (resp. object) to p, then we compute $LC(p) =$ SPO$[\mathcal{M}[u]]$ (resp. OPS$[\mathcal{M}[u]]$).

Optimizations. Since set intersections are heavily used during graph exploration to compute $LC(u)$ as well as when computing $C(u_1)$, we leverage single-instruction-multiple-data (SIMD) parallelism supported by modern processors to speed up joins (AVX2 expands integer commands to 256 bits), as previously adopted and shown success in [15], [51]. We adopt the hybrid set intersection [51]: if the cardinalities of two sets are similar, we use the merge-based method; otherwise, we adopt the Galloping algorithm [15]. The intersection cost is proportional to cardinality of the smallest set.

As an optional optimization, right before appending $\langle q_i, v\rangle$ to $\mathcal{M}$ in Line 6 of Algorithm 1, if q_i is a node variable and the bitwise OR of $sig(q_i)$ and $sig(v)$ is not equal to $sig(v)$, we can prune v and call **continue** to return to Line 5 to check the next candidate. We call this technique as hash-based pruning.

While Algorithm 1 is conceptually simple as a recursive function, it is inefficient: since we only do light computation including computing $LC(q_i)$, the overhead of recursive function calls dominate in our tests. To overcome this problem, we translate the recursive function of Algorithm 1 into an equivalent iterative algorithm formulation that is more efficient to execute. Appendix B [6] describes this iterative algorithm formulation in more detail.

IV. Experiments

Following the TLAT model, we have implemented the T-RDF system that uses the above algorithm for parallel query evaluation. In T-RDF, inter-query parallelism is straightforward where different queries can be evaluated independently and concurrently by different threads. However, some queries can be 'heavy' and hence long-running if using only one thread. Recall Figure 7 on Page 5 which shows the recursion tree of Algorithm 1. Different recursion subtrees under the same node are independent, so they can be individual tasks concurrently evaluated by different threads. Following this idea, T-RDF supports intra-query parallelism by only letting each subtree-searching task run recursively for a period of at most τ_{time}, beyond which subsequent backtracking search will wrap each traversed tree node as an independent task to be added back to the system, rather than processing the node's subtree recursively by the current thread. The additional logic is shown by the red lines in Algorithm 1. This timeout mechanism prevents straggler tasks by decomposing long-running tasks. T-RDF also has system designs to ensure bounded memory use by only allowing a bounded pool of queries and their associated tasks in memory at any time, and adopts a lineage-based task tracking method to determine when all tasks

TABLE I
DATASETS

Dataset	# Triples	# Entities	# Predicates	# Literals
YAGO-2.3.0	112,8247,05	10,123,050	96	44,228,041
YAGO-2.5.3	244,799,610	76,887,759	104	30,949,685
LUBM-1000	138,318,414	21,735,144	18	11,170,026
LUBM-2000	306,839,240	48,222,479	18	24,782,794
WatDiv	108,994,714	52,12,745	86	5,038,202
BSBM	86,805,724	12,943,339	40	7,830,243

of a query have been evaluated. Due to page limit, the details on T-RDF system design are given in Appendix C [6].

We now evaluate T-RDF and compare it with state-of-the-art SPARQL engines. Our programs were run on a server with two Intel Xeon E5-2680 v4 CPU @ 2.40 GHz. Our system code is released at [7].

Datasets and Queries. Table I summarizes the datasets we use. Following prior works, we test the programs using a real knowledge base YAGO [12], synthetic datasets generated by the Lehigh University Benchmark (LUBM) [9], the Waterloo SPARQL Diversity Test Suite (WatDiv) [5] and the Berlin SPARQL Benchmark (BSBM) [22]. YAGO extracts facts from Wikipedia and integrates them with the WordNet thesaurus, and we use two versions YAGO 2 [10] (YAGO-2.3.0) and YAGO 2s [11] (YAGO-2.5.3). We also generate two datasets of different sizes using the LUBM data generator with 1000 and 2000 universities, respectively. WatDiv and BSBM are the datasets that are also used in [47]. As Table I shows, the number of predicates is small so our PP matrices consume negligible space.

For each dataset shown in Table I, we use 12 queries of different shapes (following [23], [24]), workloads, and variable/constant settings for evaluation to cover a diverse range of SPARQL queries, which we made available at [1]. Appendix D provides more details where Figure 13 illustrates some queries used for LUMB-1000 and -2000, and for WatDiv. The queries are selected to have different shapes including chain, tree, cycle and their combination. The queries with acyclic patterns (chain and tree) have the heaviest workload since they are the least selective, followed by those with cycles (cycle and combined) which have medium workloads. We also consider queries with a constant node (so are the most selective) and a variable predicate to allow query diversity.

Comparison with Existing RDF Engines. Table II shows the performance of two variants of T-RDF, one without hash-based pruning (default setting) and one with (recall that hash-based pruning was an optimization technique described at the end of Section III-C). The performance is also compared against five state-of-the-art RDF engines MAGiQ [35], Wukong [48], gStore [2], TripleBit [74] and RDF-3X [43] for all the queries on all the four datasets shown in Table I. Among them, MAGiQ [35] is a representative

TABLE II
SYSTEM COMPARISON (TIME UNIT: MS)

		Cycle		Chain		Tree		Combine		Constant		Var Predicate	
		Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2
YAGO-2.3.0	T-RDF	86	45	660	1349	113	541	100	764	33	43	234	112
	T-RDF w/ hash	44	95	753	2709	143	625	126	814	26	40	290	204
	MAGiQ	5308	6455	6518	2831	3923	8013	4089	2806	462	1284	N/A	N/A
	Wukong	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	N/A	N/A
	gStore	100	121	808	5624	861	768	424	8258	579	595	OOT	OOT
	TripleBit	Empty	Empty	Empty	Empty	Empty	Empty	Empty	Empty	Empty	Empty	Empty	Empty
	RDF-3X	596	3143	5937	15862	4192	5072	1448	7825	71	70	2417	4969
YAGO-2.5.3	T-RDF	42	74	1922	14	195	21	780	102	32	18	84	49
	T-RDF w/ hash	32	84	2491	35	228	23	997	251	35	16	122	80
	MAGiQ	12,388	14,605	14,056	9027	13,981	14,204	18,370	20,386	1019	933	N/A	N/A
	Wukong	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	N/A	N/A
	gStore	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail
	TripleBit	265	Empty	Empty	162	Empty	Empty	Empty	3364	59	64	Empty	Empty
	RDF-3X	1281	7365	19,105	4731	12,131	1208	7722	5286	66	390	1084	2309
LUBM-1000	T-RDF	488	1283	1453	408	364	645	1323	118	36	15	73	78
	T-RDF w/ hash	573	1732	1952	630	487	1276	1752	123	30	19	85	92
	MAGiQ	15,624	14,583	15,494	8466	12,041	15,183	14,262	20,208	3983	3875	N/A	N/A
	Wukong	11,943	6313	19,458	5959	15,601	6490	5896	36,277	1963	1961	N/A	N/A
	gStore	261,297	9797	22,093	5976	263,715	5739	4861	9224	708	654	219	110
	TripleBit	1740	17,063	30,530	7196	1320	5872	6554	7841	426	193	Empty	Empty
	RDF-3X	154,729	99,059	90,516	70,947	29,147	100,613	101,732	33,316	189	191	337	216
LUBM-2000	T-RDF	398	1570	2618	1990	421	1179	1941	58	20	18	89	104
	T-RDF w/ hash	524	2753	4359	2912	821	2235	3328	194	22	20	112	147
	MAGiQ	45,371	29,259	32,515	16,758	27,877	31,335	34,411	50,966	7768	7703	N/A	N/A
	Wukong	40,345	11,643	35,792	11,023	57,823	11,969	11,967	72,254	4096	4104	N/A	N/A
	gStore	475,886	18,856	41,856	10,545	509,325	11,848	8222	15,234	1038	1002	492	251
	TripleBit	2486	7215	62,662	13,859	2955	12,145	12,420	1915	60	20	Empty	Empty
	RDF-3X	422,992	216,376	340,922	217,904	71,041	213,457	221,614	42,862	549	889	1004	1973
WatDiv	T-RDF	129	6	484	887	53	1462	656	792	12	11	16	16
	T-RDF w/ hash	163	14	531	1459	42	2193	834	839	13	8	13	17
	MAGiQ	1111	1136	1707	2152	1302	4643	1249	1708	460	276	N/A	N/A
	Wukong	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	N/A	N/A
	gStore	160	10	2543	2375	70	8820	712	3328	1945	743	641	372
	TripleBit	311	8	1710	Empty	186	18,174	286	Empty	174	84	385	53
	RDF-3X	633	461	54,031	24,441	125	59,755	755	4050	528	165	1564	1664
BSBM	T-RDF	429	369	452	335	285	258	286	331	22	15	24	15
	T-RDF w/ hash	885	380	470	386	326	328	318	392	14	12	30	14
	MAGiQ	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	N/A	N/A
	Wukong	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	N/A	N/A
	gStore	434	384	530	410	325	264	337	354	514	165	201	119
	TripleBit	853	1027	573	803	642	421	499	475	28	311	78	62
	RDF-3X	23,823	25,143	2551	3016	2084	3038	1918	1887	47	44	104	70

* **Note:** N/A = Var Predicate Not Supported; Fail = Program Crashed; Empty = Empty (Wrong) Result Returned; OOT = Running for More Than 20 Minutes and Cut

TABLE III
EFFECT OF TIMEOUT THRESHOLD τ_{time} ON RUNNING TIME (MS)

		Cycle		Chain		Tree		Combine		Constant		Var Predicate	
	τ_{time}	Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2
LUBM-1000	10	1211	1775	12,939	5174	700	1271	1771	135	21	19	80	92
	100	544	1699	12,488	4526	451	1264	1702	123	20	18	73	78
	1000	1409	2313	14,289	6852	1256	2256	2569	137	23	20	124	142
WatDiv	10	118	11	456	1241	36	7364	166	2036	5	4	16	16
	100	116	10	411	1247	35	7303	174	1364	5	4	16	16
	1000	1080	10	563	1513	56	7781	368	1382	5	4	20	25

system using sparse matrix algebra, RDF-3X [43] is a representative system using relational joins, gStore [2] is a representative system using subgraph matching and hash-based pruning, and TripleBit [74] is a representative system using compressed bit-matrix structure to improve storage compactness and querying efficiency. Wukong [48] is a representative in-memory engine using advanced key-value store and concurrent query processing design.

In our experiments, we use only 1 machine to evaluate Wukong's performance, though it also utilizes fast network design to support efficient distributed computing. TripleBit reported significant performance improvement over the other systems [74] as also confirmed by our experiments on LUBM datasets, but we found that their implementation is not stable and behave incorrectly for some queries by returning empty results. MAGiQ only copes with integer IDs so we need to convert all strings in the RDF graph and the queries into integer IDs as preprocessing. This makes MAGiQ difficult to use, though it is memory-efficient since strings are not stored at all. We use the version of MAGiQ built on top of GraphBLAS [4]. T-RDF is run with 16 threads, and the timeout threshold τ_{time} is set as 100 milliseconds which works generally well.

As Table II shows, T-RDF is consistently the fastest among all the systems on all the datasets for all the queries. Moreover, the default setting without hash-based pruning works generally better, though 'T-RDF w/ hash' shows some slight advantage on queries with light workloads. This shows that hash-based pruning is not effective and does not worth its overhead in general, especially for queries with medium to heavy workloads. MAGiQ and Wukong are orders of magnitude slower than T-RDF, and they do not support queries with variable predicates. Moreover, Wukong crashes on all datasets other than LUMB during graph loading, likely due to implementation issues. gStore and RDF-3X can be even much slower, such as on LUBM-2000. Also, gStore crashes when building the VS-tree index on YAGO-2.5.3, and runs out of time ($>$ 20 min) on YAGO-2.3.0 for queries with a variable predicate. TripleBit exhibits acceptable performance but is still much slower than T-RDF (e.g., on the LUMB datasets). However, it is not robust and return empty results (which is wrong) on many queries, such as all the queries on YAGO-2.3.0. In summary, T-RDF is the most efficient and robust system.

Effect of Timeout Threshold. Recall that T-RDF sets $\tau_{time} = 100$ ms by default. This is tuned to strike a balance between enough parallelism and the overhead of task creation and scheduling. Table III shows the the query processing time of T-RDF (due to space limit, we only show results on LUBM-1000 and WatDiv which are representative), when we vary τ_{time} as 10 ms, 100 ms and 1000 ms. We can see that $\tau_{time} = 100$ ms gives the best performance overall.

TABLE IV
EFFECT OF TIMEOUT THRESHOLD τ_{time} ON NUMBER OF CREATED TASKS

		Cycle		Chain		Tree		Combine		Constant		Var Predicate	
	τ_{time}	Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2
LUBM-1000	10	389,816	597,703	598,698	42,999	513,161	556,099	597,884	925	915	902	269	385
	100	3460	578,061	594,042	8011	12,284	79,388	582,698	160	1	1	1	1
	1000	2861	399,084	534,891	6748	2092	17,836	473,873	1	1	1	1	1
WatDiv	10	3211	1	22,704	45,816	6054	590,296	10,274	952,754	1	1	1	1
	100	1676	1	7904	32,015	1	490,918	7549	638,843	1	1	1	1
	1000	1229	1	1	1	1	90,242	1	1	1	1	1	1

TABLE V
INTER-QUERY PARALLELISM (TIME UNIT: MS)

	# of Queries	T-RDF Batch	T-RDF Serial	MAGiQ	Wukong	gStore	TripleBit	RDF-3X
LUBM-1000	10	5096	6133	94,154	92,452	504,775	57,101	460,714
	50	27,524	30,665	409,131	OOM	OOM	141,911	648,908
	100	55,972	61,330	509,973	OOM	OOM	192,448	865,638
BSBM	10	371	2782	FAIL	FAIL	1626	1729	20,083
	50	1780	13,910	FAIL	FAIL	OOM	6144	35,137
	100	2364	27,820	FAIL	FAIL	OOM	7689	52,743

* Note: OOM = Out of Memory

We also report the total number of created tasks (due to task decomposition) for each query in Table IV when τ_{time} varies. We can see that for queries with heavier workloads, a smaller τ_{time} creates many more tasks which improves load balancing but also incurs task creation and scheduling costs. As Table III has shown, $\tau_{time} = 100$ strikes a good balance between these two factors. In contrast, for light queries, often only the root task is created and since it finishes too soon, no task decomposition has happened.

Inter-Query Parallelism. Recall from Table II that each dataset has 12 queries, the first 10 of which do not contain variable predicates. Since not all the RDF engines support variable predicates, to allow comparison among all systems, we define the first 10 queries of each dataset as a query mini-batch. Note that each mini-batch contains a mix of queries of different types.

To test the inter-query parallelism, we repeat each mini-batch for once, 5 times and 10 times to create query batches of 10, 50 and 100 queries, respectively. For T-RDF, we consider two versions: 'T-RDF Batch' starts evaluating all queries simultaneously, and 'T-RDF Serial' evaluates the queries one after another. For all the other engines, we start programs to evaluate all queries simultaneously.

Table V reports the running time of answering the query batches, where due to space limit, we only show results on LUMB-1000 and BSBM. The complete results on all datasets can be found in Appendix E [6]. We can see that when queries are evaluated simultaneously, T-RDF is significantly faster than all the other engines, which may crash or run out of memory except for the two disk-based systems RDF-3X and TripleBit. RDF-3X is often the slowest, while TripleBit exhibits good performance though still not competitive to 'T-RDF Batch'. Also, 'T-RDF Batch' shows favorable performance compared with 'T-RDF Serial' where queries are evaluated one by one,

TABLE VI
QUERY STARTUP TIME (TIME UNIT: MS)

		Cycle		Chain		Tree		Combine		Constant		Var Predicate	
		Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2
LUBM-1000	Ours	18	27	35	50	52	26	31	45	26	22	5	4
	gStore	1056	1177	473	128	782	1055	919	158	30	24	36	23
	Mem Intersect	288	598	581	161	285	628	905	447	29	24	23	26
	Disk intersect	311	667	642	244	336	701	1849	547	33	25	59	53
BSBM	Ours	4	3	3	3	4	3	3	4	3	4	3	2
	gStore	15	13	29	28	15	16	14	10	22	52	13	22
	Mem Intersect	35	25	26	24	31	32	29	34	4	6	28	34
	Disk intersect	35	27	27	26	35	32	32	39	4	7	33	35

TABLE VII
MEMORY COST (UNIT: GB)

	YAGO-2.3.0	YAGO-2.5.3	LUBM-1000	LUBM-2000	WatDiv	BSBM
T-RDF	32.5	63.6	31.6	62.9	11.9	27.1
MAGiQ	3.6	10.7	5.7	12	3.7	3.8
Wukong	Fail	Fail	68.3	77	Fail	Fail
gStore	59	32	39.7	69.5	14.8	31.1
TripleBit	1.4	9.1	0.5	1.6	0.4	0.4
RDF-3X	1	5.5	1.5	2.6	0.3	0.3

especially on BSBM. This verifies the effectiveness of T-RDF's system design as shown in Figure 12 with active query list and task queues. Note that even in 'T-RDF Serial', each query is still evaluated with intra-query parallelism support, but it may not be able to fully utilize the CPU resources so having a pool of active queries at the same time is beneficial in utilizing the otherwise used CPU cores.

Query Startup Cost. Recall from Page 5 that each query has a startup stage before graph exploration, consisting of 3 steps: (1) candidate-size estimation, (2) query variable ordering and (3) $C(u_1)$ computation by set intersections. Our approach is efficient since candidate-size estimation only involves index lookups and taking minimum over result cardinalities, and we use a SIMD set intersection algorithm. We compare with a few baselines for the startup stage: (1) gStore that uses VS-tree to compute the candidates $C(u)$ for every query node variable u [78], (2) looking up disk-based indices to compute $C(u)$ by set intersections, and (3) looking up in-memory indices to compute $C(u)$ by set intersections.

Table VI reports the query startup time where due to space limit, we only show results on LUMB-1000 and BSBM. The complete results on all datasets can be found in Appendix F [6]. Comparing Table II with Table VI, we can see that the startup cost is a very small fraction of the query processing time. Also, the startup cost of our approach is a clear winner, much faster than all solutions including gStore. In fact, in gStore, since VS-tree uses hash-based signatures for pruning, the candidates can contain many false positives, leading to a much larger size of $C(u)$ than other methods, which can further increase the time of the subsequent graph exploration.

Memory Cost. We notice that the major portion of memory cost is contributed by graph loading and index construction, i.e., the startup stage. The additional memory cost caused by each query is much smaller. Therefore, we focus on measuring the memory consumption right after the startup stage for all engines, and the results are reported in Table VII. We can see that the memory cost of T-RDF is smaller than Wukong and gStore. MAGiQ is even more memory-efficient than T-RDF, but this is because it only operates on integer IDs (so does not store strings in memory) and relies on users to preprocess strings into IDs. TripleBit and RDF-3X use the least memory because they are disk-based systems. However, as Table II has shown, MAGiQ, TripleBit and RDF-3X are much slower than T-RDF despite using less memory.

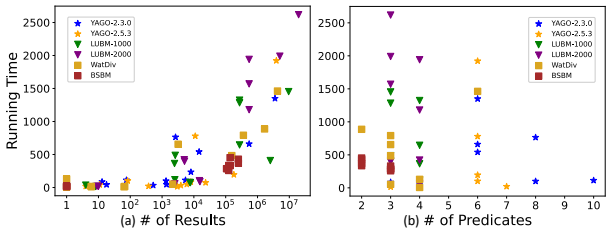


Fig. 9. Scatter Plots: Time v.s. Query Features

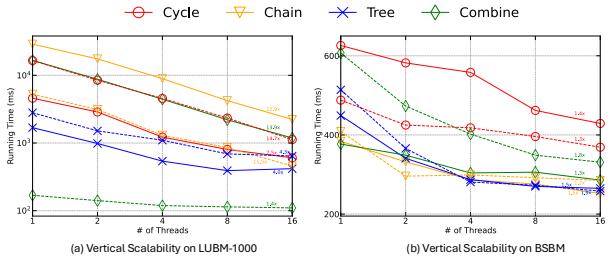


Fig. 10. Scalability (Time Unit: ms)

Running Time v.s. Query Features. Figure 9(a) (resp. Figure 9(b)) shows the running time of all our tested queries w.r.t. the # of results (resp. the # of predicates). We can see a clear trend of increasing running time when the number of results increases, which is intuitive since more results imply more searching workloads. In contrast, we cannot observe a clear relationship between running time and the # of predicates: in fact, the data points in Figure 9(b) are more clustered based on datasets, showing that the running time is more dataset-related than query-related.

Scalability of Intra-Query Parallelism. Figure 10 shows the processing time curve of T-RDF for the 8 queries with different shapes on LUMB-1000 and BSBM, where solid and dashed curves are for the two different queries of the same shape. Due to space limit, the results for all datasets can be found in Appendix G [6]. The speedup ratio is annotated at the end of each curve, and we can see that the ratio is up to 14.7× with 16 threads. This ratio is often close to the ideal ratio for queries running beyond 1 second (i.e., 10^3 ms) when running with a single thread.

V. CONCLUSION

We proposed an efficient SPARQL query engine called T-RDF, which relies on well-designed indices and Ullmann-style subgraph matching (in contrast to joins) and supports task-based intra- and inter-query parallelism. Optimization techniques were explored including eliminating recursive function calls, SIMD set intersections, and hash-based pruning.

REFERENCES

[1] Adopted Queries. https://github.com/lyuheng/T-RDF/tree/main/Query.

[2] gStore Project on GitHub. https://github.com/pkumod/gStore.
[3] gStore Website. https://www.gstore.cn/pcsite/index.html.
[4] MAGiQ GitHub Repo. https://github.com/fjamour/MAGiQ.
[5] SWAT Projects - the Lehigh University Benchmark (LUBM). http://swat.cse.lehigh.edu/projects/lubm/.
[6] T-RDF Appendix. https://github.com/lyuheng/T-RDF/blob/main/appendix.pdf.
[7] T-RDF GitHub Repo. https://github.com/lyuheng/T-RDF/.
[8] T-thinker as CCC Great Innovative Idea. https://cra.org/ccc/great-innovative-ideas/t-thinker-a-task-centric-framework-to-revolutionize-big-data-systems-research/.
[9] WatDiv query templates. http://dsg.uwaterloo.ca/watdiv/.
[10] YAGO 2. https://yago-knowledge.org/downloads/yago-2.
[11] YAGO 2s. https://yago-knowledge.org/downloads/yago-2s.
[12] YAGO Website. https://yago-knowledge.org/.
[13] I. Abdelaziz, R. Harbi, S. Salihoglu, and P. Kalnis. Combining vertex-centric graph processing with SPARQL for large-scale RDF data analytics. *IEEE Trans. Parallel Distributed Syst.*, 28(12):3374–3388, 2017.
[14] I. Abdelaziz, R. Harbi, S. Salihoglu, P. Kalnis, and N. Mamoulis. Spartex: A vertex-centric framework for RDF data analytics. *Proc. VLDB Endow.*, 8(12):1880–1883, 2015.
[15] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *SIGMOD Conference*, pages 431–446. ACM, 2016.
[16] W. Ali, M. Saleem, B. Yao, A. Hogan, and A. N. Ngomo. A survey of RDF stores & SPARQL engines for querying knowledge graphs. *VLDB J.*, 31(3):1–26, 2022.
[17] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, 2017.
[18] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for RDF data. In *WWW*, pages 41–50. ACM, 2010.
[19] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. Dbpedia: A nucleus for a web of open data. In *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer, 2007.
[20] D. Banerjee, P. A. Nair, J. N. Kaur, R. Usbeck, and C. Biemann. Modern baselines for SPARQL semantic parsing. In *SIGIR*, pages 2260–2265. ACM, 2022.
[21] F. Belleau, M. Nolin, N. Tourigny, P. Rigault, and J. Morissette. Bio2rdf: Towards a mashup to build bioinformatics knowledge systems. *J. Biomed. Informatics*, 41(5):706–716, 2008.
[22] C. Bizer and A. Schultz. The berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.
[23] A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. *Proc. VLDB Endow.*, 11(2):149–161, 2017.
[24] A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. *VLDB J.*, 29(2-3):655–679, 2020.
[25] J. Cheng, D. Yan, X. Hao, and W. Ng. Mining order-preserving submatrices under data uncertainty: A possible-world approach. In *ICDE*, pages 1154–1165. IEEE, 2019.
[26] U. Deppisch. S-tree: A dynamic balanced signature index for office retrieval. In *SIGIR*, pages 77–87. ACM, 1986.
[27] D. Dosso and G. Silvello. A scalable virtual document-based keyword search system for RDF datasets. In *SIGIR*, pages 965–968. ACM, 2019.
[28] G. Fu, C. R. Batchelor, M. Dumontier, J. Hastings, E. L. Willighagen, and E. Bolton. Pubchemrdf: towards the semantic annotation of pubchem compound and substance databases. *J. Cheminformatics*, 7:34:1–34:15, 2015.
[29] G. Guo, D. Yan, M. T. Özsu, Z. Jiang, and J. Khalil. Scalable mining of maximal quasi-cliques: An algorithm-system codesign approach. *Proc. VLDB Endow.*, 14(4):573–585, 2020.
[30] G. Guo, D. Yan, L. Yuan, J. Khalil, C. Long, Z. Jiang, and Y. Zhou. Maximal directed quasi-clique mining. In *ICDE*, pages 1900–1913. IEEE, 2022.
[31] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. Triad: a distributed shared-nothing RDF engine based on asynchronous message passing. In *SIGMOD Conference*, pages 289–300. ACM, 2014.
[32] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57. ACM Press, 1984.
[33] W. Han, J. Lee, and J. Lee. Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In K. A. Ross, D. Srivastava, and D. Papadias, editors, *SIGMOD*, pages 337–348. ACM, 2013.
[34] S. Hu, L. Zou, J. X. Yu, H. Wang, and D. Zhao. Answering natural language questions by subgraph matching over knowledge graphs. *IEEE Trans. Knowl. Data Eng.*, 30(5):824–837, 2018.
[35] F. T. Jamour, I. Abdelaziz, Y. Chen, and P. Kalnis. Matrix algebra framework for portable, scalable and efficient query engines for RDF graphs. In *EuroSys*, pages 27:1–27:15. ACM, 2019.
[36] F. T. Jamour, I. Abdelaziz, and P. Kalnis. A demonstration of magiq: Matrix algebra approach for solving RDF graph queries. *Proc. VLDB Endow.*, 11(12):1978–1981, 2018.
[37] X. Jian, Y. Wang, X. Lei, L. Zheng, and L. Chen. SPARQL rewriting: Towards desired results. In *SIGMOD Conference*, pages 1979–1993. ACM, 2020.
[38] G. Jiang, Q. Zhou, T. Jin, B. Li, Y. Zhao, Y. Li, and J. Cheng. Vsgm: view-based gpu-accelerated subgraph matching on large graphs. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 739–753. IEEE Computer Society, 2022.
[39] J. Khalil, D. Yan, G. Guo, and L. Yuan. Parallel mining of large maximal quasi-cliques. *VLDB J.*, 31(4):649–674, 2022.
[40] J. Kim, H. Shin, W. Han, S. Hong, and H. Chafi. Taming subgraph isomorphism for RDF query processing. *Proc. VLDB Endow.*, 8(11):1238–1249, 2015.
[41] T. Lin, Q. Chen, G. Cheng, A. Soylu, B. Ell, R. Zhao, Q. Shi, X. Wang, Y. Gu, and E. Kharlamov. ACORDAR: A test collection for ad hoc content-based (RDF) dataset retrieval. In *SIGIR*, pages 2981–2991. ACM, 2022.
[42] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In A. K. Elmagarmid and D. Agrawal, editors, *SIGMOD*, pages 135–146. ACM, 2010.
[43] T. Neumann and G. Weikum. RDF-3X: a risc-style engine for RDF. *Proc. VLDB Endow.*, 1(1):647–659, 2008.
[44] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.
[45] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H2RDF: adaptive query processing on RDF data in the cloud. In *WWW*, pages 397–400. ACM, 2012.
[46] W. Qu, D. Yan, G. Guo, X. Wang, L. Zou, and Y. Zhou. Parallel mining of frequent subtree patterns. In *Software Foundations for Data Interoperability and Large Scale Graph Data Analytics - 4th International Workshop, SFDI 2020, and 2nd International Workshop, LSGDA 2020, held in Conjunction with VLDB 2020, Tokyo, Japan, September 4, 2020, Proceedings*, volume 1281 of *Communications in Computer and Information Science*, pages 18–32. Springer, 2020.
[47] M. Saleem, G. Szárnyas, F. Conrads, S. A. C. Bukhari, Q. Mehmood, and A. N. Ngomo. How representative is a SPARQL benchmark? an analysis of RDF triplestore benchmarks. In L. Liu, R. W. White, A. Mantrach, F. Silvestri, J. J. McAuley, R. Baeza-Yates, and L. Zia, editors, *WWW*, pages 1623–1633. ACM, 2019.
[48] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In K. Keeton and T. Roscoe, editors, *OSDI*, pages 317–332. USENIX Association, 2016.
[49] S. Sun and Q. Luo. Parallelizing recursive backtracking based subgraph matching on a single machine. In *24th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2018, Singapore, December 11-13, 2018*, pages 42–50. IEEE, 2018.
[50] S. Sun and Q. Luo. Scaling up subgraph query processing with efficient subgraph matching. In *ICDE*, pages 220–231. IEEE, 2019.
[51] S. Sun and Q. Luo. In-memory subgraph matching: An in-depth study. In *SIGMOD Conference*, pages 1083–1098. ACM, 2020.
[52] S. Sun and Q. Luo. Subgraph matching with effective matching order and indexing. *IEEE Trans. Knowl. Data Eng.*, 34(1):491–505, 2022.
[53] S. Sun, X. Sun, Y. Che, Q. Luo, and B. He. Rapidmatch: A holistic approach to subgraph query processing. *Proc. VLDB Endow.*, 14(2):176–188, 2020.
[54] S. Sun, X. Sun, B. He, and Q. Luo. Rapidflow: An efficient approach to continuous subgraph matching. *Proc. VLDB Endow.*, 15(11):2415–2427, 2022.
[55] X. Sun, S. Sun, Q. Luo, and B. He. An in-depth study of continuous subgraph matching. *Proc. VLDB Endow.*, 15(7):1403–1416, 2022.

[56] K. Theocharidis, J. Liagouris, N. Mamoulis, P. Bouros, and M. Terrovitis. SRX: efficient management of spatial RDF data. *VLDB J.*, 28(5):703–733, 2019.

[57] H. Tran, L. Phan, J. T. Anibal, B. T. Nguyen, and T. Nguyen. SPBERT: an efficient pre-training BERT on SPARQL queries for question answering over knowledge graphs. In *Neural Information Processing - 28th International Conference, ICONIP 2021, Sanur, Bali, Indonesia, December 8-12, 2021, Proceedings, Part I*, volume 13108 of *Lecture Notes in Computer Science*, pages 512–523. Springer, 2021.

[58] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[59] C. Unger, L. Bühmann, J. Lehmann, A. N. Ngomo, D. Gerber, and P. Cimiano. Template-based question answering over RDF data. In *WWW*, pages 639–648. ACM, 2012.

[60] H. Vargas, C. B. Aranda, and A. Hogan. RDF explorer: A visual query builder for semantic web knowledge graphs. In *Proceedings of the ISWC 2019 Satellite Tracks (Posters & Demonstrations, Industry, and Outrageous Ideas) co-located with 18th International Semantic Web Conference (ISWC 2019), Auckland, New Zealand, October 26-30, 2019*, volume 2456 of *CEUR Workshop Proceedings*, pages 229–232. CEUR-WS.org, 2019.

[61] S. Wang, C. Lou, R. Chen, and H. Chen. Fast and concurrent RDF queries using rdma-assisted GPU graph exploration. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 651–664. USENIX Association, 2018.

[62] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, 2008.

[63] W. Wu, H. Li, H. Wang, and K. Q. Zhu. Probase: a probabilistic taxonomy for text understanding. In *SIGMOD Conference*, pages 481–492. ACM, 2012.

[64] M. Yahya, K. Berberich, S. Elbassuoni, M. Ramanath, V. Tresp, and G. Weikum. Natural language questions for the web of data. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, EMNLP-CoNLL 2012, July 12-14, 2012, Jeju Island, Korea*, pages 379–390. ACL, 2012.

[65] D. Yan, M. M. R. Chowdhury, G. Guo, J. Kahlil, Z. Jiang, and S. K. Prasad. Distributed task-based training of tree models. In *ICDE*, pages 2237–2249. IEEE, 2022.

[66] D. Yan, G. Guo, M. M. R. Chowdhury, M. T. Özsu, W. Ku, and J. C. S. Lui. G-thinker: A distributed framework for mining subgraphs in a big graph. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1369–1380. IEEE, 2020.

[67] D. Yan, G. Guo, M. M. R. Chowdhury, M. T. Özsu, J. C. S. Lui, and W. Tan. T-thinker: a task-centric distributed framework for compute-intensive divide-and-conquer algorithms. In *PPoPP*, pages 411–412. ACM, 2019.

[68] D. Yan, G. Guo, J. Khalil, M. T. Özsu, W. Ku, and J. C. S. Lui. G-thinker: a general distributed framework for finding qualified subgraphs in a big graph with load balancing. *VLDB J.*, 31(2):287–320, 2022.

[69] D. Yan, W. Qu, G. Guo, and X. Wang. Prefixfpm: A parallel framework for general-purpose frequent pattern mining. In *ICDE*, pages 1938–1941. IEEE, 2020.

[70] D. Yan, W. Qu, G. Guo, X. Wang, and Y. Zhou. Prefixfpm: a parallel framework for general-purpose mining of frequent and closed patterns. *VLDB J.*, 31(2):253–286, 2022.

[71] L. Yuan, G. Guo, D. Yan, S. Adhikari, J. Khalil, C. Long, and L. Zou. G-thinkerq: A general subgraph querying system with a unified task-based programming model. *IEEE Trans. Knowl. Data Eng.*, 37(6):3429–3444, 2025.

[72] L. Yuan, D. Yan, J. Han, A. Ahmad, Y. Zhou, and Z. Jiang. Faster depth-first subgraph matching on gpus. In *ICDE*. IEEE Computer Society, 2024.

[73] L. Yuan, D. Yan, W. Qu, S. Adhikari, J. Khalil, C. Long, and X. Wang. T-FSM: A task-based system for massively parallel frequent subgraph pattern mining from a big graph. *Proc. ACM Manag. Data*, 1(1):74:1–74:26, 2023.

[74] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. Triplebit: a fast and compact system for large scale RDF data. *Proc. VLDB Endow.*, 6(7):517–528, 2013.

[75] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. *Proc. VLDB Endow.*, 6(4):265–276, 2013.

[76] Z. Zhang, Y. Lu, W. Zheng, and X. Lin. A comprehensive survey and experimental study of subgraph matching: Trends, unbiasedness, and interaction. *Proc. ACM Manag. Data*, 2(1):60:1–60:29, 2024.

[77] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gstore: Answering SPARQL queries via subgraph matching. *Proc. VLDB Endow.*, 4(8):482–493, 2011.

[78] L. Zou, M. T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao. gstore: a graph-based SPARQL query engine. *VLDB J.*, 23(4):565–590, 2014.

APPENDIX A
RELATED WORK

RDF Query Engines. Most existing SPARQL query engines leverage triple join operations to process SPARQL queries. For example, the query in Figure 2 has three triple patterns with predicates 'to,' 'tc' and 'ad,' respectively. Each triple pattern generates a temporary query table with bindings by scanning the triples in the RDF data, and these query tables are then joined to produce the final results. To avoid scanning all triples, it is common to use an exhaustive indexing method that creates a full set of subject(S)–predicate(P)–object(O) permutations. For example, RDF-3X [43], [44] builds clustered B^+-trees on all six (S, P, O) permutations – (SPO, SOP, PSO, POS, OSP, OPS), and Hexastore [62] adopts a similar indexing approach. BitMat [18] and TripleBit [74] further represent RDF triples by compressed bit-matrix structure to improve storage compactness and querying efficiency especially for "heavy" queries that are complex and with low selectivity. While the systems so far are centralized, TriAD [31] instead targets a distributed shared-nothing setting. To improve performance, TriAD uses RDF graph summarization to facilitate join-ahead pruning, and supports asynchronous joins to reduce synchronization overhead. SRX [56] extends RDF-3X with support for spatial queries including range selections, spatial joins, and k nearest neighbors. An encoding scheme is designed to approximate the geometries of the RDF entities inside their integer IDs, which can be used to accelerate queries with spatial predicates and to re-encode entities upon updates.

Since join-based approaches may generate large redundant intermediate results, Trinity.RDF [75] proposes to overcome this problem by graph exploration. Specifically, the RDF data is stored in a native graph model on top of a distributed in-memory key-value store, and graph exploration is used for query evaluation with one-step pruning to reduce intermediate results. However, a final centralized join is still needed to filter out non-matching results, which can be a performance bottleneck especially for queries with cycles and/or large intermediate results as noticed by [31], [45]. Wukong [48] proposes to use graph exploration with full-history pruning (instead of the one-step pruning of Trinity.RDF) to further reduce the intermediate results and to eliminate the final join, which achieves better performance. However, Wukong focuses on RDMA-based distributed graph exploration and mainly on inter-query parallelism, rather than the design of efficient graph exploration algorithm, and the exploitation of intra-query parallelism with load balancing which are a focus of our T-RDF system. Wukong+G [61] further extends Wukong to support the offloading of "heavy" queries to GPUs, while in contrast, our T-RDF system accelerates "heavy" queries without requiring any specialized hardware accelerators.

gStore [77], [78] adopts a pure graph exploration (subgraph matching) approach to answer SPARQL queries, with interesting search space pruning techniques. The first technique is to create a signature for each data vertex v in the RDF graph by hashing the strings of v and all its adjacent predicates and neighbors in one hop to a bitmap, denoted by $sig(v)$, to facilitate the filtering of non-matching candidates for a query variable. For each query vertex u in the GP, we can do a similar bitmap hashing on those non-variable strings in u's one hop to obtain its signature, denoted by $sig(u)$. If any 1-bit in $sig(u)$ is 0 in $sig(v)$ (i.e., the bitwise OR of $sig(u)$ and $sig(v)$ is not equal to $sig(v)$), v cannot be a candidate of u since some constant predicate or subject/object in u's one hop is missing in v's one hop. The second technique is to replace all strings in the RDF graph with signatures to create a signature graph in memory for pruning, where vertex signatures are organized by an index structure called VS-tree to facilitate candidate filtering. VS-tree extends the S-tree [26] for bitmap packing and indexing, which is similar to R-tree [32] but replaces Euclidean distance between points by Hamming distance between signatures. Since upper-levels of a VS-tree are summary graphs of the RDF graph, VS-tree additionally adds edges between the "supernodes" in the summary graphs along with their "superedge" signatures to improve pruning effectiveness.

Our empirical tests found that the candidates obtained from the VS-tree are not always tight, which may unnecessarily increase the cost of the subsequent graph exploration (subgraph matching) search, so we adopt a different approach to be presented in Section III. However, the signature-based candidate pruning is effective in reducing search space during graph exploration and is thus adopted.

SPARTex [13], [14] further builds its RDF query engine on existing vertex-centric graph processing frameworks pioneered by Pregel [42]. It implements a generic SPARQL operator as a vertex-centric program, and also supports user-defined procedure (UDP) to implement vertex-centric graph algorithms (e.g., PageRank, SSSP). Graph algorithms can dynamically maintain their computation results in-memory as vertex attributes, which can be supplied as input for SPARQL and vice-versa in a pipelined fashion.

Besides join-based and graph exploration approaches, MAGiQ [35], [36] explores sparse matrix algebra as a third paradigm for RDF query engine design. MAGiQ represents the RDF graph as a sparse matrix and defines a domain-specific language of algebraic operations. SPARQL queries are translated into matrix algebra programs consisting of these algebraic operations, and existing matrix algebra libraries are called for execution. In MAGiQ, the strings in the RDF graph are converted into numerical IDs as a preprocessing step so that query evaluation is memory-efficient. However, SPARQL queries also need to be converted into numerical IDs first, which creates additional overhead. Moreover, MAGiQ translates queries into matrix algebra programs by DFS walk where a forward edge (i.e., just discovered) creates the candidates of the next node variable, which are pruned again when backtracking the edge; this edge-driven pruning strategy (join-based) is much more expensive than node-driven pruning (intersection-based) as in our T-RDF.

We refer interested readers to [16] for a more comprehensive review of the existing RDF query engines, including

their storage, indexing, query processing, and graph partitioning techniques. Saleem et al. [47] conducted a fine-grained comparative analysis of existing evaluation benchmarks from different aspects including the structuredness and relationship specialty of datasets, and the diversity of query features. It identifies factors that impact query execution time and the result sizes, such as data structuredness and the number of projection variables.

Surprisingly, there are few SPARQL query engines proposed after gStore [77], [78] and Wukong [48], and more recently, the related works mainly focus on making RDF query formulation more accessible to non-experts (rather than efficient query execution). Specifically, [20], [57], [59], [64] study the generation of SPARQL queries from natural language questions to search knowledge graphs, while [34] searches a knowledge graph by efficient top-k approximate graph matching without generating a SPARQL query first. Among other works, [27] studies keyword search over RDF datasets, [37] studies how to help a SPARQL user to rewrite a query so that it produces desired results, [60] studies how to help users incrementally build a query by visual graph navigation to search knowledge graphs, and [41] studies content-based, as opposed to metadata-based, ad hoc RDF dataset retrieval.

With the recent breakthroughs in (1) faster algorithms for subgraph matching, and (2) the think-like-a-task (TLAT) parallel computing model (both to be reviewed next), we believe it is the right timing to revisit and improve the design of SPARQL query engines to be more efficient via (parallel) subgraph matching. Notably, $\text{Turbo}_{\text{HOM}}$++ [40] also explored the use subgraph matching to answer SPARQL queries with NUMA-aware parallelism, but it is based on an older subgraph matching algorithm $\text{Turbo}_{\text{ISO}}$ [33] that is less efficient [51], hence unlikely to be competitive to our T-RDF. Moreover, $\text{Turbo}_{\text{HOM}}$++ is not open-sourced.

Subgraph Matching. As mentioned in Section I, there are many recent breakthroughs in subgraph matching. We focus on [51] here which provides an in-depth study of subgraph matching algorithms that utilize graph exploration rather than joins. Here, graph exploration is implemented by Ullmann's backtracking algorithm [58] (see Section III for more details), but different algorithms adopt different optimization techniques such as the method of filtering candidate vertices in the data graph, and the method of ordering query vertices. Sun and Luo [51] put 8 such algorithms in a unified framework with 3 stages: (1) candidate filtering for query vertices, (2) query vertex ordering, and (3) data graph exploration following the query vertex matching order using Ullmann-style enumeration. The best optimization technique(s) in each stage is identified with extensive experiments to obtain the best combination.

Sun and Luo [51] find that it is most favorable to construct an auxiliary index to maintain edges between query-vertex candidates, and to adopt efficient set intersection for local candidate computation. However, "light" SPARQL queries with high selectivity are common and for such queries, we find that creating the auxiliary index itself is more expensive than a direct graph exploration, so we do not adopt this method in T-RDF. We do adopt the techniques recommended by [51] for query vertex ordering and efficient set intersection for local candidate computation which are critical for the performance of graph exploration, and we create simple indices during preprocessing to facilitate lightweight candidate size estimation, which helps find a good query vertex matching order.

Also note that Proposition 3.3 of [53] has shown that for subgraph matching, the time complexity of Ullmann-style backtracking search gives a time complexity equivalent to the worst-case optimal join, better than previous methods that rely on binary joins. Although the efficiency is still NP-hard, the effective search space pruning techniques summarized by [51] makes the computation practically efficient.

Think-Like-a-Task (TLAT) Parallelism. The TLAT model was originally proposed in [8], [67] as an abstract computing model for parallelizing compute-intensive problems. Then, G-thinker [66], [68] was developed as the first truly CPU-scalable distributed TLAT framework for parallelizing subgraph finding problems including subgraph matching, beating prior systems by up to two orders of magnitude in speed, and scaling to graphs two orders of magnitude larger. Among the other TLAT systems, PrefixFPM [25], [46], [69], [70] is a task-centric system to mine frequent patterns (sequences, subgraphs, subtrees and matrices) in transactional settings. TreeServer [65] is a task-centric system to train models based on many decision trees. T-FSM [73] is a task-centric system for frequent subgraph pattern mining in a big graph. T-DFS [72] is a task-centric system for subgraph matching on GPUs.

While sharing the TLAT model, T-RDF is different from the above TLAT systems since those systems target one-time offline analytics, while we target online querying where we need to consider how to manage the task evaluation of multiple queries posed by users online. We remark that while T-RDF is currently implemented as a parallel prototype for a single machine, it can be easily extended for distributed processing following a similar design as in G-thinker (where subgraph matching is already a well-tested application), by maintaining vertices of an input graph in a distributed key-value store for tasks to request the necessary data.

APPENDIX B
GRAPH EXPLORATION: THE ITERATIVE VERSION

Algorithm 2 shows the iterative version of graph exploration, which is equivalent to the recursive Algorithm 1 in logic, but does not suffer from the overheads of recursive function calls. Specifically, let i_{cur} be the current level in the recursion tree of Figure 7 on Page 5, then we maintain two arrays up-to-date at any time:

1) $\mathbf{LC}[.]$ where $LC[i]$ keeps $LC(q_i)$, for $i = 1, 2, \cdots, i_{cur}$.
2) $\mathbf{idx}[.]$ where $idx[i]$ keeps the position of q_i's current match in the candidate list $LC[i]$.

Algorithm 2 essentially implements a depth-first search over the recursion tree of Figure 7. In Algorithm 2, whenever a candidate in $LC(q_i)$ is accessed, Line 9 advances the

Algorithm 2 Graph Exploration: Iterative Version

1: Allocate thread-local arrays $idx[]$, $LC[]$ by the current thread
2: $\mathcal{M} \leftarrow []$, *match*$(\mathcal{M}, 1, \pi_Q, t_{cur})$
Procedure *match*$(\mathcal{M}, i_{root}, \pi_Q, t_0)$
3: $i \leftarrow i_{root}$
4: **if** $i = 1$ **and** q_1 is a node **then** $LC[i] \leftarrow C(u_1)$ //$q_1 = u_1$
5: **else** compute local candidates of q_i as $LC[i]$
6: $idx[i] \leftarrow 1$
7: **repeat**
8: **while** $idx[i] < |LC[i]|$ **do**
9: $v \leftarrow LC[i][idx[i]]$, $idx[i] \mathrel{+}= 1$
10: append $\mathcal{M}$ with $\langle q_i, v \rangle$
11: **if** $|\mathcal{M}| = k$ **then** emit $\mathcal{M}$
12: **else if** $t_{cur} - t_0 < \tau_{time}$ **then**
13: $i \mathrel{+}= 1$
14: compute local candidates of q_i as $LC[i]$, $idx[i] \leftarrow 1$
15: **else** create task $\langle \mathcal{M}, i+1, \pi_Q \rangle$ and add to queue
16: $i \mathrel{-}= 1$
17: **if** $i < i_{root}$ **then break**
Task $\langle \mathcal{M}, i, \pi_Q \rangle$
18: *match*$(\mathcal{M}, i, \pi_Q, t_{cur})$

position to the next element for access in the next iteration. Moreover, instead of recursion, Line 13 advances the layer, and Line 14 prepares candidates for the new layer, and rewinds the candidate position to 1. If all candidates in a layer have been checked, Line 16 returns to the previous layer so that Line 8 continues to check the next candidate.

As an optional final optimization, right before appending $\langle q_i, v \rangle$ to $\mathcal{M}$ in Line 10, if q_i is a node variable and the bitwise OR of $sig(q_i)$ and $sig(v)$ is not equal to $sig(v)$, we can prune v and call **continue** to return to Line 8 to check the next candidate. We call this technique as hash-based pruning.

Algorithm 2 also supports timeout-based task decomposition to eliminate straggler tasks, as shown by the red content in Algorithm 2. Moreover, Lines 3 and 17 are added to make sure that a task only backtracks to its entry layer (i.e., the layer in the recursion tree where the task is created to process the subtree). Note that each computing thread maintains its local arrays $idx[]$ and $LC[]$ to process its assigned tasks for subgraph matching by backtracking.

Appendix C
Task-Based Parallel Query Engine

Task-Based Parallel Model. We first describe our model for inter- and intra-query parallelism. Inter-query parallelism is straightforward since the evaluations of different queries are independent, so can be processed concurrently by different computing threads.

However, some queries can be 'heavy' with significantly more computing workloads than an average query, so processing an entire 'heavy' query by one thread leads to the straggler

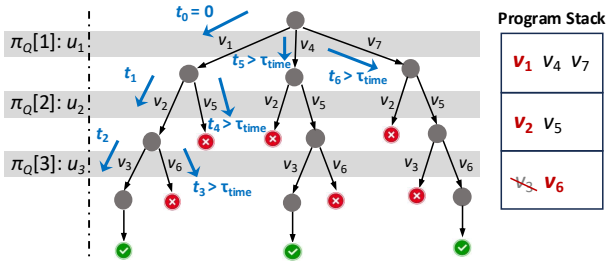


Fig. 11. Illustration of Timeout Decomposition

problem. It is, thus, desirable to decompose its workloads into independent units, called tasks, for concurrent processing by multiple threads.

Recall from Figure 7 and Algorithm 1 on Page 5 that the evaluation of a query Q is essentially enumerating the path $\mathcal{M}$ matching π_Q along a recursion tree. Therefore, different (non-overlapping) subtree branches of the recursion tree in Figure 7 can be treated as independent tasks for concurrent processing.

Based on this idea, we define a task $\langle \mathcal{M}, i, \pi_Q \rangle$ as the call of *recursive_match*$(\mathcal{M}, i, \pi_Q)$ as specified in Algorithm 1 (in fact, the iterative version *match*$(\mathcal{M}, i, \pi_Q)$ described by Algorithm 2 in Appendix B [6] is called instead). Then, for example, the two subtrees rooted at node 9 (Peter) and 11 (Tom) in Figure 7 correspond to two independent tasks $\langle \mathcal{M} = [\langle$?T, 'Peter'$\rangle], 2, \pi_Q \rangle$ and $\langle \mathcal{M} = [\langle$?T, 'Tom'$\rangle], 2, \pi_Q \rangle$, respectively. Initially, each query Q corresponds to one task $\langle \mathcal{M} = [], 1, \pi_Q \rangle$ as specified in Line 2 of Algorithm 1, which we call as the root task of Q.

Timeout-Based Task Decomposition. As [48] indicates, to achieve a high throughput, it is favorable to only use intra-query parallelism for 'heavy' queries.

Following this idea, we let each query create only a root task running *recursive_match*$(\mathcal{M}, 1, \pi_Q, t_{cur})$, where t_{cur} is the current time when the function is called (we now also consider the content in red in Algorithm 1), i.e., the beginning time of the root task. Only if the running time of the root task exceeds a threshold τ_{time} will it be decomposed into smaller tasks to be added back to the T-RDF engine for concurrent processing, to eliminate straggler tasks. The logic related to this timeout-based task decomposition strategy is highlighted in red in Algorithm 1 (and equivalently, Algorithm 2 in Appendix B [6]), and we use Algorithm 1 for description for ease of understanding (though the non-recursive version Algorithm 2 is actually run). For ease of presentation, we simplify the recursion tree previously illustrated in Figure 7 on Page 5 (which is too wide to show completely) with a more abstract form shown in Figure 11 for illustration, where $\pi_Q = [u_1, u_2, u_3]$ and at each level i, candidates $\{v_j\}$ are matched to u_i one at a time.

Specifically, in Algorithm 1, when a task begins, it records the task starting time t_0 as the current time t_{cur}. This is shown in Line 2 for a root task, and for a task resulted from previous task decomposition (i.e., by Line 10), this is shown in Line 12.

Before recursion to the next layer, Line 8 checks if at least τ_{time} time has passed since t_0 (i.e., the current task has run for at least τ_{time} time), and if so, instead of recursion at Line 9, a

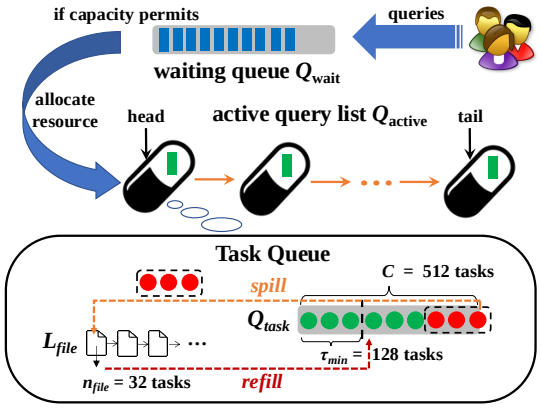


Fig. 12. T-RDF System Overview

new task $\langle \mathcal{M}, i+1, \pi_Q \rangle$ is created and added to the system by Line 10. Here, $\mathcal{M}$ needs to be deep-copied into the new task so that later when it is scheduled, it has access to $\mathcal{M}$ to run Line 12. The original task uses its $\mathcal{M}$ to backtrack to create more tasks for upper layers.

Figure 11 shows this process: when $\mathcal{M} = [\langle u_1, v_1 \rangle, \langle u_2, v_2 \rangle]$ and we are about to check $v_6 \in LC(u_3)$ ($v_3 \in LC(u_3)$ has been processed), t_3 times out so a task is created for checking this tree branch. During backtracking, three more tasks are created since t_4, t_5 and t_6 all time out; note that tasks are only decomposed to the proper layer.

System Overview. Figure 12 shows the system architecture of T-RDF. Specifically, user queries are submitted on demand and are first appended to a waiting queue Q_{wait}. T-RDF also maintains a list of active queries $q_1 \rightarrow q_2 \rightarrow \ldots \rightarrow q_k$ currently with tasks under evaluation, denoted by Q_{active}. To keep memory usage bounded, we design Q_{active} to have a limited capacity (at most 10 queries by default which works well in general). Note that the queries in Q_{wait} are just stored as query strings with negligible memory cost, while the queries in Q_{active} are maintained with their evaluation status and tasks under computation. To respect the time order of receiving queries, a computing thread always attempts to obtain the first available task starting from Q_{active}'s head for processing; and when it fails to get any task from Q_{active} and meanwhile, the capacity of Q_{active} permits (i.e., it has fewer than 10 active queries), the thread dequeues a query q from Q_{wait} to spawn a root task to process, which also moves q to Q_{active}'s tail.

Each active query in Q_{active} keeps its query status and a task queue Q_{task} as shown at the bottom part of Figure 12. We next explain how tasks are added to and fetched from Q_{task}. We require Q_{task} to have a maximum capacity of C ($C = 512$ by default). However, it is possible for a big task to generate many decomposed tasks to be inserted into Q_{task}, causing it to overflow. To keep the # of in-memory tasks bounded, if Q_{task} is full but a new task is to be inserted, we spill a batch of n_{file} tasks at the end of Q_{task} as a file to local disk to make room, where $n_{file} = 32$ by default. Note that tasks spilled from Q_{task} are written to the disk (and loaded back later) in batches of size n_{file} each, to achieve serial disk IO. We use a file list $\mathcal{L}_{file}$ to track those files spilled from Q_{task} to be loaded back to Q_{task} later when it needs a task refill (see below).

Each computing thread keeps fetching the first available task in Q_{active}, which may come from different queries so T-RDF achieves inter-query parallelism (the thread moves to the next query if Q_{task} of the current one is either empty of locked by another thread, which leads to a failed try-lock); meanwhile, each query may have many tasks so intra-query parallelism is utilized.

Whenever a computing thread that checks Q_{task} of a query in Q_{active} for task fetching finds that there are fewer than τ^{min} tasks in Q_{task} ($\tau^{min} = 128$ by default), it will take a task file from $\mathcal{L}_{file}$ of this query (if $\mathcal{L}_{file}$ is not empty) and refill its tasks into Q_{task}.

Since Q_{task} needs to be refilled from the head of the queue and to spill tasks from the tail by all computing threads, we implement Q_{task} as a deque protected by a mutex. Also, to spill tasks, a thread first locks Q_{task} and fetches n_{file} tasks at the tail of the queue, and then unlocks the queue so that other threads can access it; the thread then serializes the fetched tasks to a file and deletes those tasks from memory, without holding queue lock. The implementation of Q_{task} reuses the task container design in Section 5.3 of [39], where default parameters have been tuned to work well in general.

T-RDF is currently implemented as a parallel prototype for a single machine, but it can be easily extended for distributed processing as in G-thinker [66] by maintaining vertices of an RDF graph in a distributed key-value store for tasks to request the necessary data.

Query Progress Tracking. We now explain how T-RDF tracks when a query completes evaluation to return results to the user. Since a task may time out and be decomposed into many smaller (child) tasks, each of which may again decompose, forming a task lineage tree with a tree edge (t_{pa}, t_{cur}) if task t_{cur} is created by decomposing t_{pa}. Let us denote the number of (child) tasks generated by a task t_{pa} due to timeout by m; we track this lineage tree so that whenever a child task t_{cur} completes, it will increment a counter at t_{pa}, and if the counter equals m (i.e., a full counter indicating the completion of the entire subtree. under t_{pa}), we increment the counter at t_{pa}'s parent task. This upward process goes on, and if the root task of a query Q is reached with a full counter, Q is complete so it is removed from Q_{active} with results returned to the user.

APPENDIX D
QUERIES USED FOR EVALUATION

For each dataset, we use 12 queries of different shapes (following [23], [24]), workloads, and variable/constant settings for evaluation to cover a diverse range of SPARQL queries, which we made available at [1]. Figure 13 illustrates some queries used for LUMB-1000 and -2000, and for WatDiv. In particular, the first four queries have different shapes: chain, tree, cycle and their combination, where all nodes are variables, and all edges are constant predicates. The queries with acyclic patterns (chain and tree) have the heaviest workload since they are the least selective, followed by those with cycles

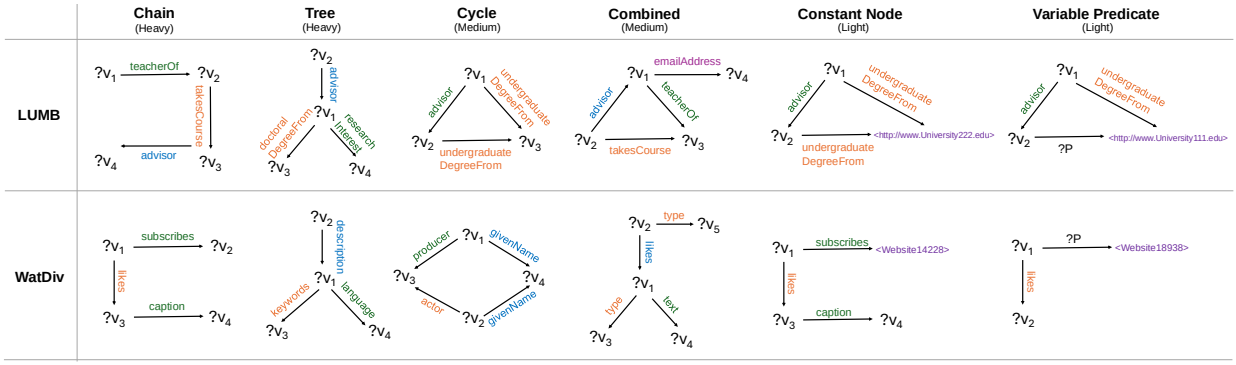


Fig. 13. Graph Diagrams of the SPARQL Queries for LUMB-1000/2000 and WatDiv

TABLE VIII
INTER-QUERY PARALLELISM (TIME UNIT: MS)

	# of Queries	T-RDF Batch	T-RDF Serial	MAGiQ	Wukong	gStore	TripleBit	RDF-3X
YAGO-2.3.0	10	3010	3734	38,183	FAIL	15139	EMPTY	47,401
	50	11,057	18,670	131,845	FAIL	OOM	EMPTY	125,293
	100	22,335	37,340	222,097	FAIL	OOM	EMPTY	143,887
YAGO-2.5.3	10	2980	3333	106,440	FAIL	FAIL	EMPTY	54,210
	50	12,262	16,665	393,702	FAIL	FAIL	EMPTY	192,371
	100	19,612	33,330	761,722	FAIL	FAIL	EMPTY	345,393
LUBM-1000	10	5096	6133	94,154	92,452	504,775	57,101	460,714
	50	27,524	30,665	409,131	OOM	OOM	141,911	648,908
	100	55,972	61,330	509,973	OOM	OOM	192,448	865,638
LUBM-2000	10	8590	10,406	259,165	243,152	1,052,744	96,810	1,381,573
	50	40,441	52,030	953,455	OOM	OOM	412,269	5,459,351
	100	86,137	104,060	1,858,184	OOM	OOM	740,660	9,075,862
WatDiv	10	1659	4492	7843	FAIL	18,518	*11,444	144,246
	50	12,523	22,460	28,608	FAIL	OOM	*56,583	189,047
	100	25,305	44,920	38,759	FAIL	OOM	*73,966	263,733
BSBM	10	371	2782	FAIL	FAIL	1626	1729	20,083
	50	1780	13,910	FAIL	FAIL	OOM	6144	35,137
	100	2364	27,820	FAIL	FAIL	OOM	7689	52,743

* Note: OOM = Out of Memory; * = Reporting Time Although Some Results Are Empty (Wrong)

(cycle and combined) which have medium workloads. The last two queries are light in workloads since they contain a constant node that is the most selective; the last query also contains a variable predicate to allow query diversity, which has a slightly higher workload. For each of the six query types, we use two queries for evaluation.

APPENDIX E
INTER-QUERY PARALLELISM

Recall from Table II that each dataset has 12 queries, the first 10 of which do not contain variable predicates. Since not all the RDF engines support variable predicates, to allow comparison among all systems, we define the first 10 queries of each dataset as a query mini-batch. Note that each mini-batch contains a mix of queries of different types.

To test the inter-query parallelism, we repeat each mini-batch for once, 5 times and 10 times to create query batches of 10, 50 and 100 queries, respectively. For T-RDF, we consider two versions: 'T-RDF Batch' starts evaluating all queries simultaneously, and 'T-RDF Serial' evaluates the queries one after another. For all the other engines, we start programs to evaluate all queries simultaneously.

Table VIII reports the running time of answering the query batches on all the datasets. We can see that when queries are evaluated simultaneously, T-RDF is significantly faster than all the other engines, which may crash or run out of memory except for the two disk-based systems RDF-3X and TripleBit. RDF-3X is often the slowest, while TripleBit exhibit good performance though still not competitive to 'T-RDF Batch'. Also, 'T-RDF Batch' shows favorable performance compared with 'T-RDF Serial' where queries are evaluated one by one, especially on BSBM. This verifies the effectiveness of T-RDF's system design as shown in Figure 12 with active query list and task queues.

APPENDIX F
QUERY STARTUP COST

Recall from Page 5 that each query has a startup stage before graph exploration, consisting of 3 steps: (1) candidate-size estimation, (2) query variable ordering and (3) $C(u_1)$ computation by set intersections. Our approach is efficient since candidate-size estimation only involves index lookups and taking minimum over result cardinalities, and we use a SIMD set intersection algorithm. We compare with a few baselines for the startup stage: (1) gStore that uses VS-tree to compute the candidates $C(u)$ for every query node variable u [78], (2) looking up disk-based indices to compute $C(u)$ by set intersections, and (3) looking up in-memory indices to compute $C(u)$ by set intersections.

Table IX reports the query startup time on all datasets. Comparing Table II with Table IX, we can see that the startup cost is a very small fraction of the query processing time. Also, the startup cost of our approach is a clear winner, much faster than all solutions including gStore. In fact, in gStore, since VS-tree uses hash-based signatures for pruning, the candidates can contain many false positives, leading to a much larger size of $C(u)$ than other methods, which can further increase the time of the subsequent graph exploration.

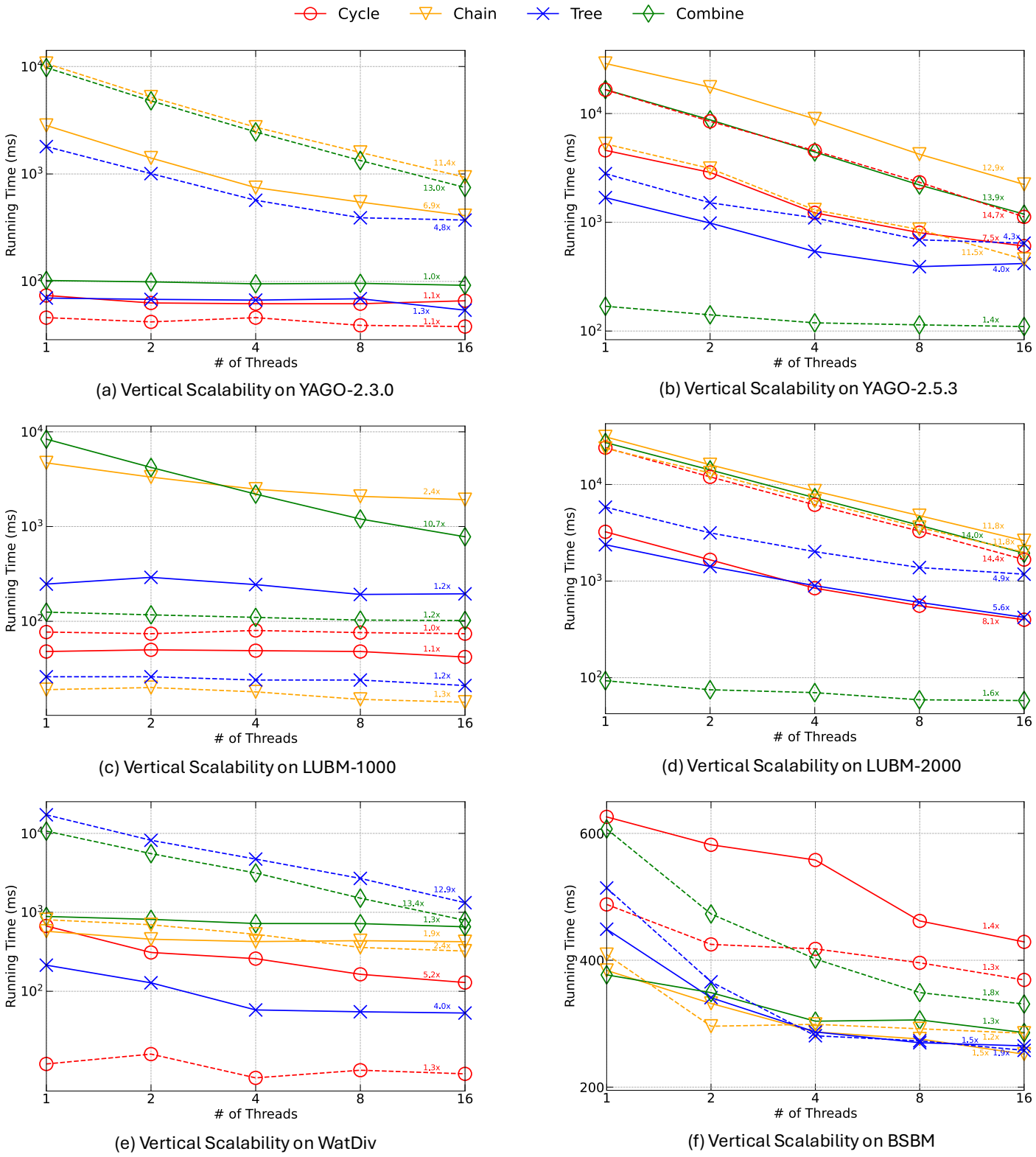
APPENDIX G
SCALABILITY OF INTRA-QUERY PARALLELISM

Figure 14 shows the processing time curve of T-RDF for the 8 queries with different shapes on all the datasets. The

TABLE IX
QUERY STARTUP TIME (TIME UNIT: MS)

		Cycle		Chain		Tree		Combine		Constant		Var Predicate	
		Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2
YAGO-2.3.0	Ours	**25**	**24**	**24**	**28**	**25**	**24**	**28**	**24**	**20**	**25**	**23**	**30**
	gStore	88	72	35	110	89	38	36	185	82	41	59	72
	Mem Intersect	44	33	83	134	124	84	125	191	99	100	102	139
	Disk intersect	51	37	94	174	149	89	155	228	121	122	152	195
YAGO-2.5.3	Ours	**30**	**24**	**8**	**27**	**27**	**30**	**14**	**23**	**22**	**21**	**13**	**20**
	gStore	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail
	Mem Intersect	59	45	27	185	105	63	109	182	82	112	90	126
	Disk intersect	80	55	42	204	160	90	142	254	144	151	174	203
LUBM-1000	Ours	**18**	**27**	**35**	**50**	**52**	**26**	**31**	**45**	**26**	**22**	**5**	**4**
	gStore	1056	1177	473	128	782	1055	919	158	30	24	36	23
	Mem Intersect	288	598	581	161	285	628	905	447	29	24	23	26
	Disk intersect	311	667	642	244	336	701	1849	547	33	25	59	53
LUBM-2000	Ours	**24**	**28**	**60**	**24**	**86**	**15**	**49**	**8**	**23**	**18**	**12**	**8**
	gStore	1478	1248	1092	598	1076	1264	1378	243	42	59	59	65
	Mem Intersect	371	624	892	183	349	535	992	302	34	29	80	82
	Disk intersect	425	792	1022	295	501	672	1692	510	44	33	104	192
WatDiv	Ours	**5**	**7**	**15**	**16**	**2**	**3**	**4**	**3**	**4**	**2**	**5**	**6**
	gStore	9	8	2437	2132	9	1687	16	209	12	11	10	14
	Mem Intersect	13	12	32	27	20	90	22	34	16	12	28	31
	Disk intersect	15	14	34	27	28	101	28	39	19	17	54	43
BSBM	Ours	**4**	**3**	**3**	**3**	**4**	**3**	**3**	**4**	**3**	**4**	**3**	**2**
	gStore	15	13	29	28	15	16	14	10	22	52	13	22
	Mem Intersect	35	25	26	24	31	32	29	34	4	6	28	34
	Disk intersect	35	27	27	26	35	32	32	39	4	7	33	35

speedup ratio is annotated at the end of each curve, and we can see that the ratio is up to 14.7× with 16 threads. This ratio is often close to the ideal ratio for queries running beyond 1 second (i.e., 10^3 ms) when running with a single thread.



(a) Vertical Scalability on YAGO-2.3.0

(b) Vertical Scalability on YAGO-2.5.3

(c) Vertical Scalability on LUBM-1000

(d) Vertical Scalability on LUBM-2000

(e) Vertical Scalability on WatDiv

(f) Vertical Scalability on BSBM

Fig. 14. Scalability (Time Unit: ms)