CONTENTS

ABSTRACT **1**
1. INTRODUCTION **1**
2. RELATED WORK **2**
3. THE ANSWER-GRAPH APPROACH **2**
4. THE EVALUATION MODEL **3**
5. THE PLANNERS **3**
5.2. THE EMBEDDING PLANNER **5**
6. EXPERIMENTS **5**
7. CONCLUSIONS **6**
REFERENCES **6**

Answer Graph: Factorization Matters in Large Graphs

Zahid Abul-Basher
University of Toronto
Toronto, Canada
zahid@mie.utoronto.ca

Nikolay Yakovets
Eindhoven University of Technology
Eindhoven, The Netherlands
hush@tue.nl

Parke Godfrey
York University
Toronto, Canada
godfrey@yorku.ca

Stanley Clark
Eindhoven University of Technology
Eindhoven, The Netherlands
s.clark@tue.nl

Mark Chignell
University of Toronto
Toronto, Canada
chignell@mie.utoronto.ca

ABSTRACT

Our *answer-graph method* to evaluate SPARQL *conjunctive queries* (CQs) finds a *factorized* answer set first, an *answer graph*, and then finds the embedding tuples from this. This approach can reduce greatly the cost to evaluate CQs. This affords us a second advantage: we can construct a cost-based planner. We present the answer-graph approach, and overview our prototype system, Wireframe. We then offer *proof of concept* via a micro-benchmark over the YAGO2s dataset with two prevalent *shapes* of queries, snowflake and diamond. We compare Wireframe's performance over these against PostgreSQL, Virtuoso, MonetDB, and Neo4J to illustrate the performance advantages of our answer-graph approach.

1 INTRODUCTION

Science is, of course, driven by observation. It is now becoming also ever more driven by data. Some of the datasets involved are unimaginably large. The data is often wildly heterogeneous, and rarely well structured as in business applications. This demands new skills, methods, and approaches of scientists, and challenges computer scientists with devising new data models, query languages, systems, and tools that better support this.

Graph-like data has become prevalent among scientific data stores and elsewhere. The data-science research community has begun to focus on how best to support the management of graph data and its analysis. One *data model* for graph databases is the *Resource Description Framework* (RDF) [18], paired with the *query language* SPARQL [8]. These have evolved as W3C standards, initially for addressing the Semantic Web. An RDF store conceptually consists of a set of *triples* to represent a directed, edge-labeled multi-graph. The triple $\langle s, p, o \rangle$ represents the directed, labeled edge from *subject* node "s" to *target* node "o" with *label* (*predicate*) "p". In RDF, nodes have unique identity. The semantics, however, is carried by the labels and how the nodes are connected. The UniProt [16] SPARQL Endpoint (dataset) [17], for example, consists of 63,376,853,475 RDF triples as of this writing. UniProt (Universal Protein resource) is a freely accessible, popular repository of protein data.

The SPARQL query language provides a formal way to query over such graph databases. Types of SPARQL queries can be thought about as small graphs themselves, so-called *query graphs*. In a SPARQL *conjunctive query* (CQ), the "nodes" are the query's *binding variables* and the "edges" between these are the labels to be matched. An "answer" to a CQ over a *data graph* **G** (that is, the graph database), denoted as $[\![\mathrm{CQ}]\!]_{\mathbf{G}}$, is a homomorphic *embedding* of the query graph into the data graph that matches the query's labels to the data graph's labeled edges. An answer is then a tuple of node ID's as a binding of the query's node variables. As such, each answer can be considered as a sub-graph matched in the data graph.

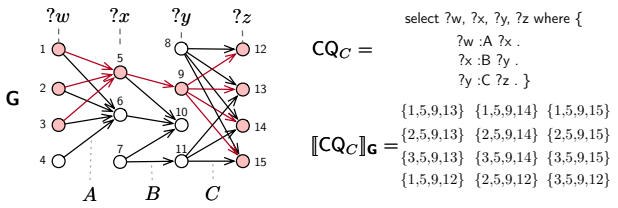

Figure 1: Example of an answer graph (shaded red).

A CQ can be quite expensive to evaluate and may require extreme resources, given both the potentially immense size of the data graph and the relative complexity of the CQ's query graph. The challenge is to reduce the expense and needed machinery. We present a novel approach to *query optimization* and *evaluation* for CQs that we call Wireframe. A set of embeddings is the CQ's answer; this in itself is *not* a graph. In Wireframe's approach, as an intermediate step, we instead find the *answer graph*, the subset of edges from the data graph that suffices to compose the CQ's embeddings. This affords us powerful advantages: the answer graph is essentially the ideal *factorization* of the embeddings;[1] and we can find a best plan by estimated cost to evaluate this answer graph by *cost-based plan enumeration* via dynamic programming.

Wireframe's runtime evaluation employs two reduction mechanisms over the accumulating answer graph—*node burnback* and *edge burnback*—to guarantee a minimal factorized edge set. Given this evaluation paradigm, it is possible to devise a cost-based planner. Wireframe's optimization and evaluation is implementation agnostic; it can be easily implemented on any RDF-system architecture.

We presented the vision of Wireframe's approach in [9]. We have since developed and implemented the approach. Herein, we demonstrate Wireframe's key advantages via a prototype implementation, and compare its performance over a micro-benchmark against competing approaches.

[1] Wireframe can guarantee the minimum answer graph for *acyclic* CQs, and the minimum answer graph modulo the choice of triangulation of the CQ for *cyclic* CQs.

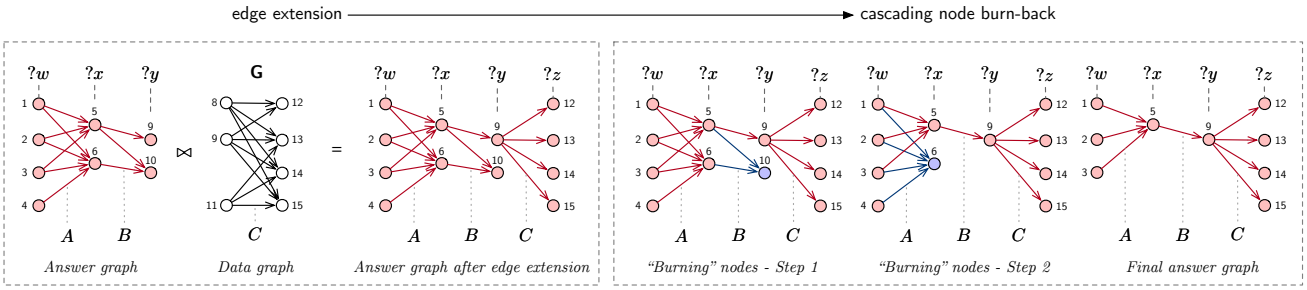

Figure 2: An example of the evaluation model for the answer graph generation.

2 RELATED WORK

RDF Systems. Over the past decade, a number of RDF systems have been developed. We can categorize them into: triple stores; property tables; column-based stores; and graph-based stores.

In triple stores, RDF data is stored in a long, but slim table with three columns where each row is a triple, $\langle s, p, o\rangle$, of RDF data [12]. For conjunctive queries (CQs), this single long, slim table approach requires self-joins over the table many times, which can lead to bad query performance. Large table scans and index look-ups can also lead to bad selectivity estimation, and, therefore, poor query optimization [1]. To overcome this, an index over the triple store for each of the six permutations over S, P, and O is often maintained [12, 24]. RDF-3x uses this approach by building a clustered B++ trees for each permutation of $\langle s, p, o\rangle$: $\langle s, p, o\rangle$, $\langle s, o, p\rangle$, $\langle p, s, o\rangle$, $\langle p, o, s\rangle$, $\langle o, s, p\rangle$, and $\langle o, p, s\rangle$ [12, 13]. It also maintains additional nine aggregate indexes to include the six binary and three unary projections of $\langle s, p, o\rangle$, which is useful for selectivity estimation. The aggregate indexes eliminate the need for expensive self-joins, therefore, improving query performance. TripleBit [26] constructs a compact triple matrix to minimize the use of indexes during query evaluation.

In contrast to triple stores, property tables use a flat but fat table where each row represents a subject value with columns for distinct property values [20]. In large, sparse RDF data, many cells of this single flat-fat table can be *null*, due to the absence of object values for the given subjects and predicates. To overcome this, Jena [20] clusters properties into different groups, and creates multiple property tables accordingly. BitMat [3] proposed a 3-D bit cube to represent subjects, predicates, and objects.

In column-based stores, RDF data is stored using multiple two-column tables [1]. There is a table for each unique property, with one column for the subject, and the other for the object. The tables can be stored using either physical row-store or column-store. This approach provides superior performance whenever there are value-based restrictions on properties. It can scale poorly, however, when the size of tables varies [15].

Graph-based stores are designed to handle graph manipulation over RDF data generally outside of the scope of SPARQL queries [6, 11]. These systems focus on specialized graph operations over RDF data [19]. For better performance, gStore [27] constructs VS-tree and VS*-tree index to evaluate both exact and wildcard SPARQL queries using subgraph matches. In [2], a compressed k^2-triples technique is used to run SPARQL queries in memory.

Factorization and Join Algorithms. Factorized databases are compact representations of relational tables [4]. They not only reduce the memory footprint while evaluating queries, but also reduce the query processing time by avoiding redundancy. This idea is even more effective for graph databases and queries, as we can show that our answer graph is an ideal factorization.

The *semijoin* operator can improve performance by ensuring everything in the *outer* (left) table *joins* with the *inner* (right) table. Wireframe's burnback mechanisms implement a form of semijoin, in a way. We likely could re-engineer our Wireframe burnback mechanisms via semijoin, were we to implement atop a platform providing a highly efficient semijoin.[2]

Worst-case optimal join algorithms use query decompositions for joins, accounting for the structural properties of the query along with the input relation statistics. This is in contrast to a traditional database where joins are evaluated "one join at a time" without taking the structure of the query into account [14].

Work that is related to ours in its mechanics is that of [25]. In [25], they reduce the transmission cost in distributed environments by generating a plan—*i.e.*, a sequence of semijoins—to evaluate acyclic conjunctive queries over datasets partitioned across different servers. Of course, our objectives in Wireframe and that of [25] are different, necessitating different methodologies. That said, our approach has advantages. Their algorithm for an acyclic CQ requires traversing the query tree *twice*. Wireframe does not need to. Their work does not apply to cyclic queries, whereas Wireframe does.

3 THE ANSWER-GRAPH APPROACH

In [4], the authors introduce the concept of *factorization* as a query-optimization technique for relational databases. Their technique is designed, and works exceptionally well, for schema and queries for which cross products of projections of the answer tuples all show up as answer tuples. This happens, for instance, in schema not in *fourth normal form*. Evaluating for these projected tuples first and then cross-producting them later can be a much more efficient strategy. Deciding how best to *factorize*—how to project into sub-tuples—is difficult, however.

For CQs, this last part is trivial: the factorization of the embedding tuples is fully down to component node pairs, corresponding to the labeled edges. This is our *answer graph*.[3] Factorization is *sometimes* a significant win for evaluating relational queries; it is *virtually always* a win for evaluating graph CQs.

An answer graph, AG, for a CQ is a subset of the data graph **G** that suffices to compute the embeddings for the CQ. We call the *minimum* such subset the *ideal answer graph*, iAG. The iAG is often quite small, significantly smaller than the set of embeddings, and extremely much smaller than **G**. Thus, we evaluate a CQ's embeddings in two steps: first, we find its iAG; then we compose

[2] This is future work of ours.

[3] This is demonstrably true when the CQ is *tree* shaped. This is arguable when the CQ has cycles. In the latter case, the factorization can be characterized as projections to tuples of node pairs and node triples (*triangles*).

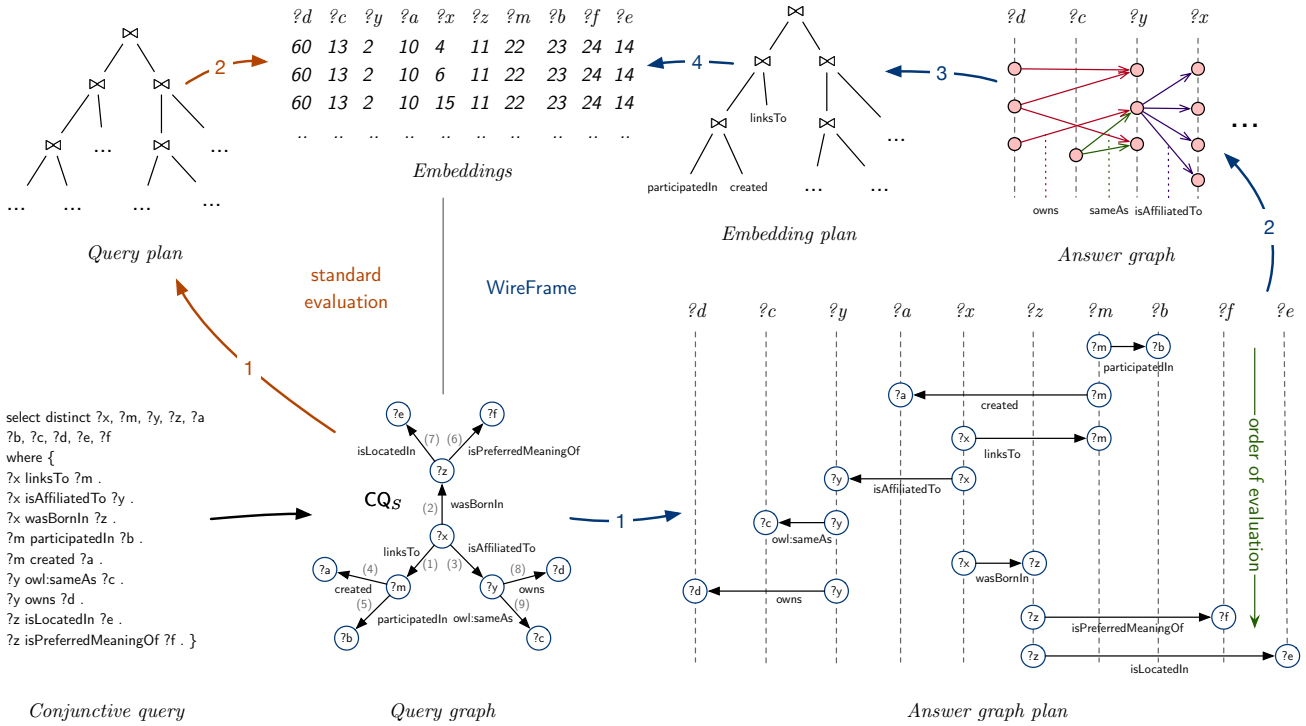

Figure 3: WIREFRAME: a two-phase cost-based optimizer for conjunctive queries.

the embeddings, which we call *defactorization*, from the iAG, rather than from **G**. This two-step approach can be significantly more efficient.

Consider the data graph **G** and the *chain* query CQ_C in Fig. 1. The query finds all tuples of nodes $\langle w, x, y, z\rangle$ from **G** such that $\langle w, x\rangle$ is connected by an edge labeled A, $\langle x, y\rangle$ by B, and $\langle y, z\rangle$ by C. Due to multiplicity from A-edges *fanning in* to, and C-edges *fanning out* of, B values, the embedding set is *twelve* tuples. Our answer graph consists of just *eight* labeled node pairs (in red). Such differences are greatly magnified when on a larger scale.

Our answer-graph approach affords us a second key advantage. We can devise a cost-based query optimizer based on dynamic programming to construct a *query plan*. A plan for us is simply a specified order of the CQ's query edges with which to evaluate to matching answer-graph edges. Our evaluation strategy for such plans is explained next, and our WIREFRAME optimizer for choosing plans is presented in Section 5.

4 THE EVALUATION MODEL

Our evaluation model for CQs then becomes two phase: *answer-graph generation* and *embedding generation*.

Answer-graph generation. For each query edge of the plan, in turn, our answer graph (AG) is populated with the matching labeled edges from **G** that meet the join constraints with the current state of the AG. Call this an *edge-extension* step. Then nodes in the AG that failed to extend are *removed*, and this "node burnback" cascades.

Consider the CQ with query edges $\langle ?w, A, ?x\rangle$, $\langle ?x, B, ?y\rangle$, and $\langle ?y, C, ?z\rangle$ in Fig. 2. Assume that the state of the AG after evaluating for query edges $\langle ?w, A, ?x\rangle$ and $\langle ?x, B, ?y\rangle$ is as shown in the figure, and that the next query edge to be evaluated is $\langle ?y, C, ?z\rangle$. This next edge is connected to the previously evaluated edges by the node variable $?y$. When retrieving data edges from **G** with label C (the query edge's label), one needs to ensure that the sources of retrieved data edges match to one of nodes bound to $?y$ in AG. After populating AG thusly, many nodes of $?y$ in AG may be "unattached" to any of the new edges; these nodes are marked to be removed during the node burnback procedure. In Fig. 2, the new answer graph has an unattached node, 10. During node burnback, this node is removed along with all of its edges, $\langle 5, 10\rangle$ and $\langle 6, 10\rangle$. Removing these edges can result in more unattached nodes, such as node 6, which no longer has any edge with the label A (contrasted with node 5). Thus, in the next iteration, node 6 is removed with all of its edges, $\langle 1, 6\rangle$, $\langle 2, 6\rangle$, $\langle 3, 6\rangle$, and $\langle 4, 6\rangle$. The node burnback procedure then terminates, as no further unattached nodes result.

Embedding generation. The embedding tuples are then generated over the answer graph by joining the answer edges appropriately. Given the *ideal* answer graph iAG and an acyclic CQ, the order in which we join is immaterial. No k-ary tuple is ever eliminated during a join with a next query edge from the iAG. This step is often quite fast, given the iAG is small. Evaluating this directly from the data graph **G**, on the other hand—which is what other evaluation methods for CQs do—can be exceedingly expensive. Fig. 3 illustrates, comparing a standard evaluation with our two-phase answer-graph approach.

5 THE PLANNERS

5.1 The Answer-Graph Planner

Plan Cost. The *edge walk* is our unit for estimating a plan's cost: the retrieval of a matching edge from **G**. To estimate the number of edge walks, *node* and *edge* cardinality estimations are made for each successive edge extension. Note that the cost of node burnback is amortised: every edge added that does not survive

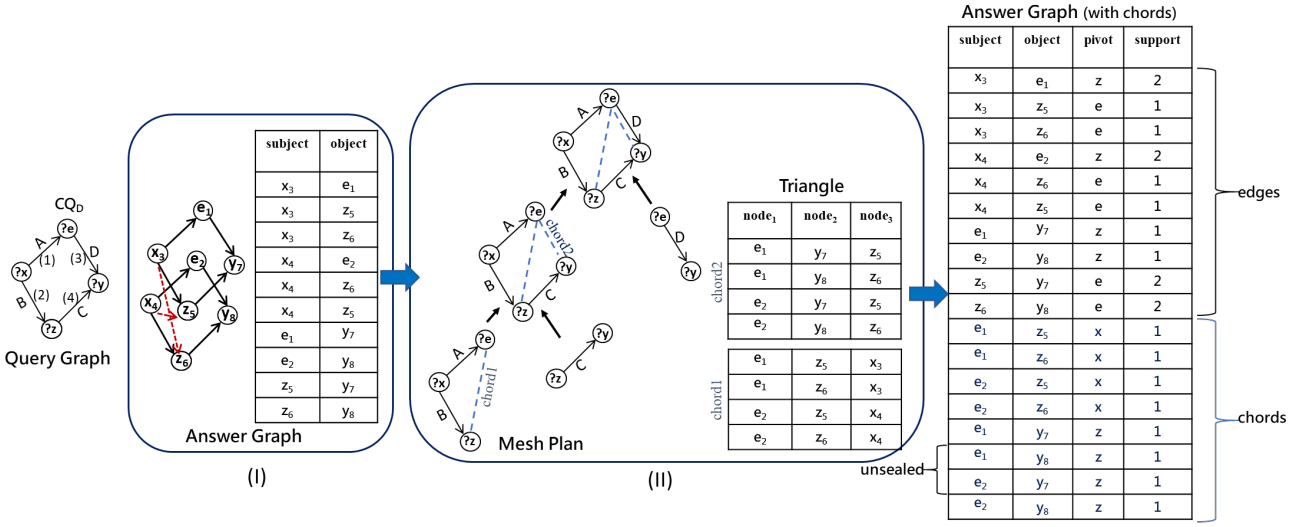

Figure 4: Triangulating a cyclic CQ using a mesh plan

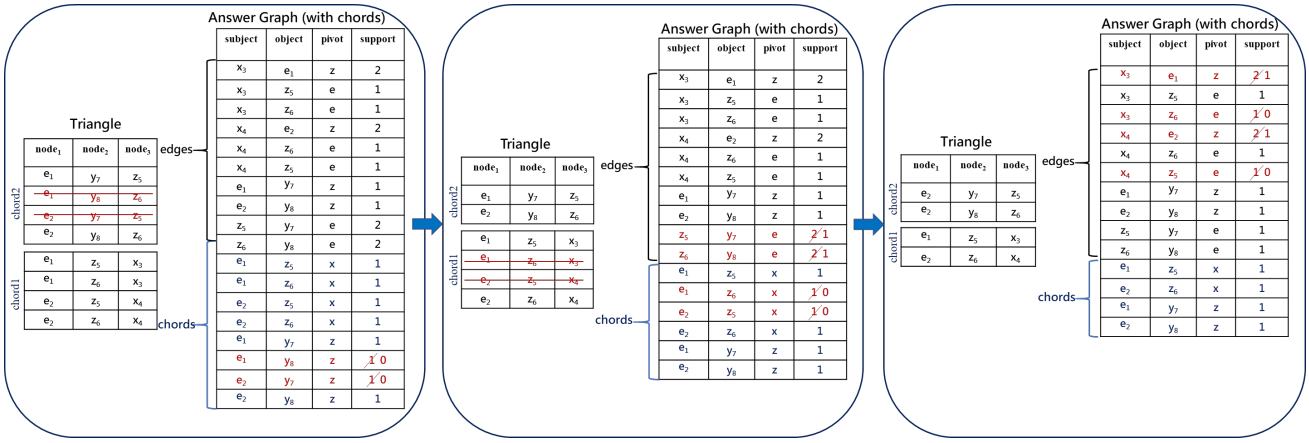

Figure 5: A running example of edge burnback procedure

to the iAG is, at some point, removed. WIREFRAME employs cardinality estimators drawn from a catalog consisting of 1-gram and 2-gram edge-label statistics computed offline [7, 10, 22, 23].

The Edgifier. A plan is a sequence of the CQ's query edges to be materialized. We employ a bottom-up, dynamic-programming algorithm to construct the edge order based on cost estimation (which relies upon the cardinality estimations).

When the query graph of a CQ has cycles—a *cyclic* query—there is an additional part to planning. Node burn-back suffices to generate the ideal answer graph for acyclic queries, but not for cyclic. The example in Fig. 4(I) illustrates why. *Spurious* edges (in red)—e.g., $\langle x_3, z_6 \rangle$ and $\langle x_4, z_5 \rangle$—can remain that do not participate in any final embeddings—i.e., $\langle x_3, z_5, y_7, e_1 \rangle$ and $\langle x_4, z_6, y_8, e_2 \rangle$. Nevertheless, one can still use this non-ideal answer graph to generate the embeddings using defactorization.

Even so, it is possible to reduce significantly further the answer graph. To cull spurious edges requires an *edge burnback* procedure in addition to node burnback. This requires the CQ's cycles have been *triangulated*; node *triples* are materialized in addition to the node *doubles* (the AG edges) during evaluation. Triangulation is the choice of which additional "query edges", which we call *chords*, to add.

The Triangulator. For cyclic CQs, in addition to the query-edge enumeration, cycles in the query graph of length greater than three are *triangulated* by adding *chord* edges. During evaluation, a chord is maintained as the intersection of the materialized joins of the opposite two edges for each triangle in which it participates. There are many different ways one can triangulate a CQ; the materialization cost depends on the order and choice of chord bisection of cycles (down to triangles). We employ a bottom-up dynamic programming algorithm to generate a bushy plan—we call this a *mesh plan*—that dictates such an order and choices.

The mesh plan, when executed with node burnback but not edge burnback, guarantees that the node sets, but not necessarily the edge sets, will always be minimal. A correct answer graph, AG, will be found, but it is no longer guaranteed to be ideal. as spurious edges may remain in the AG, as demonstrated in Fig. 4(I). The embeddings can, of course, be found from this non-ideal AG.

Edge Burnback. With the addition of *edge burnback* mechanism at runtime, we can guarantee again that we find the ideal AG (iAG) when answering cyclic CQs, *modulo* the choice of chords that were added. This works by checking the chords' materializations to chase what needs to be removed on cascade. This ensures that spurious edges are removed.

Fig. 4(II) shows a mesh plan which establishes as sequence of chords to triangulate the given query graph CQ_D. At the bottom of the plan, the first chord $\langle ?e, ?z\rangle$ is added, which creates the triangle $\bigtriangleup_{?e,?x,?z}$ by joining two query edges $\langle ?x, ?z\rangle$ and $\langle ?x, ?e\rangle$ on their common node $?x$. At the next step up, a second chord $\langle ?e, ?y\rangle$ is added to create a triangle $\bigtriangleup_{?e,?z,?y}$ by joining the previous chord $\langle ?e, ?z\rangle$ and the query edge $\langle ?z, ?y\rangle$ on their common node $?z$. At the root of the plan, a "seal" process occurs between the chord $\langle ?e, ?y\rangle$ of the left subplan and the query edge $\langle ?e, ?y\rangle$ of the right subplan, where it constructs the full cycle of the query graph. Each time we triangulate, we record the node triples (as triangles) drawn accordingly from **G** in the Triangle table, and the data chords in the Answer Graph table. We also record in the support column of the Answer Graph table the number of different triangles that each data edge (or data chord) participate, grouped by its opposite node label (which we call the *pivot*). For example, in the Answer Graph table, the edge $\langle x_3, e_1\rangle$ has a support of 2 with the pivot of z; that is, it is part of two triangles where the opposite node is of the label z. We can find these from the Triangle table: $\bigtriangleup_{e_1 z_5 x_3}$ and $\bigtriangleup_{e_1 z_6 x_3}$.

The seal process marks all uncommon data edges and data chords between the right and left subplans—we call these *unsealed* edges—by updating their supports in the Answer Graph table to 0. Unsealed edges represent the arcs that are not part of any cycle. Data edges that reside only on such arcs cannot be part of the ideal AG. The task of edge burnback is to remove these edges. It begins by removing all unsealed edges, along with triangles in which they participate. Removing a triangle involves removing all its data edges, which might also belong to other triangles in the query graph, thereby removing those triangles too. This process cascades until no unsupported triangle is left to be removed. Whenever we remove a triangle, we decrease the support of all of its constituent edges by one in the Answer Graph table. When the support of any data edge or chord in the Answer Graph becomes 0, then it is safe to remove it along with the triangles in which it participates. The process cascades until there are no edges with zero-support remaining in the Answer Graph table.

Fig. 5 demonstrates the process of edge burnback. After the sealing process, the unsealed set of edges are $\{\langle e_1, y_8\rangle, \langle e_2, y_7\rangle\}$, as shown in the Answer Graph table of Fig. 4(II). We next update the support of these two edges to 0 in the Answer Graph table. We then call the edge burnback procedure. This first removes the zero-support edges from the Answer Graph table. Next, this deletes triangles $\bigtriangleup_{e_1 y_8 z_6}$ and $\bigtriangleup_{e_2 y_7 z_5}$ which those edges participated from the Triangle table. The support for each of the edges in the Answer Graph table that was part of a recently deleted triangle is decremented by 1. For example, for the removed triangle $\bigtriangleup_{e_1 y_8 z_6}$, the support is decremented for edges $\langle z_6, y_8\rangle$ and $\langle e_1, z_6\rangle$. For the removed triangle $\bigtriangleup_{e_2 y_7 z_5}$, the support is decremented for edges $\langle e_2, z_5\rangle$ and $\langle z_5, y_7\rangle$. In the next iteration, the zero-support edges are removed from the Answer Graph table, which leads, in turn, to triangles $\bigtriangleup_{e_1 z_6 x_3}$ and $\bigtriangleup_{e_2 z_5 x_4}$ being deleted from the Triangle table. The support is then decremented for the edges that participated in those deleted triangles. And so forth. This process halts once there is no edge left with zero-support in the Answer Graph table. The resulting answer graph is then the ideal AG (iAG).

The overhead of edge burnback must be balanced off against the benefit of obtaining the iAG versus a larger, non-ideal AG. This is work in progress. In our experiments, our evaluation over cyclic CQs is without edge burnback.

5.2 The Embedding Planner

Plan Cost. When generating the embeddings for an acyclic CQ from its iAG, the order in which we join (connected) answer edges is immaterial. As the k-ary tuples are extended, no intermediate results are ever lost. Thus, for this, no planning is required.

The Defactorizer. On the other hand, when the CQ is cyclic, or when the AG provided is non-ideal, intermediate results can be lost. The join order then matters. We call this process *defactorization*. Alternative plans for embedding materialization are synonymous with choosing this join order. It is possible to do this again via a cost-based approach via a bottom-up, dynamic programming algorithm, using our catalog statistics.

6 EXPERIMENTS

Prototype. We have implemented a prototype, Wireframe, that runs on top of PostgreSQL, a popular relation database system. Wireframe implements the two phases described in Section 5, each with a separate planner and evaluator. The planner for the first phase outputs an optimal left-deep tree plan that indicates the execution order of the query edges. The evaluator then takes the tree plan to evaluate the query edges in sequence. For the second phase, we presently use a greedy approach to generate a tree plan based on the available statistics from the AG phase. The node burnback procedure is implemented via procedural SQL.

Environment. To evaluate Wireframe's performance, we use the YAGO2s dataset [21] containing 242M triples with 104 distinct predicates. With a select set of 10 acyclic and 10 cyclic CQs, we compare query execution times against PostgreSQL v11.0 (PG), Virtuoso v6.01 (VT), MonetDB v11.31 (MD), and Neo4J v.3.5 (NJ). All experiments were conducted on a server running Ubuntu 18.04 LTS with two Intel Xeon X5670 processors and 192GB of RAM.

Micro-benchmark. For the queries, we implemented a query miner that generates valid, non-empty queries over a dataset using query templates (with placeholders for edge labels). For our experiments, we use two templates, CQ_S and CQ_D, as shown in Figures 3 and 4, respectively. With these two templates, we mined 218,014 snowflake-shaped queries and 18,743 diamond-shaped queries. For our preliminary experimental study, we chose top ten queries in the size of final embeddings for each shape.

While in [5], it is argued that there are no use cases for cyclic queries, many, including us, have argued there certainly are. In [14], they discuss how triangle queries, the simplest of cyclic, have become increasingly popular for social networks, biological motifs, and graph databases. And that cyclic queries have not been used much yet in practice has been due in large part to that they have been too expensive to evaluate.

Comparators. For PostgreSQL and MonetDB, the dataset was imported as a triple store, with indexes on the string dictionary, and six composite indexes over the permutations of *subject*, *predicate*, and *object*. We set the size of the memory pool to eight GB for all of the systems, except for MonetDB (which sets its own resource allocations based on the server). We repeat execution of each query five times, taking the average of the last four runs (*i.e.,* warm cache), as reported in Table 1. The execution time is the time spent to retrieve all the result tuples for a query.

Results. One can observe from Table 1 that the size of the answer graph is exceedingly smaller than the number of embeddings. For instance, for the second snowflake-shaped query, the AG is 2,867 times smaller than the number of embeddings. It is no surprise, therefore, that Wireframe (WF) achieves good performance; it

CQ_S	**Snowflake-shaped Queries** (1/2/3/4/5/6/7/8/9)	PG	WF	VT	MD	NJ	|iAG|	|Embeddings|
1	diedIn/influences/actedIn/owns/wasCreatedOnDate/actedIn/created/hasDuration/wasCreatedOnDate	66	**4**	*	*	*	1660	2931986
2	hasChild/influences/actedIn/actedIn/wasBornIn/created/actedIn/hasDuration/wasCreatedOnDate	63	**3**	246	*	*	993	2847184
3	isCitizenOf/influences/actedIn/exports/wasCreatedOnDate/actedIn/created/hasDuration/wasCreatedOnDate	37	**7**	287	*	*	1140	2670339
4	isMarriedTo/influences/actedIn/actedIn/wasBornOnDate/created/actedIn/hasDuration/wasCreatedOnDate	59	**3**	286	*	*	3317	2569017
5	isMarriedTo/influences/actedIn/wasBornOnDate/isMarriedTo/actedIn/created/wasCreatedOnDate/hasDuration	57	**17**	268	*	*	3580	2127992
6	isMarriedTo/influences/actedIn/hasGender/isMarriedTo/actedIn/created/hasDuration/wasCreatedOnDate	57	**12**	268	*	*	3580	2123951
7	diedIn/isMarriedTo/actedIn/owns/wasCreatedOnDate/actedIn/created/hasDuration/wasCreatedOnDate	30	**15**	266	*	*	10761	2111948
8	isMarriedTo/influences/actedIn/hasFamilyName/isMarriedTo/created/actedIn/hasDuration/wasCreatedOnDate	32	**14**	261	*	*	3580	2102297
9	isMarriedTo/hasChild/actedIn/wroteMusicFor/created/created/actedIn/hasDuration/wasCreatedOnDate	35	**9**	256	*	*	7330	1786626
10	isMarriedTo/influences/actedIn/actedIn/created/created/directed/hasDuration/wasCreatedOnDate	39	**4**	237	*	*	3317	1533188
CQ_D	**Diamond-shaped Queries** (1/2/3/4)	PG	WF	VT	MD	NJ	|AG|	|Embeddings|
11	isLocatedIn/linksTo/isCitizenOf/livesIn	*	**39**	*	*	*	813311	59695937
12	livesIn/isCitizenOf/isLocatedIn/linksTo	*	**81**	*	*	*	833355	58785214
13	isCitizenOf/wasBornIn/linksTo/diedIn	*	**12**	*	*	297	132961	3141996
14	isCitizenOf/diedIn/linksTo/wasBornIn	*	**21**	*	*	296	251054	3124213
15	wasBornIn/isAffiliatedTo/linksTo/playsFor	*	**37**	*	*	*	470196	2310680
16	wasBornIn/playsFor/linksTo/isAffiliatedTo	*	**39**	*	*	*	471520	2299729
17	isConnectedTo/linksTo/extractionSource/byTransport	*	**33**	67	*	140	112040	1312372
18	created/rdfs:label/linksTo/isPreferredMeaningOf	*	264	**65**	203	130	772994	169380
19	linksTo/isPreferredMeaningOf/created/skos:prefLabel	*	114	**22**	111	135	766785	169324
20	diedIn/linksTo/wasBornIn/graduatedFrom	*	**12**	92	*	195	68720	106214

Table 1: Query execution time (sec) in different systems (* denotes terminating the query after 300 seconds).

avoids the redundant edge-walks that arise from many-many joins. While the second snowflake-shaped query took 63 seconds on PostgreSQL, it only took three seconds on Wireframe. The AG approach requires a much smaller memory footprint, which can be beneficial for traditional database systems that heavily use secondary storage. The approach also competes well against main-memory intense systems such as Neo4J and Virtuoso. For the cyclic, diamond-shaped queries, employing only node burnback does not guarantee the ideal answer graph, as discussed above. We have found that the resulting AGs can be significantly larger than the ideal, sometimes close to the number of embeddings. For this reason, Wireframe was slower for some of the cyclic queries, notably 18 and 19. Even so, its performance over cyclic queries is quite good. With further plan- and run-time optimization with edge burnback, we believe that the performance will be stellar. One can view our approach as an additional optimization technique on top of traditional databases to handle SPARQL CQs or to boost the performance of existing RDF systems.

7 CONCLUSIONS

We have clear objectives for our next steps. *First*, one has a richer plan space when considering bushy plans for both our first and second phases. The challenge is to devise a suitable cost model for searching the bushy-plan space via dynamic programming. *Second*, when the size of an answer graph is distant from the ideal, generating the embeddings can be costly. Triangulation promises to reduce this significantly. This requires investigating the trade-offs between the added cost for maintaining the triangle materializations and the reduced cost from generating the embeddings from the significantly smaller ideal AG. *Lastly*, we are to explore further optimizations within this space. Large graphs are meant to be queried.

REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. VLDB '07, pages 411–422. VLDB Endowment, 2007.

[2] S. Álvarez-García, N. R. Brisaboa, J. D. Fernández, and M. A. Martínez-Prieto. Compressed k2-triples for full-in-memory rdf engines. *arXiv preprint arXiv:1105.4004*, 2011.

[3] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: A scalable lightweight join query processor for rdf data. WWW '10, pages 41–50, New York, NY, USA, 2010. ACM.

[4] N. Bakibayev, D. Olteanu, and J. Závodnỳ. FDB: A query engine for factorised relational databases. *VLDB*, 5(11):1232–1243, 2012.

[5] A. Bonifati, W. Martens, and T. Timm. An analytical study of large sparql query logs. *The VLDB Journal*, 29(2):655–679, 2020.

[6] V. Bonstrom, A. Hinze, and H. Schweppe. Storing rdf as a graph. In *Proceedings of the IEEE/LEOS 3rd International Conference on Numerical Simulation of Semiconductor Optoelectronic Devices*, pages 27–36, 2003.

[7] S. Christodoulakis. *On the estimation and use of selectivities in database performance evaluation.* CS Dept., U. of Waterloo, 1989.

[8] A. S. Eric Prud'hommeaux. SPARQL query language for RDF. W3C recommendation, 15 january, 2008.

[9] P. Godfrey, N. Yakovets, Z. Abul-Basher, and M. H. Chignell. Wireframe: Two-phase, cost-based optimization for conjunctive regular path queries. In *AMW*, 2017.

[10] M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys (CSUR)*, 20(3):191–221, 1988.

[11] A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura. A path-based relational rdf database. In *Proceedings of the 16th Australasian Database Conference - Volume 39*, ADC '05, pages 95–103, Darlinghurst, Australia, Australia, 2005.

[12] T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.

[13] T. Neumann and G. Weikum. x-rdf-3x: Fast querying, high update rates, and consistency for rdf databases. *Proc. VLDB Endow.*, 3(1-2):256–263, Sept. 2010.

[14] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record*, 42(4):5–16, 2014.

[15] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for rdf data management: Not all swans are white. *Proc. VLDB Endow.*, 1(2):1553–1563, Aug. 2008.

[16] The UniProt consortium (author notes), UniProt: a worldwide hub of protein knowledge. *Nucleic Acids Research*, 47(D1):D506–D515, January 2019.

[17] UniProt SPARQL endpoint. https://sparql.uniprot.org/,2020.

[18] W3C: Resource description framework (rdf). http://www.w3.org/TR/rdf-concepts/,2004.

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

[20] K. Wilkinson and K. Wilkinson. Jena property table implementation, 2006.

[21] YAGO2s: A high-quality knowledge base. http://yago-knowledge.org/resource/. Max Planck Institut Informatik.

[22] N. Yakovets. *Optimization of Regular Path Queries in Graph Databases.* PhD thesis, York University, 2016.

[23] N. Yakovets, P. Godfrey, and J. Gryz. Query planning for evaluating SPARQL property paths. In *SIGMOD*, pages 1–15. ACM, June 2016.

[24] Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma, and Y. Pan. Efficiently querying rdf data in triple stores. In *Proceedings of the 17th international conference on World Wide Web*, pages 1053–1054. ACM, 2008.

[25] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, volume 81, pages 82–94, 1981.

[26] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. Triplebit: a fast and compact system for large scale rdf data. *Proceedings of the VLDB Endowment*, 6(7):517–528, 2013.

[27] 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, May 2011.