Spec-QP: Speculative Query Planning for Joins over Knowledge Graphs

Madhulika Mohanty
Indian Institute of Technology, Delhi
Hauz Khas, New Delhi, India
madhulikam@cse.iitd.ac.in

Maya Ramanath
Indian Institute of Technology, Delhi
Hauz Khas, New Delhi, India
ramanath@cse.iitd.ac.in

Mohamed Yahya*
Bloomberg
London, United Kingdom
yahya.mohamed@gmail.com

Gerhard Weikum
Max Planck Institute for Informatics
Saarland Informatics Campus, Germany
weikum@mpi-inf.mpg.de

ABSTRACT

Knowledge Graphs (KGs) have become ubiquitous in organisations. They provide a unified and structured model to store the data as well as facilitate effective search to fulfill many complex information needs. One of the ways to query these KGs is to use SPARQL queries over a database engine. Since SPARQL follows exact match semantics, the queries may return too few or no results. Recent works have proposed *query relaxation* where the query engine judiciously replaces a query predicate with similar predicates using weighted relaxation rules mined from the KG. However, the space of possible relaxations is potentially too large to fully explore and users are typically interested in only top-k results, so such query engines use top-k algorithms for query processing. Nevertheless, they still process all the relaxations, many of whose answers do not contribute towards top-k answers. We propose Spec-QP, a query planning framework that speculatively determines which relaxations will have their results in the top-k answers. This reduces the computational overheads and gives faster response times with good precision over top-k results. We tested Spec-QP over two database engines, PostgreSQL and Virtuoso, with two datasets – XKG and Twitter – to demonstrate the efficiency of our planning framework at supporting relaxations in query engines.

1 INTRODUCTION

Knowledge Graphs (KGs) such as YAGO [34], DBPedia [2] and Freebase [5] are typically stored as RDF triples of ⟨s p o⟩ where s is the subject, o is the object and p is the predicate. These RDF KGs are queried using the SPARQL query language, that, at its core consists of *triple patterns*. For example, the following SPARQL query asks: "Which singers also write lyrics and play guitar and piano?".

```
SELECT ?s WHERE{
    ?s 'rdf:type' <singer>.
    ?s 'rdf:type' <lyricist>.
    ?s 'rdf:type' <guitarist>.
    ?s 'rdf:type' <pianist>
}
```

where ?s is a variable to be bound in each of the four triple patterns and to be returned as a result.

*Work done while at the Max Planck Institute for Informatics.

Original	Relaxations
`<singer>`	`<vocalist>`,`<jazz_singer>`, `<artist>`
`<lyricist>`	`<writer>`
`<guitarist>`	`<musician>`, `<instrumentalist>`
`<pianist>`	`<percussionist>`

Table 1: Example relaxations

An exhaustive list of such singers in the KG can be computed, but users who issue such queries typically want only the top-k, *ranked* results. Ranking of SPARQL query results has been studied before in [9, 12, 23] and they typically make use of *scores* for each triple in the KG[1]. However, a problem that users sometimes face when they issue such queries is *low recall*. That is, the KG may not have k results to return (in some cases, the KG may have *zero* results if one or more of the triple patterns do not have a match). In these cases, it is desirable to *relax* the query by replacing one or more of the triple patterns, while ensuring that the query still reflects the original information need. For example, a possible relaxation of the query above is to change the triple pattern ⟨?s 'rdf:type' <singer>⟩ to ⟨?s 'rdf:type' <vocalist>⟩. Previous works have dealt with doing these relaxations *automatically* and ranking the corresponding results [10, 18, 31, 42]. In this paper, we address the problem of *efficiently evaluating* these relaxed queries.

Query Processing. Processing queries and their relaxations to return top-k results is computationally expensive. For example, assuming that every triple pattern in the above query has relaxations as shown in Table 1, would lead to a total of 48 unique queries (that is, original query, query with one relaxation, query with two relaxations, etc.). A naive method would compute the results to each query, sort the results by score and return the top-k.

Since the user is looking for only top-k answers, the naive method can be improved upon by using top-k operators. They can compute results from *all* relaxations simultaneously, but in a way that drastically reduces wasteful computations. The following two top-k operators can be employed for achieving this: *Incremental Merge* [35] (to process the relaxations for a given triple pattern) and *Rank Join* [19] (to compute (partial) join results in sorted order). However, this method still results in wasted resources, since not all relaxations contribute a result to the top-k.

[1]The scores could be based on confidence values, popularity, etc.

Approach and contributions. In this paper, we propose a *speculative* approach for pruning the space of possible relaxations for a given query. We make use of precomputed statistics about the distribution of scores of the matches to triple patterns in order to speculate on the requirement of relaxations for each triple pattern in order to get top-k results. This precomputed metadata is an approximation of the score distribution of the answers from the corresponding triple pattern and not the actual scores. This is like computing histograms, or simpler. Each of these statistics can be computed in one pass as part of the statistics collection phase of any database system that does cost-based optimization of queries.

When a user enters a query, we estimate the top answer scores that can be achieved using the possible relaxations. This estimation is done using the score distributions and the join cardinality estimates. We then prune those relaxations which are unlikely to contribute triples to the top-k answers based on the top score estimates. This results in reduced computation and faster response times. Also, the amount of search space traversed is reduced by pruning unnecessary relaxations and this, in turn, leads to reduction in memory requirements. The runtime reduction combined with reduced memory requirements leads to an overall improvement on memory consumed over time. This is especially beneficial for servers where the total resource consumption per query is important, as it is inversely proportional to the achievable throughput. Or equivalently, the cost of running the server for a given load is proportional to the cost per query and this improvement implies that the server can run the service with lesser budget in terms of money. *Note that our work is orthogonal to any query engine as it can be deployed on top of any existing RDF-specific database engine.*

Our main contributions are summarized as follows.

i. A model for the score distribution of individual triple patterns.
ii. A technique to estimate the scores of answers to a query using the above model and using it to predict the presence of answers from each triple pattern's relaxations in the top-k.
iii. Pruning the space of relaxations to achieve significantly faster response times while maintaining high accuracy, thereby aiding *cost-effective* exploration of KGs.
iv. Thorough experimental evaluation of the proposed technique over two database engines – PostgreSQL and Virtuoso – with two real world datasets to demonstrate its efficiency over the baseline.

Organisation. The rest of the paper is organised as follows: section 2 introduces useful definitions and explains the top-k operators based query processing approach. Section 3 outlines Spec-QP, the proposed speculative approach to query planning and its execution. Section 4 discusses the experimental results. Section 5 lists the related work and finally section 6 concludes the paper with future work directions.

2 PRELIMINARIES

This section introduces some preliminary definitions that will be used henceforth.

Definition 2.1. **Knowledge Graphs (KGs)**
Given a set of entities $\mathbf{E}$ and predicates $\mathbf{P}$, a triple t is a tuple $t = \langle \mathsf{s}\ \mathsf{p}\ \mathsf{o} \rangle$ such that, $t \in \mathbf{E} \times \mathbf{P} \times \mathbf{E}$, $\mathsf{s} \in \mathbf{E}$, $\mathsf{p} \in \mathbf{P}$ and $\mathsf{o} \in \mathbf{E}$. Here, s is called the "subject", p is the "predicate" and o is the "object" of the triple t. Each triple is associated with a score, denoted by $S(t)$. These scores represent confidence values or popularity of the triples as previously studied in [9, 12, 23]. A set of such tuples can be represented as a graph, which we call a Knowledge Graph, $\mathbf{KG} \subseteq \mathbf{E} \times \mathbf{P} \times \mathbf{E}$.

Definition 2.2. **Triple pattern query**
A triple pattern is of the form $q = \langle \mathsf{SPO} \rangle$, where S, P and O could either be entities or predicates from the KG or variables. Variables are always prefixed with a question mark. A triple pattern matches any triple in the KG having the same values in the designated field. The variables are then bound to the corresponding values in the triple. A triple pattern query is a set of triple patterns, $\mathbf{Q} = \{q_1, q_2, ... q_n\}$.

Definition 2.3. **Answer for a Triple pattern query**
Given a triple pattern query $\mathbf{Q}$ and a KG, an answer for the query, denoted by A, is a mapping of the variables in $\mathbf{Q}$ to values in the KG such that the application of this mapping to each triple pattern $q_i \in \mathbf{Q}$, denoted $A(q_i)$, results in a triple in the KG. The set of all the answers to a query is denoted by the set, $\mathbf{A}$. That is,

$$\mathbf{A}(q_i) = \{A(q_i) : A \in \mathbf{A}\} \quad (1)$$

Definition 2.4. **Score of an answer**
The relative score of a triple t which matches the triple pattern q is denoted by $S(t|q)$ and is computed as follows:

$$S(t|q) = \frac{S(t)}{\max\limits_{t_i \in \mathbf{A}(q)} (S(t_i))} \quad (2)$$

The value ranges between 0 and 1. The score of an answer A to a query $\mathbf{Q}$ is the aggregation of the relative scores of the triples resulting from applying the answer mapping to each triple pattern q_i in the query. That is,

$$S(A|\mathbf{Q}) = \sum_{q_i \in \mathbf{Q}} S(A(q_i)|q_i) \quad (3)$$

The triple and answer scores have been studied previously in [10, 18, 31, 42].

Definition 2.5. **Weighted relaxation rule**
A weighted relaxation rule r is a triple $r = (q, q', w)$ which implies that triple pattern q can be relaxed to q', and $w \in [0, 1]$ denotes the reduction in scores of the triples matching the relaxed triple pattern. Automatic computation of relaxations and the corresponding weights have been studied in [10, 42].

For example, ⟨`?x 'rdf:type' <singer>`⟩ could be relaxed to ⟨`?x 'rdf:type' <vocalist>`⟩ with a weight of 0.8, i.e., r = (⟨`?x 'rdf:type' <singer>`⟩, ⟨`?x 'rdf:type' <vocalist>`⟩, 0.8).

Definition 2.6. **Relaxed Query**
Given a query $\mathbf{Q}$ and a relaxation $r = (q, q', w)$, we say that r applies to $\mathbf{Q}$ if $q \in \mathbf{Q}$. The result of applying r to $\mathbf{Q}$ is a new query $\mathbf{Q}' = (\mathbf{Q} \setminus q) \cup q'$ called the relaxed query. The relaxed query so obtained can further be relaxed by relaxing any of $\mathbf{Q}' \setminus q'$ triple patterns. The score of an answer A obtained through relaxation r applied to a query $\mathbf{Q}$ is equal to $w \times S(A|\mathbf{Q}')$.The score is reduced further for each subsequent relaxation in a similar manner. Since the same answer could be obtained from multiple relaxed queries, the score of an answer A with respect to the original query and a space of possible relaxations is defined as the maximum score obtained through any (relaxed) query,

$$S(A) = \max_{\mathbf{Q}'} (w \times S(A|\mathbf{Q}'))$$

, where $w = 1$ for the original query, $\mathbf{Q}$.

2.1 Non-Speculative Query Processing (NSpec-QP)

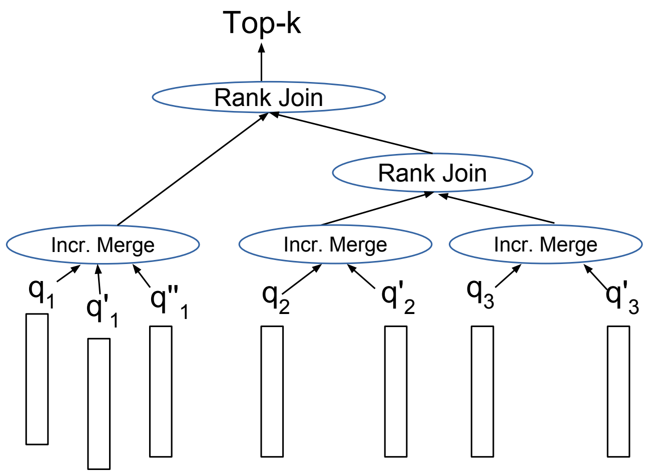


Figure 1: Query plan generated by NSpec-QP for the query $Q = \{q_1, q_2, q_3\}$. One incremental merge operator is required for each triple pattern and its relaxations. A rank join operator takes in two sorted lists and produces a ranked list of (partial) answers from the join.

As mentioned in the Introduction, we can compute results from *all* relaxations simultaneously using two top-k operators: *Incremental Merge* [35] and *Rank Join* [19]. The execution strategy is essentially a variant of the Fagin's NRA algorithm [11]. It computes the exact top-k as it computes all applicable relaxations. It has been used by systems supporting relaxations such as TriniT [42].

Given the query $Q = \{q_1, q_2, q_3\}$, and the relaxations, $r_1 = (q_1, q_1', w_1), r_2 = (q_1, q_1'', w_2), r_3 = (q_2, q_2', w_3)$ and $r_4 = (q_3, q_3', w_4)$, Figure 1 shows the query plan generated using this approach. Incremental Merge is used to efficiently scan the list of matches to a triple pattern and all its relaxations to output only one merged list sorted in descending order of scores. Each of the three incremental merge operators in the example takes as inputs the sorted lists of matches[2] for each triple pattern, q_1, q_2 and q_3 and their relaxations, each multiplied by their relaxation weights. Each of them outputs a combined sorted list of triples for each triple pattern along with its relaxations. The rank join computes a join of the two sorted inputs in an incremental manner until enough results have been produced, while minimising the number of answers read from each list to get top-k answers. This helps avoid computing the entire join and then sorting over it. The inputs for Rank Joins are either the outputs of Incremental Merges or Rank Joins. Both operators use priority queues for already seen answers and maintain upper bounds to estimate scores of other answers that can be obtained by reading further into the lists at any given point. This avoids accessing entire lists of (partial) answers and aids early termination.

However, the top-k operators still process relaxations from *all* the triple patterns, many of which do not contribute triples towards the top-k answers. Our technique aims to eliminate this inefficiency.

[2]Recall that each triple is associated with a score.

3 SPEC-QP – THE SPECULATIVE FRAMEWORK FOR OPTIMIZING QUERY PLANS

We propose Spec-QP, a speculative query planning framework which speculates the useful relaxations for a query. It uses a predictor to predict whether the relaxations of a triple pattern in a query are likely to be required for producing the top-k answers. We eliminate the relaxations for those triple patterns which are predicted to be not required. The predictor uses an expected score estimator to estimate the expected scores at given ranks for a (relaxed) query and then, predicts the requirement of relaxations of a triple pattern for getting top-k answers based on the estimates. The estimator is based on precomputed statistics about the distribution of the scores for triple pattern matches. We first describe the estimator and then give details of the speculative planning approach.

3.1 Expected score estimator

The expected score estimator is based on order statistics and estimates the expected scores at given ranks for the original as well as relaxed queries. These are used by the query planner to predict the presence of answers from a relaxation in top-k.

The m matching triples for a triple pattern, q_i, have scores represented by the independent and identically distributed (i.i.d.) random variables $X_{i1}, X_{i2}, ..., X_{im}$, each with a common distribution, $f_i(x)$. Here, $f_i(x)$ is the probability distribution for the scores of the answers for a triple pattern (or relaxation), q_i, from the *KG*. The cumulative distribution function (cdf) is represented by $F_i(x)$. The set $\{X_{i1}, X_{i2}, ..., X_{im}\}$ is a sample of size m taken from the distribution $F_i(x)$. The set of the observed values of answer scores $\{x_{i1}, x_{i2}, ..., x_{im}\}$ of random variables $\{X_{i1}, X_{i2}, ..., X_{im}\}$ is called a realization of the sample. $X_{i(1)}, X_{i(2)}, ..., X_{i(m)}$ are random variables resulting from arranging the values of each of $X_{i1}, X_{i2}, ..., X_{im}$ in increasing order, and $X_{i(j)}$ is called the j^{th} order statistic.

Given these random variables and their distributions, we need to estimate the score distribution for the answers of the query, Q. $X_{Q1}, X_{Q2}, ..., X_{Qn}$ are the random variables representing the scores of the n answers to the query, Q (possibly composed of a single triple pattern). $X_{Q(1)}$ is the first order statistic corresponding to the lowest scoring answer among all the n answers of Q, and $X_{Q(n)}$ is the n-th (or largest) order statistic corresponding to the highest scoring answer (ranked 1). A relaxed answer would appear in top-k only when its expected highest score ($X_{Q'(n')}$) amongst its n' answers exceeds the expected k^{th} highest score of the original query ($X_{Q(n-k+1)}$[3]). In order to compute the expected value at a given rank, we use the result given in [7]:

For i.i.d. random variables, $X_1, X_2, ..., X_m$ each with a common distribution, $f(x)$, the expected value of i^{th} order statistic, $X_{(i)}$ can be approximated as $E(X_{(i)}) \approx F^{-1}(\frac{i}{m+1})$ where $F(x)$ denotes the cdf and m is the size of the sample.

Using this, the expectation of $X_{Q(i)}$ can be approximated as $E(X_{Q(i)}) \approx F_Q^{-1}(\frac{i}{n+1})$ where $F_Q(x)$ denotes the cdf of the scores for the answers to the query, Q and n is the no. of answers of Q .

We now give the details of the construction of the probability density function (pdf) of these random variables.

[3]Note that it is $n - i + 1$ and not i since the n^{th} order statistic represents the highest value with rank 1.

3.1.1 Score Distributions for the matches to Triple Patterns: For every triple pattern q_i in the *KG*, we store the following 4 precomputed statistics about the scores $\boldsymbol{\sigma^i}$ of the matching triples:

- m_i: the total number of triples matching the triple pattern.
- σ_r^i: the score of the answer at rank r where r represents the rank within which 80% of the score mass is contained for the triple pattern matches.
- S_r^i: the cumulative score of the answers over all the ranks 1 through r.
- $S_{m_i}^i$: the cumulative score of the answers over all the ranks 1 through m_i.

We now estimate the score distribution for answers to triple pattern q_i.[4] $f_i(x)$ and $F_i(x)$ are used to denote the pdf and cdf respectively.

The pdf can be modelled as a 2-bucket histogram in the following way:

$$f_i(x) = \frac{S_{m_i}^i - S_r^i}{S_{m_i}^i} \frac{1}{\sigma_r^i} \text{ for } 0 \le x < \sigma_r^i$$
$$\frac{S_r^i}{S_{m_i}^i} \frac{1}{1-\sigma_r^i} \text{ for } \sigma_r^i \le x \le 1$$

This pdf gives us the following cdf:

$$F_i(x) = ax \text{ for } 0 \le x < \sigma_r^i$$
$$bx + c \text{ for } \sigma_r^i \le x \le 1$$

where

$$a = \frac{S_{m_i}^i - S_r^i}{S_{m_i}^i} \frac{1}{\sigma_r^i} \text{ and } b = \frac{S_r^i}{S_{m_i}^i} \frac{1}{1-\sigma_r^i}$$
$$\text{and } c = \frac{S_{m_i}^i - S_r^i}{S_{m_i}^i} - \frac{S_r^i}{S_{m_i}^i} \frac{\sigma_r^i}{1-\sigma_r^i}$$

Our technique depends on statistical estimates – specifically, what the score of the k^{th} result is for a specific (original) query. This estimate can be made as accurate as possible, provided sufficient space and time are available. At one extreme, we can assume uniform distribution for the score of a triple pattern and at the other, we could consider every single data point (the actual distribution). In particular, if we had every single data point as a single-bucket histogram, that would then give us 100% accuracy on the k^{th} score. But this would be no different from actually computing the result. Our solution of 2-bucket approximation strives for the sweet spot on this spectrum and is based on the fact that even though datasets are different, their score distributions typically follow a power law distribution – a "fat" head, and a "long" tail. The narrow and tall bucket represents the interval which has 80% of the score mass. The longer bucket represents the long tail having only 20% of the score mass.

3.1.2 Score Distribution for the Triple Pattern Query: The score of an answer for the triple pattern query is the sum of the scores of the individual triples in the answer. Since each triple is contributed by one triple pattern in the query and we have estimates for their scores, we can estimate the scores for answers to the query using the following approach.

[4]Note that the ranks will not be explicitly reflected here, it is just the distribution of the answer score values from which each score in $\{X_{i1}, X_{i2}, \ldots, X_{im}\}$ is assumed to be independently sampled.

```
Input: The query $\mathbf{Q} = \{q_1, q_2, \ldots q_n\}$.
Output: The query plan, QP
QP ← {{Q1}}, where Q1 = Q
$f_Q(x) \leftarrow f_1 * f_2 * .. * f_n(x)$
Get $E_Q(k)$ from "expected score estimator".
for $q_i \in$ Q do
    $q_i' \leftarrow$ top-weighted relaxation for $q_i$
    $Q' \leftarrow Q - \{q_i\} \cup \{q_i'\}$
    $f_{Q'}(x) \leftarrow f_1 * f_2 * \ldots * f_i' * .. * f_n(x)$
    Get $E_{Q'}(1)$ from "expected score estimator".
    if $E_{Q'}(1) > E_Q(k)$ then
        QP = {{Q1} − {$q_i$},{$q_i$}}
    end
end
return QP
```
Algorithm 1: PLANGEN generates the query plan.

Let us assume our triple pattern query, $\mathbf{Q} = \{q_1, q_2\}$. $\{X_{11}, X_{12}, \ldots, X_{1m}\}$ represents the m triples matching q_1 and $\{X_{21}, X_{22}, \ldots, X_{2m'}\}$ represents the m' triples matching q_2. The scores for triples matching these triple patterns have the distributions $f_1(x)$ and $f_2(x)$ respectively, as defined earlier. The scores of **Q**'s n answers are represented by the random variables $X_{Q1}, X_{Q2}, \ldots, X_{Qn}$. Each of these is a sum of two random variables, one from $\{X_{11}, X_{12}, \ldots, X_{1m}\}$ and another from $\{X_{21}, X_{22}, \ldots, X_{2m'}\}$. The pdf for the sum of the random variables is given by the convolution of the two individual pdfs, $f_1 * f_2(x)$. Hence, the pdf for the scores of the answers to the query is given by the convolution of the pdf's of the scores for matches to the constituent triple patterns. The resulting pdf is a multi-piece-wise linear function. Given the number of results in the combined distribution, $n = m_{12}$, we can estimate σ_r^{12}, S_r^{12} and S_n^{12} using the expected score computation from order statistics. This again results in a two-bucket histogram for the distribution of the scores of the answers to the query. For the computation of m_{12}, we use the estimates for join selectivity[5], ϕ_{12} as $m_{12} = m * m' * \phi_{12}$. For three or more triple patterns, we repeat the above process the required number of times to get the final histogram representing the score distribution for answers to the query.

3.1.3 Score prediction. Once we have constructed the pdf and cdf representing the scores for the answers of a given query, we can estimate the expected score, $X_{Q(n-i+1)}$ at a given rank i as $E(X_{Q(n-i+1)}) \approx F_Q^{-1}(\frac{n-i+1}{n+1})$ where $F_Q(x)$ denotes the cdf of the query answer scores and n is the no. of answers for **Q**. Given these estimates for scores at various ranks, we now generate the query plan.

3.2 Query Planning

Query Plan: Given a query **Q**, a query plan consists of subsets of triple patterns $\mathbf{Q_1}, \mathbf{Q_2}, \ldots, \mathbf{Q_s}$ where

i. each $\mathbf{Q_i}$ consists of one or more triple patterns from **Q**,
ii. the $\mathbf{Q_i}$'s are pairwise disjoint, and
iii. the union of $\mathbf{Q_i}$'s equals **Q**.

For example, a query plan for the query $\mathbf{Q} = \{q_1, q_2, q_3\}$, will be $\{\{q_1, q_3\}, \{q_2\}\}$. The singletons correspond to the triple patterns which require relaxations.

[5]Traditional database systems use multiple heuristics to estimate join selectivity. For the purpose of this work, we have taken exact join selectivity values.

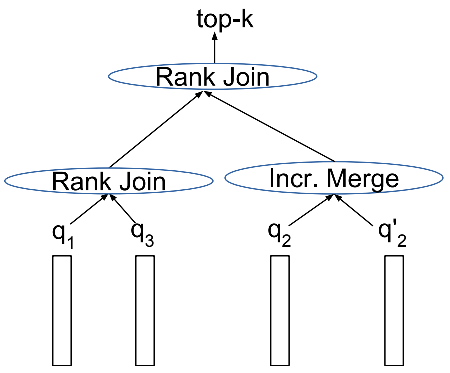


Figure 2: Query Plan when Q = $\{q_1, q_2, q_3\}$ and only q_2's relaxations are predicted to be in top-k. Only q_2 requires an incremental merge. q_1 and q_3 are joined using a rank join over the sorted answer lists for each of them. One rank join is required to join these results.

3.2.1 Query plan generation. The key task in the planning approach is to identify the triple patterns whose relaxations do not contribute towards the top-k answers. We save on computations over such triple patterns by never processing their relaxations. For each triple pattern, only the top-weighted relaxation has the highest top score due to normalization of scores as per Definition 2.4, i.e, the top score from each relaxation is equal to its weight. Hence, we need to check only the top-weighted relaxation for each triple pattern for its potential to contribute answers towards top-k.

Given a query **Q** and the score distribution for each triple pattern, the query plan is generated as outlined in Algorithm 1. PLANGEN first predicts the requirement of relaxations for each triple pattern. For prediction, the query planner uses the "expected score estimator" described in Section 3.1, which gives estimates of the expected scores at k^{th} rank for the original query, $E_{\mathbf{Q}}(k)$ and top rank for the highest weighted relaxed query, $E_{\mathbf{Q}'}(1)$ (for a given triple pattern at a time). If the topmost score from the relaxed query obtained by relaxing a given triple pattern exceeds the k^{th} score from the original query, it predicts that the triple pattern's relaxations are required. Note that our estimator takes into account join score distributions and join cardinalities for estimating the expected score for a given query.

The query plan, **QP** returned by PLANGEN will have at most one subquery, $\mathbf{Q_1}$ of size > 1, called the "join group" (non-relaxed triple patterns), the rest will be only singletons (triple patterns to be relaxed).

3.2.2 Query Execution. Given a speculative query plan **QP** = $\{\mathbf{Q_1}, \mathbf{Q_2}, .., \mathbf{Q_s}\}$ with s subsets generated by the speculative query planner, we execute it in the following manner.

(1) The join group, $\mathbf{Q_1}$ is executed as (left-deep) rank joins over the answer lists (sorted by score) for each triple pattern. Note that, none of the triple patterns in this group are relaxed.
(2) The singletons are processed by Incremental Merge operator for each.
(3) Rank joins are performed over the join group and singletons.

Figure 2 illustrates this approach for the query **Q** = $\{q_1, q_2, q_3\}$, when we predict that only q_2 needs to be relaxed. The query plan to be executed is $\{\{q_1, q_3\}, \{q_2\}\}$. We use a rank join to compute the join between sorted lists of matches for q_1 and q_3 and require incremental merge only for q_2 and its relaxations. One rank join is required to join these results. The equivalent NSpec-QP plan for this query will be $\{\{q_1\}, \{q_2\}, \{q_3\}\}$, i.e., all triple patterns occur as different subsets and each of them are processed by incremental merges followed by rank joins over all these incremental merges (refer Figure 1).

4 EXPERIMENTAL EVALUATION

This section discusses the experimental evaluation performed for demonstrating the performance of the speculative planner.

4.1 Setup

Baseline. We compare Spec-QP with the NSpec-QP system (refer Section 2.1) which involves Incremental Merges for relaxations and Rank Joins for joins. It processes *all* the relaxations and outputs the true top-k. Existing works which focus on optimized computation of top-k joins without relaxations or on determining relaxations for user queries are orthogonal to our work. The scoring scheme used by the existing systems supporting relaxations do not use fine-grained scores (scores for individual triples). For our setting, NSpec-QP is the closest baseline to the best of our knowledge. We have not shown comparisons with the naive method (i.e., compute answers to all combinations of relaxations and then sort them to get top-k) because it is obvious that it is the most inefficient technique.

Datasets. We have evaluated over the following two datasets:

i. Extended Knowledge Graph (XKG)[42]:
   a. Format: RDF format dataset consisting of YAGO2s triples and "textual" content triples constructed from Clueweb by using OpenIE techniques and Named Entity Disambiguation (NED). The triple scores for YAGO2s triples are equal to the number of inlinks into the subject. Triple score for Clueweb data is equal to the number of times a particular triple was encountered during extraction.
   b. Size: XKG has about 105 million triples.
ii. Twitter:
   a. Format: Constructed from trending tags over the month of April 2017 using Twitter Streaming API. The triples are of the form: $\langle tID, hasTag, T\rangle$ where tID is the unique ID for a tweet containing term T. The score for each triple is equal to the number of retweets for the tweet in that triple.
   b. Size: 18 million unique triples.

Queries and relaxations. The evaluation queries and relaxations for the datasets are as follows:

i. XKG: We evaluated on 65 queries which were manually constructed so as to have non-empty result sets. Each query had 2-4 triple patterns and each triple pattern had at least 10 relaxations. The relaxations were obtained using the scheme outlined in [42].
ii. Twitter: A query over this dataset queries for IDs of those tweets which have all the queried terms. For example, the following query queries for IDs of all those tweets which contain the terms '`#intoyouvideo`', '`#ariana`' and '`dangerous`':

```
SELECT ?s WHERE{
    ?s <hasTag> <#intoyouvideo>.
    ?s <hasTag> <#ariana>.
```

k	XKG	Twitter
10	0.7	0.72
15	0.88	0.78
20	0.91	0.8

Table 2: Precision over each dataset.

```
    ?s <hasTag> <dangerous>
}
```

The testset of 50 queries was constructed manually using combinations of most frequent tags and terms. Each query had either 2 or 3 triple patterns, with each triple pattern having at least 5 relaxations. The relaxations were generated using the co-occurrence frequencies i.e. the relaxation weight, w for the relaxation, r = (⟨`?s <hasTag>` <T_1>⟩,⟨`?s <hasTag>` <T_2>⟩,w) will be equal to:

$$w = \frac{\#tweets\_having\_T_1\_and\_T_2}{\#tweets\_having\_T_1}$$

For example, a possible relaxation for `<#intoyouvideo>` is `<video>`.

Note that the number of results decrease on increasing the number of triple patterns in a query. Due to this, we have restricted our testsets to have only 2-4 triple patterns' queries with non-empty result sets. Also, even though each triple pattern has at least 5-10 relaxations, relaxing only one or two triple patterns alone generates about 100 additional answers. Our planner aims to be able to predict the useful relaxations.

Metrics. We measure the following metrics for each query to demonstrate the quality and efficiency of our technique:

i. Quality:
   a. Precision: The fraction of true top-k results (of NSpec-QP) in the top-k results of Spec-QP. Note that precision and recall have identical values in our setup, because they have the same denominator k.
   b. Prediction accuracy: The number of queries for which we could identify all and only correct relaxations.
   c. Score error: The average of absolute error for Spec-QP vs. NSpec-QP top-k scores, i.e.,
   $\frac{1}{k}\sum_{i=1..k}\left|score_i^{NSpec-QP} - score_i^{Spec-QP}\right|$
   We also note the standard deviation.
ii. Efficiency:
   a. Runtimes: We measure the time taken to plan and execute each query.
   b. Memory used: We measure the total no. of answer objects created as it directly corresponds to the amount of search space traversed to arrive at top-k answers. This number includes all the intermediate answer objects encountered by Incremental Merges and Rank Joins.

System setup. The experiments were conducted on a Dell Blade server with 24 Intel(R) Xeon(R) CPU E5-2420 @ 1.90GHz processors and 32GB RAM. The database engine was used to retrieve the matches for triple patterns in sorted order. Each query was evaluated using both the techniques- NSpec-QP and Spec-QP, over two database engines, postgresql-9.5 and Virtuoso, for three values of k, namely 10, 15 and 20. To have a warm cache, we conducted 5 consecutive runs for each query and considered the average of the last 3 runs for each technique.

4.2 Quality evaluations

We first discuss the quality of results obtained by Spec-QP and then provide the statistics for runtimes and memory consumptions. *Note that the quality of results will be the same over any database engine as it depends only on the accuracy of our speculative technique.*

4.2.1 Precision. The precision values for the datasets are given in Table 2. The precision is about 0.7-0.9 for both the datasets, i.e., about 80% of the answers belonged to true top-k. Also, since the answers are sorted according to the scores, the answers outside the true top-k appeared at lower ranks. That is, for a query having a precision value of 0.8 for k=10, top-8 answers belonged to the true top-10.

4.2.2 Prediction Accuracy. A detailed analysis of the number of queries for which we could predict the correct relaxation(s) over each dataset is given in Table 3. Each query required some triple patterns to be relaxed to generate top-k answers. The prediction accuracy is at least 70% for all types of queries over XKG and queries requiring 3 relaxations over Twitter. As the value for k was increased, queries increasingly required relaxations to generate sufficient answers. For Twitter, most of the queries required all triple patterns to be relaxed. This is due to the absence of sufficient triples corresponding to each term and fewer relaxations (predicate is not relaxed) for each triple pattern. Nevertheless, we were able to identify the requirement of all the relaxations in such a scenario.

4.2.3 Average score error. To judge the quality of approximate results returned by Spec-QP, we computed the score deviations of the approximate answers at each rank given by Spec-QP from the true top-k. The average score deviations for various values of k are given in Table 4. The percentages in brackets show the average percentage deviation from the original scores. Note that the maximum possible score for an answer to a 2 triple pattern query can be 2, for a 3 triple pattern query, it will be 3 and so on.[6]

XKG. Even though k=10 has lowest precision, the score deviations from true top-k answers are low (about 0.1 for 2 triple pattern queries). That is, for a query with 2 triple patterns if the actual answer at a given rank has a score of 1.5, the score of the approximate answer would be about 1.4. The deviations are even lower (only about 0.01) for higher values of k and tolerable for achieving faster runtimes.

Twitter. There is only one 2 triple pattern query that required both triple patterns to be relaxed but had a wrong speculation of relaxations for all values of k. However, its score deviation is constant over all values of k as it has only 11 results (including relaxations). The deviations are only 0.5 for 3 triple pattern queries with $k = 10$, which is only 16% deviation from the original scores. The deviations for higher values of k are very low being only 6% in the best case. For k=20, for a query with 3 triple patterns if the actual answer at a given rank has a score of 2.5, the score of the approximate answer is about 2.32.

4.3 Efficiency evaluations

The average runtimes and memory values over PostgreSQL and Virtuoso for XKG grouped by the number of triple patterns in

[6]This is because the maximum score for a matching triple for each triple pattern can be 1.

Dataset	XKG			Twitter		
k	10	15	20	10	15	20
queries requiring 1 relaxation	5(6)	5(5)	-(-)	-	-	-
queries requiring 2 relaxations	21(30)	22(26)	18(19)	1(2)	1(2)	1(2)
queries requiring 3 relaxations	12(18)	16(19)	27(31)	35(48)	38(48)	39(48)
queries requiring 4 relaxations	7(11)	14(15)	14(15)	-	-	-

Table 3: Prediction accuracy for various values of k grouped by the number of triple patterns requiring relaxations in the queries to generate true top-k results. The number indicates the number of queries for which Spec-QP could identify all and only these relaxations. The numbers in brackets show the total number of such queries.

Dataset	XKG			Twitter	
k \ #TP	2	3	4	2	3
10	0.1(5%)±0.1	0.2(8%)±0.3	0.1(3%)±0.2	0.16(8%)±0.0	0.5(16%)±0.5
15	0.08(4%)±0.08	0.1(3%)±0.2	0.01(1%)±0.04	0.16(8%)±0.0	0.32(10%)±0.3
20	0.07(4%)±0.06	0.07(2%)±0.1	0.01(1%)±0.03	0.16(8%)±0.0	0.18(6%)±0.1

Table 4: Avg. score deviations for the approximate top-k from the true top-k grouped by the number of triple patterns (#TP) in the queries. The percentages in brackets show avg. percentage deviation from the score of the true answer at that rank.

the queries and the number of relaxations required by them have been given in Figures 3 and 4 respectively. The graphs for Twitter are given in Figures 5 and 6.

4.3.1 Runtime comparisons. It is evident from the runtime graphs (refer Figures 3 and 5) that Spec-QP is faster than NSpec-QP in all cases. It avoids unnecessary computation of *all* relaxations when only few relaxations are capable of giving top-k answers. Most of the queries require only 2 or 3 relaxations (Refer Table 3) to produce top-k answers and Spec-QP either identifies the correct relaxation(s) or gives good quality approximate results. Also, fewer the number of relaxations required, faster is Spec-QP over NSpec-QP. For k=15 and k=20, the gain margin lowers but Spec-QP is still faster than NSpec-QP. This is because on seeking more answers, the original query is insufficient to get top-k answers and needs multiple relaxations. It is especially prominent for XKG queries with 4 triple patterns; for k=15 and k=20, none of the queries could get top-k answers with less than 3 relaxations. The difference in the runtimes however clearly demonstrates the savings achieved by eliminating the requirement of even 1 relaxation. In particular, Spec-QP outperforms NSpec-QP by a factor of upto 1.5 for queries with 3 triple patterns.

Note that the key optimization for sorted access, in any database engine, is to use ordered index scans. This is what PostgreSQL does too. Also, even if the underlying database system is further optimized for sorted results, both techniques would benefit and therefore the gains of Spec-QP over NSpec-QP would be of the same order. PostgreSQL is faster than Virtuoso as it uses indices intensively for optimized retrieval. We have shown results over Virtuoso to demonstrate the practical applicability of our technique over any quad store.

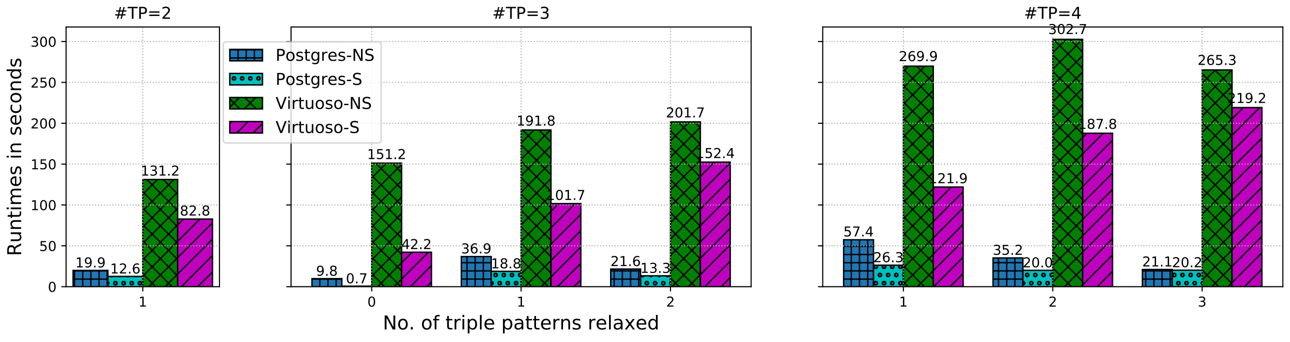
4.3.2 Memory requirement comparisons. We measured the total number of answer objects created as it directly corresponds to the amount of search space traversed to arrive at top-k answers. This number includes all the intermediate answer objects encountered by the incremental merges and rank joins.

The memory comparison graphs are given in Figures 4 and 6 for XKG and twitter respectively. We found that NSpec-QP consumes the most memory for all the cases. This is because it traverses a significant amount of the search space, consisting of the original query and all its possible relaxations, in order to compute the top-k. Spec-QP consumes upto 2-3x less memory to compute top-k answers as it prunes a significant amount of the search space. The savings by Spec-QP is achieved by eliminating the need for processing relaxations of triple patterns which do not contribute triples towards top-k answers. This is particularly evident for the cases where the queries with 3 triple patterns require only 1 or 2 relaxations. The memory requirements reduce by a factor of 2.5 over both the datasets.

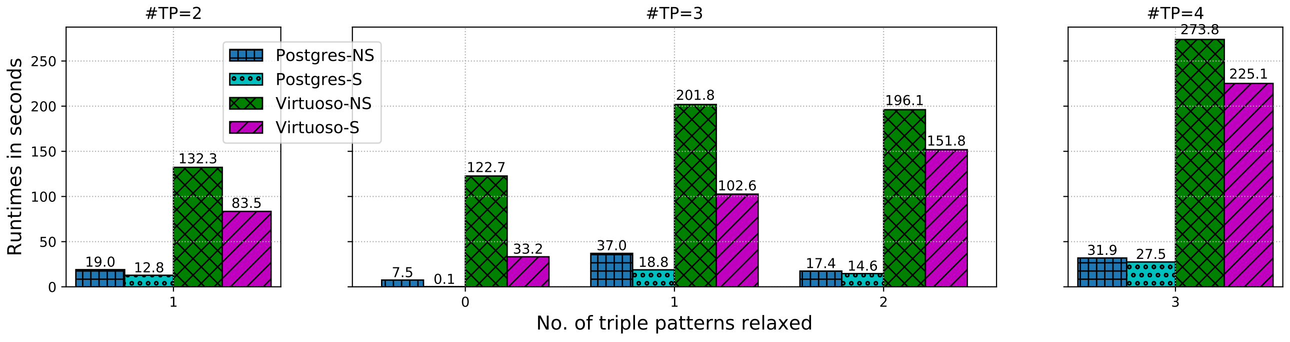
4.4 Discussion and remarks

We showed that Spec-QP prunes unnecessary relaxations and has faster response times over the baseline for various values of k. It predicts the correct relaxations 70-80% of the time with good approximations for answers outside top-k. Our technique depends on statistical estimates – specifically, what the score of the k^{th} result is for a specific (original or relaxed) query. Our solution of 2-bucket approximation strives for the sweet spot between assuming a uniform distribution and the actual distribution (every single data point in individual buckets of the histogram), and is built on the fact that even though datasets are different, their score distributions typically follow a power law distribution – a "fat" head, and a "long" tail.

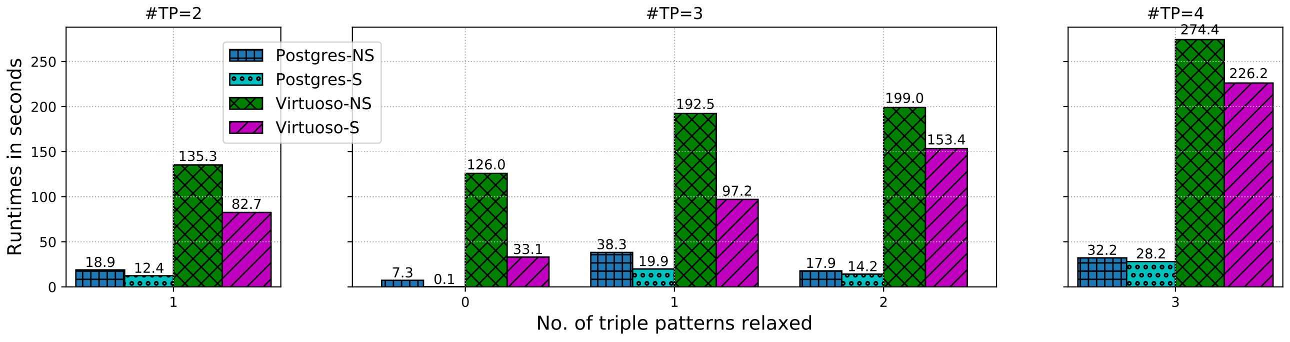
The strategic pruning of relaxations using our model also reduces the search space traversed and in turn, the memory requirements for each query. Spec-QP is especially useful for servers where the total resource consumption per query is inversely proportional to the achievable throughput. The cost per query determines the cost of running the server for a given load. For instance, the queries having 3 triple patterns have 1.5x faster response times and 2.5x less memory requirements resulting in an overall 4x gain. This implies that the server can run the service with 4x less money. Hence, Spec-QP provides cost-efficient support for flexible querying using relaxations over SPARQL query



(a) Runtimes for k=10.



(b) Runtimes for k=15.



(c) Runtimes for k=20.

Figure 3: Runtimes comparisons over XKG queries for k=10, 15 **and** 20 **grouped by the no. of triple patterns (#TP) in the query and the number of relaxations required. All the legends in the graphs for efficiency have 'NS' for NSpec-QP and 'S' for Spec-QP.**

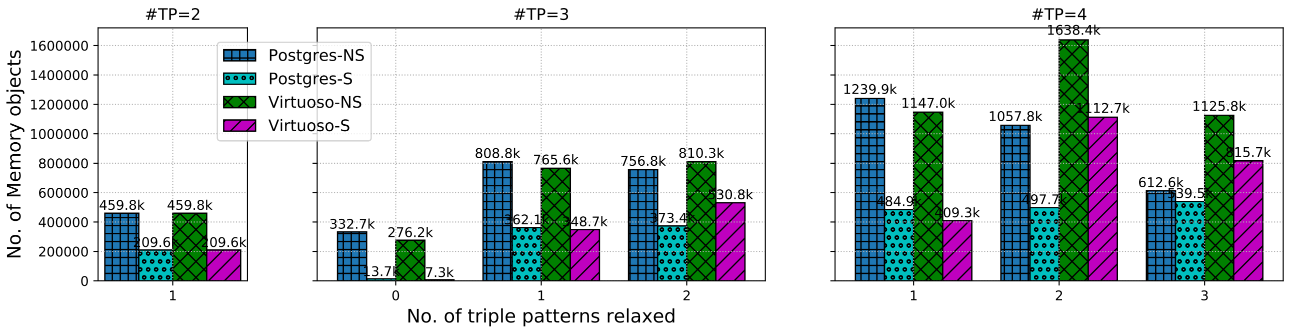
engines. This, in turn, aids effective exploration of knowledge graphs by new users.

5 RELATED WORK

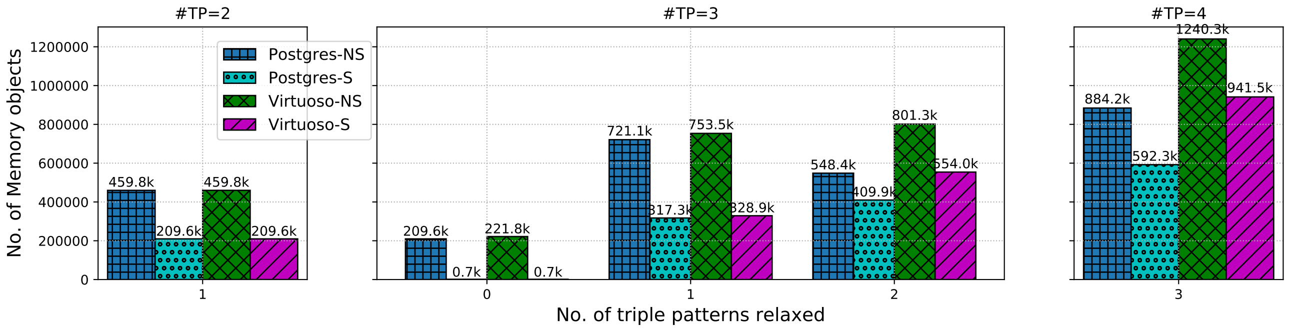
Top-k query processing

FRPA [13] and Hash Rank-Join (HRJN*) [20] represent the state-of-the-art relational rank-join algorithms. HRJN* has been shown to perform well in practice, however, FRPA showed that it was not instance-optimal for a variant of the rank join problem that they considered. HRJN[21] is based on ripple join algorithm. It maintains two hash tables in-memory for storing the input tuples seen so far, the stored input tuples are used for finding join results. These results are then given as inputs to a priority queue, which outputs them in the order specified by the ranking function. Nested Loops Rank Join (NRJN) [19] is similar to HRJN except that unlike HRJN it does not store input tuples, but rather follows a nested-loop strategy. Pull/Bound Rank Join (PBRJ) [33] is an algorithm template that generalized previous rank join algorithms and provided tight upper bounds. DRJN [8] is an efficient algorithm for computing rank joins in distributed systems. This body of work is orthogonal to our problem.

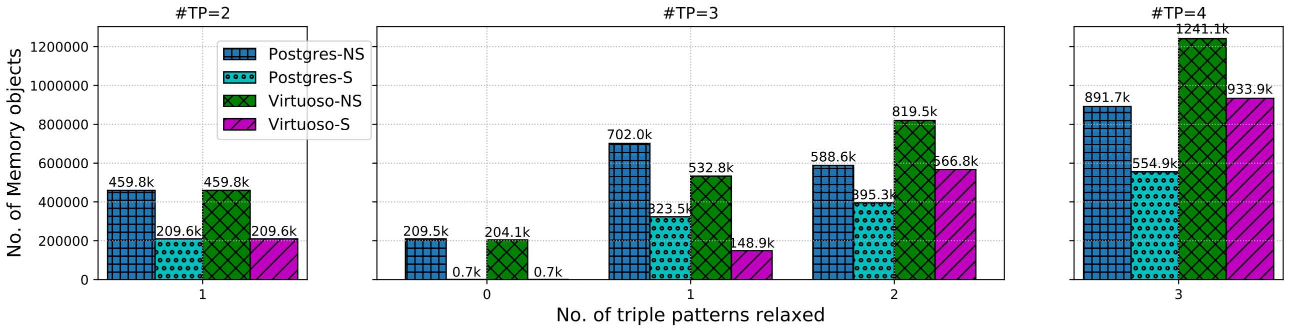
Theobald et. al. [37] dealt with top-k query evaluation for joins over multiple index lists with pruning providing probabilistic guarantees. It uses histograms and dynamic convolutions to predict the top-k. Our case, however differs in that we consider



(a) Memory for k=10.



(b) Memory for k=15.



(c) Memory for k=20.

Figure 4: Memory comparisons over XKG queries for k=10, 15 **and** 20 **grouped by the no. of triple patterns (#TP) in the query and the number of relaxations required. All the legends in the graphs for efficiency have 'NS' for NSpec-QP and 'S' for Spec-QP.**

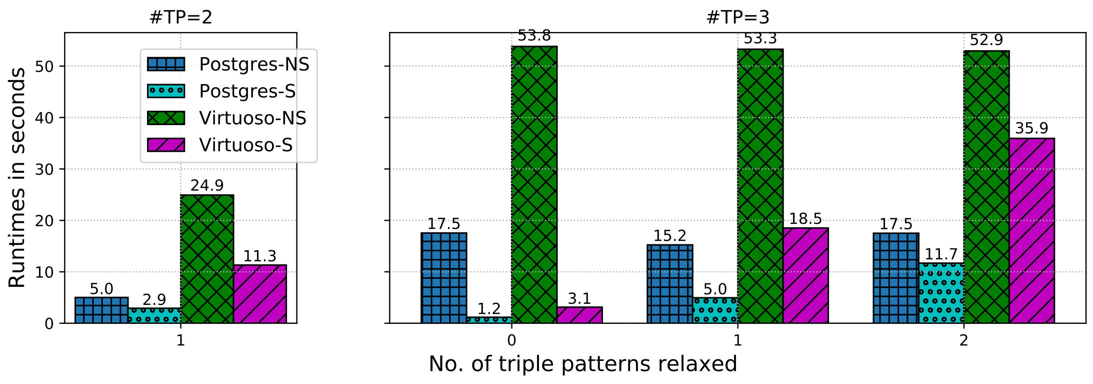
graph structured data and also, support multiple relaxations. The IO-Top-k [4] deals with top-k query evaluation with pruning using sorted access (SA) scheduling. Other works include top-k processing over xml data [36] and for data distributed over multiple nodes [44].

Top-k queries on graphs

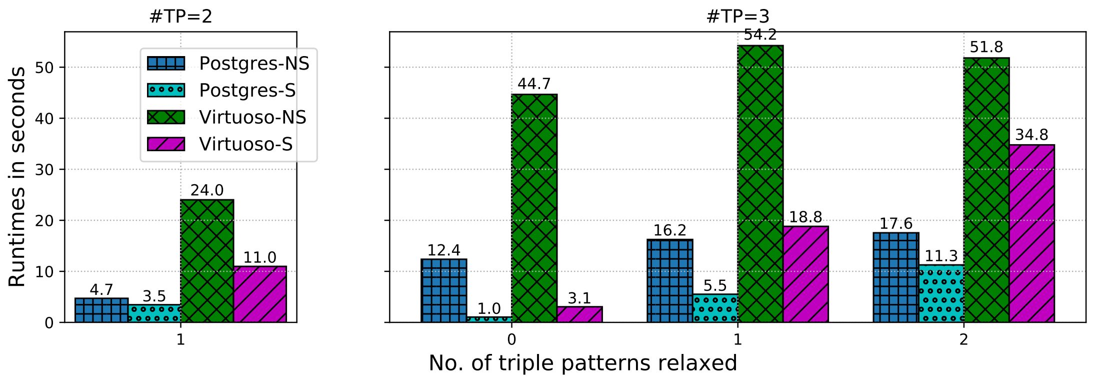
Only few works address the problem of top-k processing over RDF graphs. The SPARQL-RANK framework proposed by [27] makes use of different index permutations used in native triple-stores for fast random access and early termination. Another framework introduced by Wang et al. [41] used MS-tree-based filtering and pattern-matching functions to evaluate top-k answers. The work in [43] uses an approach similar to HRJN[21] for computing top-k star joins. However for RDF data, SPARQL-RANK showed experimentally that it outperformed HRJN. The performance gain was attributed to the unsorted nature of numerical attributes present in indexes build by RDF engines. QUARK-X [25] proposes using extra indexes and metadata to process top-k queries on RDF graphs. All of these works however do not optimize over possible relaxations.

Query Reformulation in IR

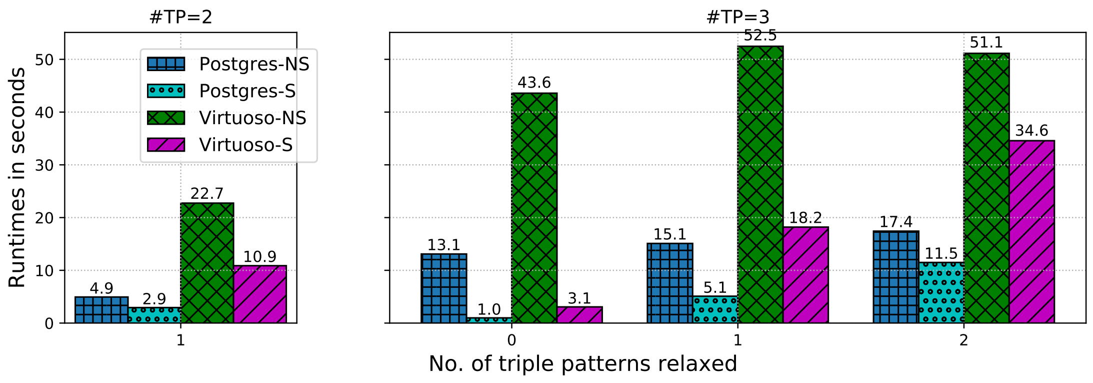
Various strategies have been proposed to reformulate queries in IR over documents. These include measures of query similarity [3], or using summary information included in the query-flow graph [1]. Another approach by Hristidis et. al. [16] relies on



(a) Runtimes for k=10.



(b) Runtimes for k=15.



(c) Runtimes for k=20.

Figure 5: Runtimes comparisons over Twitter for k=10, 15 and 20 grouped by the no. of triple patterns (#TP) in the query and the number of relaxations required. All the legends in the graphs for efficiency have 'NS' for NSpec-QP and 'S' for Spec-QP.

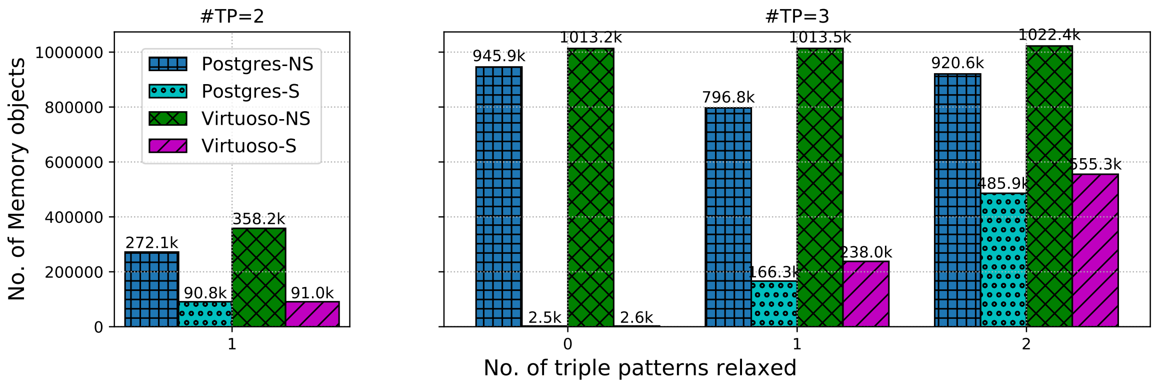
suggesting keyword relaxations by relaxing those which are least specific based on their idf score. These reformulations can be used as relaxations for our setting.

Faceted Search (Many answers problem)

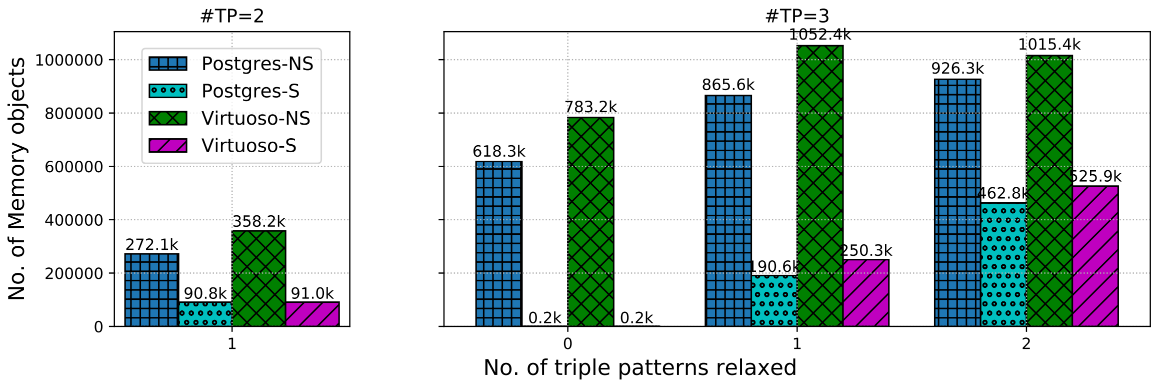
A related optimization problem is the one encountered when we have many-answers, i.e. those where given an initial query that returns a large number of answers, the objective is to design an effective drill-down strategy to help the user find acceptable results with minimum effort [22, 26, 32]. We solve a related problem, where we try to solve both empty-answer and many-answers problem in an efficient manner by generating additional scored answers using relaxations.

Query Relaxation in relational databases

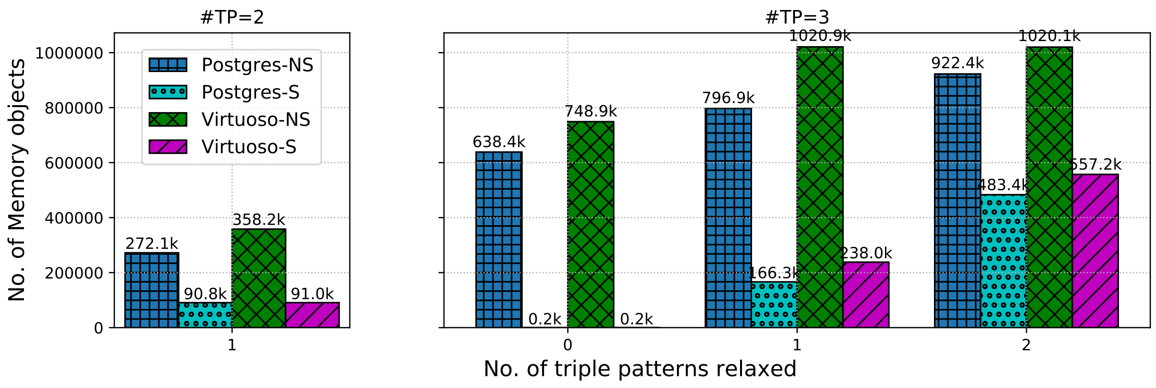
Query relaxation in relational databases is quite common. The work [24] relaxes joins and selections in relational databases by suggesting alternative queries based on the "minimal" shift from the original query. Another work [40] suggests user ranking of the query edges so as to generate relevant differential queries with minimum deviation. "Why Not" queries are studied in [6, 38], where, given a query Q that did not return a set of tuples S that the user was expecting to be returned, they design an alternate



(a) Memory for k=10**.**



(b) Memory for k=15**.**



(c) Memory for k=20**.**

Figure 6: Memory comparisons over Twitter for k=10**,** 15 **and** 20 **grouped by the no. of triple patterns (#TP) in the query and the number of relaxations required. All the legends in the graphs for efficiency have 'NS' for NSpec-QP and 'S' for Spec-QP.**

query Q' that (a) is very similar to Q, and (b) returns the missing tuples S, however the rest of the returned tuples should not be too different from those returned by Q. The paper [28] relaxes one constraint at a time and is interactive. It also tries to minimize the cost by suggesting low cost relaxations which lead to non-empty answers. DebEAQ [39] first tries to debug why the query is returning empty answer and then tries to relax it with minimum change to the original query. It is also limited only to property graphs.

Query Relaxation over graphs

The closest to our works are those which deal with relaxations over graphs. The work in [15, 29–31] considers query relaxation for conjunctive regular path queries. Users are allowed to specify query predicates which can have approximations and/or relaxations (using *APPROX* and *RELAX* operators respectively) during query time. The system then computes the approximations/relaxations with their relative evaluation costs to support query rewriting. Another work [17] computes approximate answers using a Bayesian network to rank and score relaxed queries. Two algorithms are described in [18]. The first algorithm is based on best-first strategy and relaxed queries are executed in order. They prune relaxations which do not give new results. The other algorithm executes the relaxed queries as a batch and avoids the unnecessary execution cost. The idea of Maximal Succeeding Subqueries (MSSs) is exploited in [14] using Lattice-based and Matrix-based approaches to minimally refine the user query. The scoring scheme used by these existing systems supporting relaxations however, do not use fine-grained scores (scores for individual triples) as in our case. TriniT [42] proposes the notion of eXtended Knowledge Graphs (XKG) with fine-grained scores for triples and allows relaxations for queries over them. It uses a technique similar to NSpec-QP to evaluate the queries.

6 CONCLUSION AND FUTURE WORK

We have proposed Spec-QP, a strategy for top-k query processing in a scenario where a query can have multiple relaxations. To achieve this, we have used a speculative approach for pruning the relaxations which are not likely to contribute answers to the top-k results. The speculation is based on precomputed statistics about the distribution of scores for triple pattern matches. The relaxations of triple patterns predicted to not contribute towards top-k answers are not processed, thereby reducing top-k computations and leading to faster response times and reduced memory requirements.

We have experimented over two real world datasets – XKG and Twitter – to show that Spec-QP is a cost-efficient technique for supporting relaxations. This is especially useful for servers in aiding exploratory querying over knowledge graphs by new users without an exponential increase in the budget. We also demonstrated the practical usability of our technique by implementing it over two popular database engines – PostgreSQL and Virtuoso. As future work, we would like to support more complicated relaxations for the queries like replacing a triple pattern with a chain of triple patterns, etc. Another orthogonal area of work is to find meaningful and useful relaxations for a given triple pattern.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. The first author was partially supported by the Student Exchange program under Indo-German Max Planck Center for Computer Science (IMPECS) and funds from Max Planck Institute for Informatics (MPI-INF) over an internship. She would also like to thank the alumni donors of the CS&E department at IIT Delhi for supporting her research.

REFERENCES

[1] Aris Anagnostopoulos, Luca Becchetti, Carlos Castillo, and Aristides Gionis. 2010. An optimization framework for query recommendation. In *WSDM*.
[2] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. 2007. DBpedia: A Nucleus for a Web of Open Data. In *ISWC*.
[3] Ricardo A. Baeza-Yates, Carlos A. Hurtado, and Marcelo Mendoza. 2004. Query Recommendation Using Query Logs in Search Engines. In *EDBT Workshops*.
[4] H. Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. 2006. IO-Top-k: Index-access Optimized Top-k Query Processing. In *VLDB*.
[5] Kurt D. Bollacker, Robert P. Cook, and Patrick Tufts. 2007. Freebase: A Shared Database of Structured General Human Knowledge. In *AAAI*.
[6] Adriane Chapman and H. V. Jagadish. 2009. Why not?. In *SIGMOD*.
[7] H.A. David and H.N. Nagaraja. 2004. *Order Statistics*. Wiley.
[8] Christos Doulkeridis, Akrivi Vlachou, Kjetil Nørvåg, Yannis Kotidis, and Neoklis Polyzotis. 2012. Processing of Rank Joins in Highly Distributed Systems. In *ICDE*.
[9] Shady Elbassuoni, Maya Ramanath, Ralf Schenkel, Marcin Sydow, and Gerhard Weikum. 2009. Language-model-based ranking for queries on RDF-graphs. In *CIKM*.
[10] Shady Elbassuoni, Maya Ramanath, and Gerhard Weikum. 2011. Query Relaxation for Entity-Relationship Search. In *ESWC*.
[11] Ronald Fagin, Amnon Lotem, and Moni Naor. 2003. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.* 66, 4 (2003).
[12] Azam Feyznia, Mohsen Kahani, and Fattane Zarrinkalam. 2014. COLINA: A Method for Ranking SPARQL Query Results through Content and Link Analysis. In *ISWC*.
[13] Jonathan Finger and Neoklis Polyzotis. 2009. Robust and efficient algorithms for rank join evaluation. In *SIGMOD*.
[14] Géraud Fokou, Stéphane Jean, Allel Hadjali, and Mickaël Baron. 2015. Cooperative Techniques for SPARQL Query Relaxation in RDF Databases. In *ESWC*.
[15] Riccardo Frosini, Andrea Calì, Alexandra Poulovassilis, and Peter T. Wood. 2017. Flexible query processing for SPARQL. *Semantic Web* 8, 4 (2017).
[16] Vagelis Hristidis, Yuheng Hu, and Panagiotis G. Ipeirotis. 2010. Ranked queries over sources with Boolean query interfaces without ranking support. In *ICDE*.
[17] Hai Huang and Chengfei Liu. 2010. Query Relaxation for Star Queries on RDF. In *WISE*.
[18] Hai Huang, Chengfei Liu, and Xiaofang Zhou. 2012. Approximating query answering on RDF databases. *WWW* 15, 1 (2012).
[19] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. 2003. Supporting Top-k Join Queries in Relational Databases. In *VLDB*.
[20] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. 2004. Supporting top-k join queries in relational databases. *VLDB J.* (2004).
[21] Ihab F. Ilyas, Rahul Shah, Walid G. Aref, Jeffrey Scott Vitter, and Ahmed K. Elmagarmid. 2004. Rank-aware Query Optimization. In *SIGMOD*.
[22] Abhijith Kashyap, Vagelis Hristidis, and Michalis Petropoulos. 2010. FACeTOR: cost-driven exploration of faceted query results. In *CIKM*.
[23] Gjergji Kasneci, Fabian M. Suchanek, Georgiana Ifrim, Maya Ramanath, and Gerhard Weikum. 2008. NAGA: Searching and Ranking Knowledge. In *ICDE*.
[24] Nick Koudas, Chen Li, Anthony K. H. Tung, and Rares Vernica. 2006. Relaxing Join and Selection Queries. In *VLDB*.
[25] Jyoti Leeka, Srikanta Bedathur, Debajyoti Bera, and Medha Atre. 2016. *Quark-X*: An Efficient Top-K Processing Framework for RDF Quad Stores. In *CIKM*.
[26] Chengkai Li, Ning Yan, Senjuti Basu Roy, Lekhendro Lisham, and Gautam Das. 2010. Facetedpedia: dynamic generation of query-dependent faceted interfaces for wikipedia. In *WWW*.
[27] Sara Magliacane, Alessandro Bozzon, and Emanuele Della Valle. 2012. Efficient Execution of Top-K SPARQL Queries. In *ISWC*.
[28] Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. 2013. A Probabilistic Optimization Framework for the Empty-Answer Problem. *PVLDB* 6, 14 (2013).
[29] Alexandra Poulovassilis. 2018. Applications of Flexible Querying to Graph Data. In *Graph Data Management, Fundamental Issues and Recent Developments*.
[30] Alexandra Poulovassilis, Petra Selmer, and Peter T. Wood. 2016. Approximation and relaxation of semantic web path queries. *J. Web Sem.* 40 (2016).
[31] Alexandra Poulovassilis and Peter T. Wood. 2010. Combining Approximation and Relaxation in Semantic Web Path Queries. In *ISWC*.
[32] Senjuti Basu Roy, Haidong Wang, Gautam Das, Ullas Nambiar, and Mukesh K. Mohania. 2008. Minimum-effort driven dynamic faceted search in structured databases. In *CIKM*.
[33] Karl Schnaitter and Neoklis Polyzotis. 2010. Optimal algorithms for evaluating rank joins in database systems. *ACM Trans. Database Syst.* (2010).
[34] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: a core of semantic knowledge. In *WWW*.
[35] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. 2005. Efficient and self-tuning incremental query expansion for top-k query processing. In *SIGIR*.
[36] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. 2005. An Efficient and Versatile Query Engine for TopX Search. In *VLDB*.
[37] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. 2004. Top-k Query Evaluation with Probabilistic Guarantees. In *VLDB*.
[38] Quoc Trung Tran and Chee-Yong Chan. 2010. How to ConQueR why-not questions. In *SIGMOD*.
[39] Elena Vasilyeva, Thomas Heinze, Maik Thiele, and Wolfgang Lehner. 2016. DebEAQ - debugging empty-answer queries on large data graphs. In *ICDE*.
[40] Elena Vasilyeva, Maik Thiele, Christof Bornhövd, and Wolfgang Lehner. 2014. Top-k Differential Queries in Graph Databases. In *ADBIS*.
[41] Dong Wang, Lei Zou, and Dongyan Zhao. 2015. Top-k queries on RDF graphs. *Inf. Sci.* 316 (2015).
[42] Mohamed Yahya, Denilson Barbosa, Klaus Berberich, Qiuyue Wang, and Gerhard Weikum. 2016. Relationship Queries on Extended Knowledge Graphs. In *WSDM*.
[43] Shengqi Yang, Fangqiu Han, Yinghui Wu, and Xifeng Yan. 2016. Fast top-k search in knowledge graphs. In *ICDE*.
[44] Hailing Yu, Hua-Gang Li, Ping Wu, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient Processing of Distributed Top-k Queries. In *DEXA*.