Algorithm 2 Graph Exploration: Iterative Version

```
1:  Allocate thread-local arrays idx[], LC[] by the current thread
2:  M ← [], match(M, 1, π_Q, t_cur)
    Procedure match(M, i_root, π_Q, t_0)
3:    i ← i_root
4:    if i = 1 and q_1 is a node then LC[i] ← C(u_1)   //q_1 = u_1
5:    else compute local candidates of q_i as LC[i]
6:    idx[i] ← 1
7:    repeat
8:      while idx[i] < |LC[i]| do
9:        v ← LC[i][idx[i]], idx[i] += 1
10:       append M with ⟨q_i, v⟩
11:       if |M| = k then emit M
12:       else if t_cur − t_0 < τ_time then
13:         i += 1
14:         compute local candidates of q_i as LC[i], idx[i] ← 1
15:       else create task ⟨M, i + 1, π_Q⟩ and add to queue
16:     i −= 1
17:     if i < i_root then break
    Task ⟨M, i, π_Q⟩
18:   match(M, i, π_Q, t_cur)
```

A 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 6, then we maintain two arrays up-to-date at any time:

(1) **LC**[.] where $LC[i]$ keeps $LC(q_i)$, for $i = 1, 2, \cdots, i_{cur}$.
(2) **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 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.

Table 8: 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)

B INTER-QUERY PARALLELISM

Recall from Table 2 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 5 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 10 with active query list and task queues.

C QUERY STARTUP COST

Recall from Page 6 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

Table 9: Query Startup Time (Time Unit: ms)

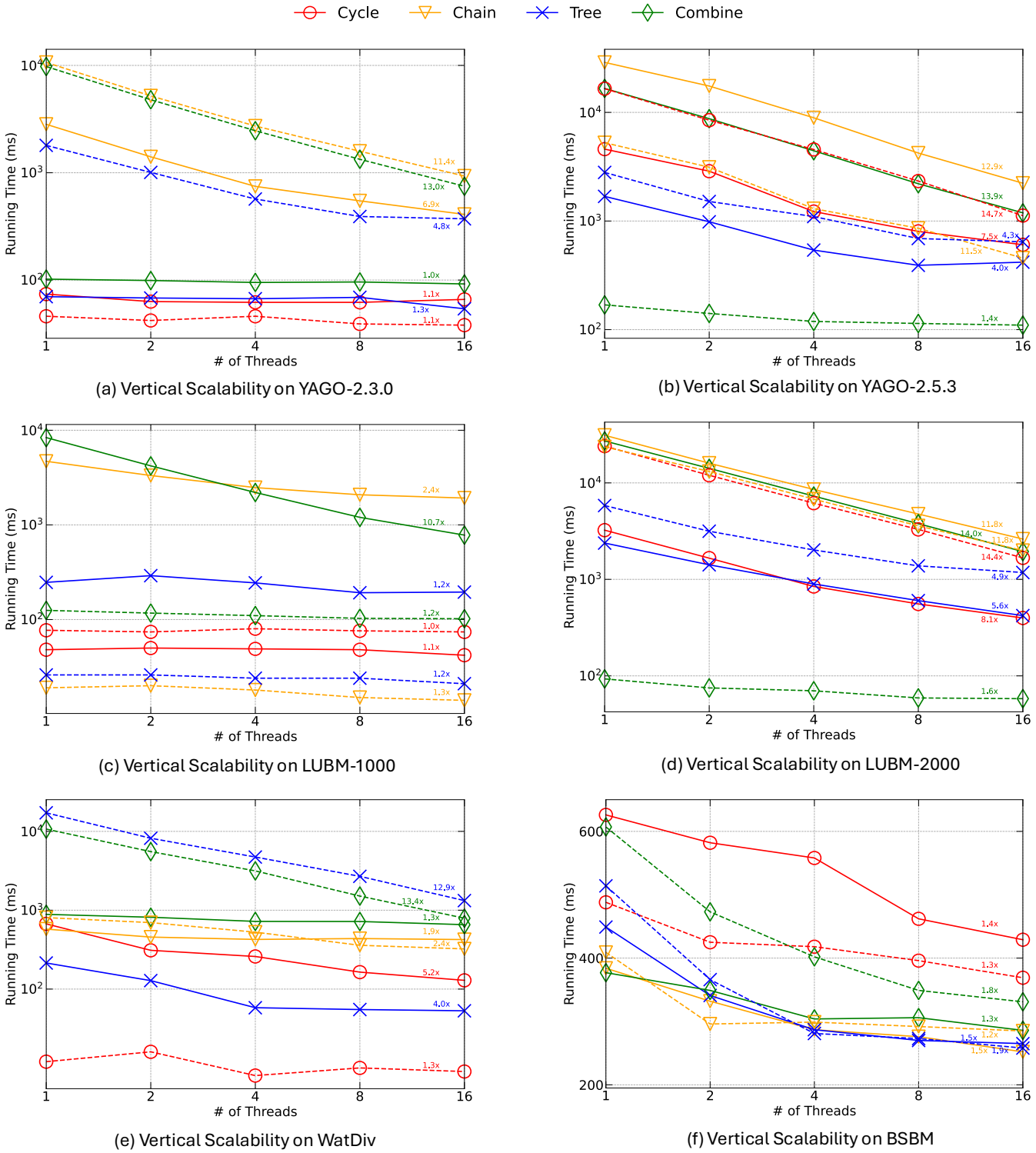
		Cycle		Chain		Tree		Combine		Constant		Var Predicate	
		Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2
YAGO-2.3.0	Ours	25	24	24	28	25	24	28	24	20	25	23	30
	gStore	88	72	35	110	89	38	36	185	82	41	59	72
	Mem Intersect	44	33	83	134	124	84	125	191	99	100	102	139
	Disk intersect	51	37	94	174	149	89	155	228	121	122	152	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

u [70], (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 9 reports the query startup time on all datasets. Comparing Table 2 with Table 9, 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.

D 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 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

Figure 14: Scalability (Time Unit: ms)