Quark-X: An Efficient Top-K Processing Framework for RDF Quad Stores

Jyoti Leeka
IIIT Delhi, India
jyotil@iiitd.ac.in

Srikanta Bedathur
IBM Research, India
sbedathur@in.ibm.com

Debajyoti Bera
IIIT Delhi, India
dbera@iiitd.ac.in

Medha Atre
IIT Kanpur, India
atrem@cse.iitk.ac.in

ABSTRACT

There is a growing trend towards enriching the RDF content from its classical Subject-Predicate-Object triple form to an annotated representation which can model richer relationships such as including fact provenance, fact confidence, higher-order relationships and so on. One of the recommended ways to achieve this is to use *reification* and represent it as N-Quads –or simply *quads*– where an additional identifier is associated with the entire RDF statement which can then be used to add further annotations. A typical use of such annotations is to have quantifiable confidence values to be attached to facts. In such settings, it is important to support efficient top-k queries, typically over user-defined ranking functions containing sentence-level confidence values in addition to other quantifiable values in the database. In this paper, we present *Quark-X*, an RDF-store and SPARQL processing system for reified RDF data represented in the form of quads. This paper presents the overall architecture of our system – illustrating the modifications which need to be made to a native quad store for it to process top-k queries. In Quark-X, we propose indexing and query processing techniques for making top-k querying efficient. In addition, we present the results of a comprehensive empirical evaluation of our system over Yago2S and DBpedia datasets. Our performance study shows that the proposed method achieves one to two order of magnitude speed-up over baseline solutions.

Keywords

Top-K; RDF; SPARQL

1. INTRODUCTION

The Resource Description Framework (RDF) has become a common way to represent semantically linked data on the web, and in knowledge-bases such as Yago, DBpedia, FreeBase etc. As these diverse semantic sources are used in an integrated manner, there is a move towards richer representations of semantic resources from classical Subject-Predicate-Object (SPO) triple forms. One of the recommended ways to model such higher-order relationships within RDF is to use the notion of *reification* [20] which allows SPO-like statements to be made about other statements using the built-in vocabulary of RDF. Reification helps to represent various annotations such as confidence value associated with a statement. For instance, Yago2 knowledge-base makes extensive use of reification to annotate its extracted facts with their confidence, geo-location, and time [11]. Note that these reified statements can be stored either as triples in traditional triple-stores, stored using the recently proposed approach *singleton property* approach [25] or stored as *quads* – the approach we pursue in this paper.

We have focused on efficiently evaluating top-k queries over such reified RDF data. We were motivated by the following two observations: first, a large fraction of semantic resources such as Yago and DBpedia contain facts linking to quantifiable values which naturally suggests the use of queries containing ORDER-BY. This is further amplified by the use of reification to attach annotations such as *confidence* and *time*. Secondly, despite the SPARQL recommendation of using ORDER-BY / LIMIT operators, efficient processing of top-k queries with a user-defined ranking function within RDF/SPARQL setting have received limited attention [19, 33].

Take, for instance, the following query from the Yago2S dataset: *Find the top ten leaders of countries with the highest rate of inflation and the lowest rate of economic growth, and yet these leaders own luxurious items e.g., jewels, private homes, sports clubs etc.*. Since information about such possessions may also be rumours, there is confidence value associated with the ownership statement which is one of the factors in the ranking function. Most RDF processing systems handle such queries by first collecting the results and then sorting them in-memory based on the user-specified function; this approach is not very scalable. On the other hand, commonly used rank-join approaches also can not be effectively applied due to the extensive combination of joins based on ranking attributes as well as other non-quantifiable predicates in a SPARQL query.

A straightforward way of storing RDF data is using the relational model which enables the use of top-k algorithms designed for relational databases used for RDF data. The property table technique [30, 18, 36] and vertically partitioned approach [1, 29] for storing RDF data are two such techniques which can draw advantage from top-k algorithms proposed for relational databases. However, many researchers have shown that these models of storing RDF in relational databases are not efficient in handling complex query patterns seen in SPARQL queries [35, 23, 24]. Simple triple-store model of storing RDF in relational tables is also not effective for top-k querying since: (a) self-joins incurred by top-k algorithms over a large table are bound to be expensive, (b) either of the two access methods viz., the sorted or the random access [14] are not suitable because of unsorted nature of quantifiable values in RDF

CIKM'16 , October 24-28, 2016, Indianapolis, IN, USA

DOI: http://dx.doi.org/10.1145/2983323.2983727

and the complex pattern matching model of SPARQL queries (cf. Section 4.2).

Quark-X overcomes these limitations by introducing a combination of adaptively switching block-wise sorted and random accesses based on their cost estimates along with its semantic encoding of identifiers to improve locality of reference of subjects associated with quantifiable predicates. To the best of our knowledge, these features are not explored until now in relational top-k processing systems as well as RDF quad stores.

Quark-X pursues the idea of exhaustive indexing (popularized by RDF-3X [24]), and adapts it for the storage of reified RDF stored as quads. Starting with this, the Quark-X system adds following key features:

1. The use of compact in-memory synopsis indexes –called *S-indexes*– in addition to on-disk fine-grained indexes –called *Q-indexes*– for quantifiable predicates involved in user-defined ranking functions. This is in the same spirit as building impact-layered indexes in information retrieval, but carefully redesigned for use for ranking in reified RDF.
2. Intelligent reassignment of identifiers to RDF resources so that entities associated with similar predicates and reification structures are collocated on disk.
3. We propose novel Rank-Hash Join (RHJ) algorithm designed to utilize the synopsis indexes, by selectively performing range scans for quantifiable facts early on – this is crucial to the overall performance of SPARQL queries which involve a large number of joins.
4. Processing of data in blocks whenever possible, which enables simultaneous processing of multiple buckets of S-indexes to quickly generate the top-k results or reach early-termination criterion.

We evaluate Quark-X by comparing it with two state-of-the-art commercial RDF management systems – Jena-TDB-2.13.0 [16], and Virtuoso 7.2 [32](a highly optimized RDBMS for storing RDF) as well as two academic RDF systems – SPARQL-RANK [19] and RDF-3X [24]. We also develop a query workload, which represents various usage patterns of SPARQL with ORDER-BY/LIMIT over reified RDF datasets. Our performance results demonstrate that Quark-X is significantly faster than comparable systems in both, cold as well as warm cache settings, while needing a very small memory-footprint during query processing.

The rest of the paper is organized as follows. Section 2 introduces preliminaries. Section 3 includes related work done in the field of RDF stores, Databases, and Information Retrieval. Quark-X is explained in detail in Section 4 and 5 – these sections explain Quark-X's indexing and query processing subsystems respectively. Update management is explained in section 6. Section 7 explains the implementation details of Quark-X. Section 8 explains our evaluation framework. Section 9 includes our experimental evaluation, and Section 10 concludes the paper.

2. PRELIMINARIES

RDF expresses data in the form of *subject(s), predicate(p), object(o)* triples. Subject and Object represent any two connected resources on the web. The connection between them is represented through a property(a.k.a. predicate). SPARQL [10] is the standard query language for querying RDF. The SPARQL queries we consider have the following format:

```
SELECT [projection clause]
WHERE [graph pattern]
ORDER BY [ranking function]
LIMIT [top-K results]
```

The `SELECT` clause includes a set of variables that should be instantiated from the RDF knowledge base (variables in a SPARQL query are denoted by a "?" prefix. Please see Figure 2). A graph pattern in the `WHERE` clause consists of *triple patterns* in the following forms:

1. *s p o*
2. *r* **rdf:subject** *s*. *r* **rdf:predicate** *p*. *r* **rdf:object** *o*,

here, *r* stands for *fact id* (or reification id). *s, p, o* and *r* can be either bound to constants, or unbound variables in the query. The predicates starting with prefix rdf are part of the RDF reification vocabulary to explicitly declare the various parts of a RDF statement identified through its identifier *r*. The `ORDER BY` clause in the query allows a user-defined ranking function to establish the order of bindings of the projected variables (the `SELECT` clause).

Although SPARQL 1.1 standard enables a large array of possible ranking functions to be used here, in this work we limit ourselves to *convex monotonic* functions involving quantifiable (i.e., numerical) values. Examples of convex monotonic functions are: $(a + b)$,$((a * b)$ for $a, b > 0)$, $(a/b$ for $b \leq a$ & $a > 0)$. The `LIMIT` clause helps control the number of results returned.

2.1 Running Example

For ease of exposition, we use reified RDF listing shown in Figure 1(in a format popularized by YAGO2S) as a running example throughout this paper. For illustration, we have used synthetic numbers for estimated affected population. The top-k SPARQL query over the running example snippet that we use is given in Figure 2. This query *finds top-2 food products which contain toxins, ranked based on number of victims, product's cost and toxin's concentration in the product. The toxicity of a chemical compound has associated confidence value which needs to be included in the ranking function.*

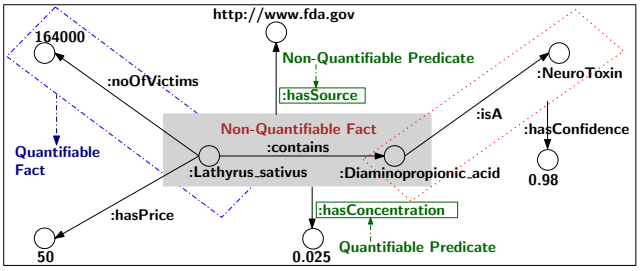
For the rest of the paper, we adopt the following terminology: We call the query patterns containing quantifiable predicates as *quantifiable query patterns*, and the remaining query patterns are called *non-quantifiable query patterns* (abbreviated as *NQP*). Subjects of quantifiable query patterns are called *quantifiable variables*. The example query in Figure 2 has (?product <hasPrice> ?price) as a quantifiable query pattern. Also, the facts containing quantifiable values in our knowledge base are called *quantifiable facts*.

3. RELATED WORK

In this section, we briefly discuss and contrast our work with the related work from RDF and relational databases as well as from information retrieval. We will limit ourselves to the discussion on top-k query processing, and direct the interested reader to W3C recommendations on supporting annotations and reification in RDF (cf. Section 4.3 in [27]), and their usage in real-world semantic datasets such as Yago [11].

RDF/SPARQL: Although modern RDF systems such as Virtuoso and Jena have fairly sophisticated SPARQL query processing, their approach to top-k queries is to collect all the results of a query, sort them or use an in-memory priority queue to compute top-k answers. This approach is expensive as the query engine needs to process all solutions, even though only k of them are requested by the user.

The other approach involves *early termination* in an explicit manner. We are aware of only a few such approaches in the context of SPARQL – albeit only over triple-stores – which we discuss next. The SPARQL-RANK framework proposed by Magliacane et al. [19] makes use of different index permutations used in native triple-stores for fast random access during top-k processing, and applies early-termination criterion. They propose an algorithm, which requires the left-most index used in the join plan to be sorted based on the ranking function, and then it randomly probes the right-side




```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns\#>
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
#@ <id1>
:Lathyrus_sativus :contains :Diaminopropionic_acid.
#@ <id2>
:Lathyrus_sativus :noOfVictims "164000"^^xsd:int.
#@ <id3>
:Lathyrus_sativus :hasPrice "50.0"^^xsd:double.
#@ <id4>
<id1> :hasSource <http://www.fda.gov>.
#@ <id5>
:Diaminopropionic_acid :isA :NeuroToxin.
#@ <id6>
<id5> :hasConfidence "0.98"^^xsd:double.
#@ <id7>
<id1> :hasConcentration "0.025"^^xsd:double .
```

Figure 1: Example RDF knowledge graph

```
SELECT ?product ?chemical ?source
  ((f1(?countVictims) + f2(?price)) * f3(?conc) *
       f4(?conf) as ?rank)
WHERE {
  ?product :noOfVictims ?countVictims.
  ?product :hasPrice ?price.
  ?reif rdf:subject ?product; rdf:predicate :
      contains; rdf:object ?chemical.
  ?reif :hasConcentration ?conc.
  ?reif :hasSource <http://www.fda.gov>;
  ?reif1 rdf:subject ?chemical; rdf:predicate :
      isA; rdf:object ?toxin.
  ?reif1 :hasConfidence ?conf.
} ORDER BY DESC(?rank) LIMIT 2
```

Figure 2: Running Example Query

index. Thus, when the right-side index is large, the performance of rank join suffers. Also, the requirement for the left-side index to be sorted based on ranking function makes it unsuitable for arbitrary user-defined ranking functions.

In another framework introduced by Wang et al. [34], quantitative entities in the RDF dataset are separated out into an *MS-tree* index. In the first step of query processing, candidate entities are located using the MS-tree index that are then used as *seeds* for performing breadth-first (BFS) traversals over the graph to find matching subgraphs. If the query requires only a few highly correlated predicates, the algorithm may end up storing many unnecessary nodes in the queue, making the retrieval of the first entity possible only after several iterations. On the other hand, our approach does not require unrelated predicates and entities to be stored together. However, we did not empirically compare our work with this work as it is still unclear how to apply their BFS based candidate generation phase on reified databases.

Relational Databases: Hash Rank-Join (HRJN) [12] and Nested Loops Rank Join (NRJN) [13] represent the state-of-the-art relational rank-join algorithms. HRJN [12] is based on ripple join algorithm [9]. 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 in-turn fed to a priority queue, which outputs them in the order specified by the ranking function. NRJN is similar to HRJN except that unlike HRJN it does not store input tuples, but rather follows a nested-loop strategy. However for RDF data, SPARQL-RANK showed experimentally that it outperformed HRJN [19]. The performance gain was attributed to the unsorted nature of numerical attributes present in indexes build by RDF engines. We show in a later section that we outperform SPARQL-RANK by a large margin by targeting precisely the numerical attributes once again. Hence, we did not compare Quark-X with algorithms like HRJN for relational databases.

Information Retrieval (IR): *Block-max* index structure proposed by Ding et al. [7] is one of the effective approaches proposed in the IR community for retrieving top-k documents efficiently. The block-max index stores documents sorted by document ids in a block-partitioned inverted index. In contrast, the identifiers in our approach are sorted by scores. Our approach is somewhat similar to impact-layered indexes, where the posting list is divided into layers such that the higher layer posting list has a lower score than the layer below it. Additionally, top-k ranking algorithms proposed in the IR community are different from those proposed in the RDF and relational databases community – in IR, bindings of just one variable needs to be retrieved, whereas in relational database as well as RDF setting, bindings of multiple variables need to be retrieved, leading to unsorted orders.

4. QUARK-X INDEXING

Quark-X pursues the exhaustive indexing approach for RDF databases made popular by RDF-3X [23], and additionally develops an indexing framework for efficiently answering top-k queries. This section presents the details of the indexing framework in Quark-X for top-k processing. It consists of:

1. two indexes, an in-memory synopsis index called S-index and a bucket-ordered quantifiable Q-index, on *quantifiable facts* in the database,
2. a semantic re-encoding strategy of identifiers which utilizes the information gathered during indexing to remap the ids involved in quantifiable facts.

The index creation process, and the semantic re-encoding of identifiers is illustrated in Figure 3.

Similar to most state-of-the-art RDF engines, Quark-X encodes typically long URIs and strings in RDF as fixed-length numerical identifiers and maintains these mappings in a dictionary structure. The size of the dictionary is further reduced by identifying frequently occurring common maximal prefixes among URIs in the database which are encoded first into integers. These prefixes are represented using their encodings in URIs they occur in, and these prefix encoded strings are stored in the dictionary. For example, if the URI prefix http://yago-knowledge.org/resource is mapped to integer 1, then the URI http://yago-knowledge.org/resource/contains is prefix-encoded to 1/contains and then stored in the mapping dictionary.

4.1 Quantifiable Indexes

Quark-X introduces two special indexes for quantifiable facts in the database, designed to help in efficient early pruning of results and in preserving interesting orders.

S-index. The S-index is an in-memory synopsis index which stores for each quantifiable predicate a statistical metadata summarizing all the entities and their associated quantifiable values in a compact form. In particular, histogram-based information is maintained for each quantifiable predicate to describe the value distribution. For simplicity, we employ an equi-depth histogram over

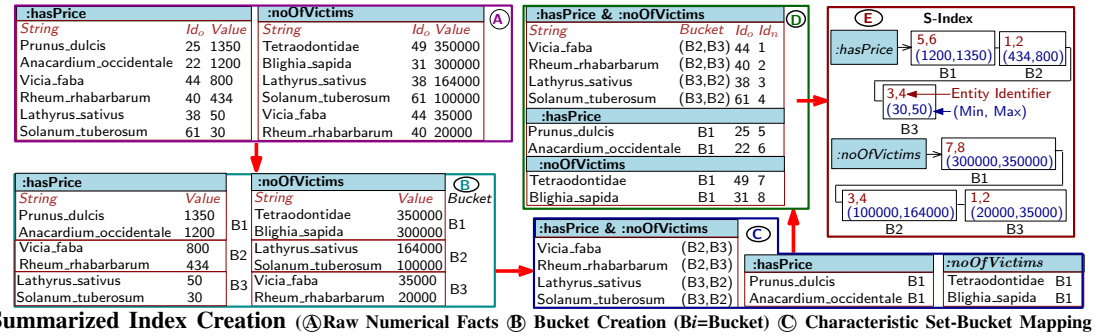


Figure 3: Summarized Index Creation (Ⓐ Raw Numerical Facts Ⓑ Bucket Creation (B*i*=Bucket) Ⓒ Characteristic Set-Bucket Mapping Ⓓ Assign new collocated ids (Id_o: old identifier, Id_n: new identifier) Ⓔ S-Index) [1]

the range of values for each quantifiable predicate. We can easily employ other forms of histograms as required. Associated with each bucket of the histogram, we maintain:

- the lower and upper values, $min[b]$ and $max[b]$, defining the range of quantifiable values covered by the bucket b,
- the set of *subject ids* associated with the quantifiable value falling within this bucket range.

Although it is possible to store the set of subject ids as a Bloom filter, we chose not to do so as it can not be used to maintain *sorted orders* within each bucket, which, as we show later, can be used to further speed up top-k processing.

We illustrate the process of constructing the S-index in Figure 3 for our running example data from Figure 1. The S-index constructed at this stage is called a *temporary S-index*. In this example, the predicate *:hasPrice* has values in the range of [30, 1350], and entities associated with the predicate *:hasPrice* have been divided into 3 buckets. Similarly, the predicate *:noOfVictims* too has 3 buckets. For brevity, the buckets for the other quantifiable predicates, viz., *:hasConfidence* and *:hasConcentration*, are not shown. The semantic encoding strategy we explain next necessitates re-mapping of subject ids in the S-index as shown in step E.

Q-Index. Since S-index buckets only store upper- and lower-bounds of scores in the bucket, we maintain an additional B+-tree index called *Q-index* on *predicate id*, *bucket number*, *subject id* and the corresponding *quantifiable value*. This index is used for finding the exact numerical values associated with a subject and a (quantifiable) predicate. Once the candidate buckets are determined using S-index, it is quite straightforward to use Q-index to retrieve quantifiable values in the order of their subject ids.

4.2 Semantic Encoding of Identifiers

In traditional RDF engines the URIs and strings in the database are encoded typically in their order of appearance in the database or hashing. Both these techniques have been shown to be suboptimal for compressibility and, more importantly, for efficient join processing [31]. In top-k ranking join queries, the problem is further exacerbated by the requirement that identifier assignment should not only help the classical equi-join processing, but also preserve the ordering over quantifiable values.

To address this, during *S-index* construction the RDF terms are re-encoded and re-mapped in the dictionary as well as the S-index buckets. These new encodings are derived through the *soft-schema* present in the RDF data which stems from the fact that multiple RDF statements are used to describe the "properties" of a subject. To illustrate this, consider the following query patterns from query given in Figure 2 – (?product :noOfVictims ?countVictims), (?product :hasPrice ?price) – both the patterns describe a food product, its price and the number of victims through the predicates *:noOfVictims* and *:hasPrice*. These two predicates are also strongly correlated in the database since this pairing of predicates holds for only food products. Many entities can likewise be uniquely identified by the predicates connected to them. This observation has been used earlier for improving the cardinality estimates of RDF queries [22], where they name the set of predicates connected to an entity as its *characteristic set*.

The *semantic encoding scheme* employed by Quark-X can be understood by using a subset of the example knowledge base shown in Figure 1. In that knowledge base, subjects like *Rheum_rhabarbarum*, *Vicia_faba*, etc. are described by two quantifiable predicates – *:noOfVictims* and *:hasPrice*. On the other hand, subjects like *Prunus_dulcis*, *Blighia_sapida* etc. have information pertaining to just one quantifiable predicate, either *:noOfVictims* or *:hasPrice*. Resulting in the following 3 characteristic sets: *(:noOfVictims and :hasPrice)*, *(:noOfVictims)* and *(:hasPrice)*.

Using the temporary S-index, the subjects that have quantifiable predicates belonging to a characteristic set are assigned their corresponding buckets (illustrated in steps A through C in Figure 3). While doing so, subjects falling within each characteristic set are ordered according to their predicate values, instead of subject values. Next, ids are assigned to the sorted subjects in a manner so that subjects belonging to the same characteristic set and bucket are allocated consecutive ids. This is shown in Figure 3, step D. Then the ids present in the *temporary S-index* are re-mapped using the mappings generated at the end of step D. The resulting semantically encoded S-index is shown in the step E of Figure 3.

5. QUARK-X QUERY PROCESSING

Now, we turn our attention to the top-k query processing in Quark-X and describe how our quantifiable indexes are utilized to evaluate queries efficiently. Since S-index is an in-memory synopsis index which summarizes the quantifiable value distribution, we aim to use it greedily for join-ahead pruning early on in the plan. The query compiler simply extracts the quantifiable predicates from the query before generating a cost-based query plan on non-quantitative patterns. The S-index based join-ahead pruning over quantifiable query patterns is added subsequently so as to generate candidate ids while maintaining interesting orders.

Note that S-index buckets are processed bucket at a time in the order of quantifiable value, and the same sequence is retained in subsequent (non-quantifiable) joins as well. We call this as *S-index filtering of non-quantitative index*, and it plays an important role in the query evaluation pipeline we present next.

[1] Example data manually extracted from the following two books:
1. K.R. Natarajan. India's poison peas. Chemistry, 49(6), 1976. (obtained from the FDA Poisonous Plant Database).
2. David R. Briggs. Naturally-occurring toxicants in some nutritionally significant plant foods and fish.

The query processing in Quark-X proceeds in three stages: first, the *S-index join* is performed using early-termination over S-index synopsis to generate candidate ids with lower and upper bounds on the associated quantifiable values. Next, these candidate ids are used for join-ahead pruning over non-quantifiable query patterns in *NQP-join* or non-quantifiable query pattern joins, and, finally, the *SQ-index join* is performed where the final list of top-k results with their complete order-by scores is generated. We describe each of these stages next.

The entire query processing flow is illustrated for our running example in Figure 4. In the rest of this section, we regularly reference various named parts of this figure for the ease of exposition.

5.1 S-index Join

Given a monotonic scoring function, we use the quantifiable value distribution maintained in S-index for early-termination using neighborhood expansion [37]. Note that this in itself is not an entirely novel technique – similar ideas have been explored in distributed top-k processing [21], scan depth estimation [26], and others for enabling early termination [14]. We use S-index in a more effective way by combining the quantifiable value sorted candidate ids generated by the index with non-quantifiable joins. Recall that Quark-X employs semantic encoding of identifiers (cf. Section 4.2), which ensures that identifiers belonging to a characteristic set to be clustered. This works synergistically with S-index to provide increased sequential scans on other indexes.

In our running example from Section 2.1, the S-index join on quantifiable predicates :hasPrice and :noOfVictims, retrieves the following bucket pairs: $L1' = (B2, B3)$ with join results $(1, 2)$ and max-score bound 28×10^6 and $L2' = (B3, B2)$ with join result $(3, 4)$ with max-score bound 8.2×10^6. The left-bottom block of Figure 4 with label E_1' depicts the S-index join result $L2'$. Observe that the identifiers in $L2'$ – *viz.*, $(3, 4)$ – are also present in the PSOR index (PSOR stands for an index sorted lexicographically in the order of **P**redicate, **S**ubject, **O**bject, **R**eification ids) for the predicate :contains which is a non-quantifiable part of the query, as illustrated in the grey block E_2' of Figure 4. Such a collocation significantly speeds up the query processing by preferring range-scans over random probes.

5.2 Non-quantifiable Predicate Joins

We now turn our attention to the join processing between non-quantifiable query predicates using the S-index join results for join-ahead pruning. Note that S-index induced score computation takes place on *quantifiable variables* – i.e., subjects with quantifiable predicates in the query. Therefore, by treating ordering over quantifiable variables as an *interesting order* [12], we can generate rank-aware plans which can exploit the S-index results.

Generation of such plans is quite different from the classical top-k ranking systems which focus on maintaining interesting orders inferred from join conditions, group-by and order-by clauses, while ignoring enforcing interesting orders on other attributes like quantifiable variables. Thus they end up materializing all results of a join before generating the final ranked list. To remedy this, we propose a novel rank-join operator called *Rank-Hash Join (RHJ)* which we describe next.

5.2.1 Rank-Hash Join (RHJ) algorithm

The key idea of our Rank-Hash Join is to use the results from the S-index join step to adaptively decide if the right-side index of the join has to be *probed* or *to be scanned* fully. It does so in a manner similar to a classical hash-join, but differing in not requiring the entire result of the left hand side of the join to be in hash table. Instead, RHJ needs to maintain only the ids from a single bucket in the S-index in the hash table. Since we process all elements in the hash-table completely before pulling the next bucket from the underlying S-index join, it is guaranteed not to miss any results.

The RHJ algorithm builds hash-table on the *left-side* using tuples filtered through from the S-index join stage. For retrieving tuples from *right-side* of the join, the following two index choices exist: (1) index on the sorted order of joining variables, which we term as **index-1**, that can enable efficient disk skips by using the ids retrieved from left hand side of the join in a *sideways information passing* optimization [24], and (2) index on the sorted order of quantifiable variables, which we term as **index-2**, over which we can limit the number of pages required to be scanned by utilizing the entity ids from the underlying S-index. These indices are adaptively selected, triggered by a condition based on the cost calculated using a cost model on these index alternatives which estimates the number of disk pages required to be scanned.

The cost model for index-1 Due to its sorted order, the number of pages required to be fetched from this index in the worst case is equal to the number of tuples retrieved from left sub-plan. Hence, the number of tuples satisfying the left sub-plan is taken to be the estimate of the cost of this candidate plan. Specifically,

$$C_1 = N \times \prod_{i \in qp} S_i \times S_{nqp},$$

where, S_{nqp} is the selectivity of the non-quantifiable predicate in the left sub-plan; S_i denotes the selectivity of i-th quantifiable predicate in left sub-plan; N is the number of elements in a bucket (note that we use equi-depth buckets), and qp are the quantitative predicates in the query.

The cost model for index-2 As a result of semantic encoding, the ids within a bucket are stored consecutively on a disk page (or in adjacent disk pages) which can be retrieved with only one seek. Therefore, in the worst case the number of pages required to be fetched from this index is equal to number of buckets whose score is above l_k where l_k is the score of the k-th best scoring element seen so far, i.e.,

$$C_2 = \text{ number of buckets having score greater than } l_k.$$

After estimating the cost of each candidate plan, the query re-optimizer chooses the plan with smallest cost amongst the two choices.

Note that, due to *S-index filtering of non-quantitative index*, our algorithm incurs zero-cost for switching indexes (plans) at *"materialization" points* [6], i.e. decision points where plans are changed. In RHJ, materialization points are points at which next bucket is retrieved from leftmost index. For example in our running example, points at which buckets L1 (left-middle with label B), L2 (middle with label E), L3 (right-middle with label H) are retrieved from the leftmost index are materialization points. We believe, there is only one prior work by Ilyas et al. [15] in DBMSs, which also uses adaptive query processing(AQP) for efficient top-k retrieval. However, unlike our proposal, the state-saving techniques proposed in [15] wastes a significant amount of already done work while switching plans (with state-of-the-art rank join algorithms like HRJN, NRJN, etc).

Our novel RHJ algorithm borrows ideas from classical hash join and adaptive query processing (AQP) research and applies it in the context of RDF/SPARQL processing, to the best of our knowledge, for the first time.

RHJ Stepwise Description: Now we go through the workings of the RHJ algorithm illustrating it step-by-step based on our running example from Section 2.1. We refer to the steps illustrated in the

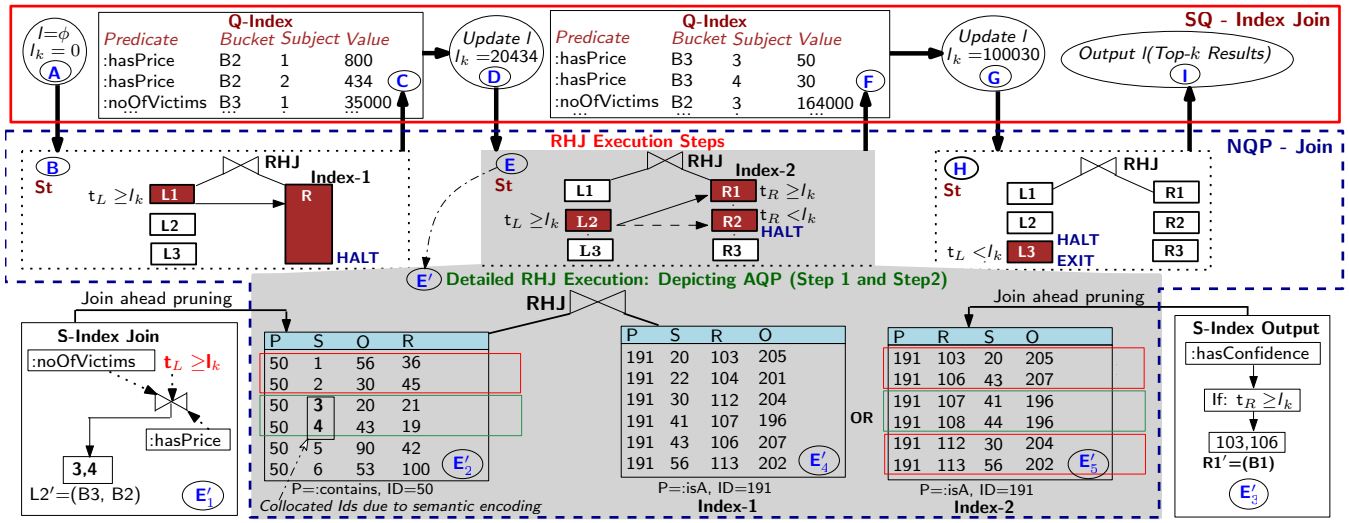


Figure 4: Quark-X Query Processing (AQP: Adaptive Query Processing; Reading order from label A to I)

schematic diagram in Figure 4. In the rest of this section, we use l to denote the list with top-k results, with l_k denoting the score of the k-th result. For ease of exposition, our description is ordered using the labels used in Figure 4 starting from label A at the top-left of the figure until label I at the top-right of the figure.

A: Initialize the algorithm with $l = \emptyset$ and $l_k = 0$.

B: Left and right-side are joined using *classical hash join* where:
- *Left side:* S-index join pulls the first bucket $L1' = (B2, B3)$, which is used as a filter over P=:contains index, which helps create candidate result block $L1$.
- *Right side:* Uses index-1 as illustrated in the grey portion of the figure with label E'_4. The **index-1** (PSRO ordered index) is preferred here over the **index-2** (PRSO-ordered) alternative based on the cost considerations described above. In particular, we observe that the ids 56 and 30 in column O, are present only in bucket $R3$ of **index-2** which necessitates retrieval of all pages from $R1$ and $R2$ from disk, making it extremely inefficient. In comparison, **index-1** is used, it can utilize the sideways information passing optimization since this index has the joining variable in sorted order. This results in only 2 disk seeks in the worst-case (assuming tuples corresponding to 56 and 30 are stored on separate disk pages).

C: Find quantifiable values from *Q-index* using candidate buckets retrieved from *S-index* in B.

D: The retrieved quantifiable values are aggregated using ranking function in top-middle block with label D. The resulting k tuples with highest score are then stored in l. The k^{th} maximum scoring element l_k is passed to NQP-join stage – it helps in performing early-termination check by verifying that, *there is no combination of buckets that has a maximum score (for descending) more than the k-th result*.

E: Left and right are again joined using *classical hash join* in middle block with label E.
- Left Side: *S-index Join* (left-bottom block with label E'_1) pulls next bucket $L2' = (B3, B2)$ (shown in left-bottom with label E'_1), which is in-turn used as filter over P=*:contains* index (grey block with label E'_2), which helps create block $L2$
- Right-side: Uses index-2 as illustrated in the grey block with label E'_5. Note, here index-scan is speed-up by pulling buckets from S-index (shown in right-bottom block with label E'_3).

We explain next the reason for choosing **index-2** (PRSO). From grey block with label E'_2, we observe block $L2$ requires 2 disk seeks for the two identifiers 20 and 43 when **index-1** (PSRO) – shown with label E'_4 in grey block – is used. However, using l_k (score of k^{th} result), we observe that only bucket $R1$ (shown with label E'_5) is needed from the right side index. Thus, it is beneficial to only scan $R1$ (requiring 1 disk seek) of **index-2** instead of performing a full scan on the **index-1**.

F: Find quantifiable values from *Q-index* for the new set of candidate buckets retrieved from *S-index* in E.

G: The retrieved quantifiable values are again aggregated using ranking function. The resulting k tuples with highest score are then stored in l. The k^{th} maximum scoring element l_k is passed to NQP-join stage – which helps in performing early-termination check (described in D).

H: When pulling next bucket from S-index Join, we find early-termination condition is satisfied (as score of k-th element is greater than maximum score (t_L) of yet to be retrieved buckets i.e. $l_k > t_L$), hence the algorithm terminates.

I: Outputs top-k scores stored in list l.

5.2.2 SQ - Index Join

The S-Index join stage passes bindings of entity ids along with their corresponding bucket numbers to NQP-Join stage, for quantifiable variables appearing in sorted order in index scans. For finding the explicit numerical values of other quantifiable variables which appear in unsorted order in index scans, we first find their corresponding bucket numbers using S-Index.

Using the bucket number retrieved either using the approach described above or using S-Indexes, we find the exact numerical values using Q-Index. The obtained numerical values are stored in a list l, and the list is in-turn used to find the score of the last element l_k with respect to the ranking function.

6. UPDATES

Quark-X handles updates similar to the other state-of-the-art RDF management system like RDF-3X and similarly assumes that updates are mostly insertions, are far fewer compared to queries and can be batched together. During batch updates, Quark-X creates differential indexes for S-index, Q-index, and permutations of SPOR which are stored in main memory and are merged with the main index at suitable intervals. For recovering in case of failure, differential indexes are additionally stored in log files on disk.

During query processing, these indexes, and the main Quark-X indexes undergo merge-join; however, being small, these incur little overhead. While re-assigned ids in main indexes helped us in clustering together semantically similar ids on disk, thereby reducing disk seeks, the differential indexes reside entirely in main memory

and can avoid disk seek altogether. Id-assignment is therefore not done in these indexes and is deferred until they are merged with the main index at which point all ids are reassigned afresh.

7. IMPLEMENTATION

In this work, we assumed system architecture of a quad-store that provides with a mapping dictionary between ids and strings. In line with this, we have used RQ-RDF-3X [17] as a baseline framework for our implementation. **RQ-RDF-3X** is a quad store with exhaustive clustered B+-indexing of all permutations of SPOR. RQ-RDF-3X stores all the 24 permutations over quads, many of which are superflorous to begin with, thus, we dropped the following: 1) permutations where R appeared in the third position (e.g. SPRO) were removed because R is always unique, therefore, a sorted order of O for SPR does not provide additional help during joins; 2) all permutations where R appeared in the first place (except RSPO) were removed since sorted order of R, always unknown, does not give any advantage for speeding-up range scans. But when R is a joining variable, its sorted ordering helps perform joins efficiently; however, only one index (RSPO) is sufficient for this purpose.

After explaining RQ-RDF-3X, now we explain how processing data in *blocks* improved query execution time of Quark-X. At the beginning of query processing, in the S-Index join stage, the candidate set of buckets required to be retrieved is equal to the set of all plausible buckets. Accordingly, the S-Index aggregates buckets until a fraction of k entities are retrieved and these aggregated buckets are then passed to the NQP-Join stage. As the query execution proceeds, a stage comes when it is possible to restrict the candidate set of buckets based on their score and the score of the k-th result. At this stage, we divide these candidate set of buckets into fractions. It is evident that upon incremental processing, the threshold score would get tighter, which helps in early termination. We also access data in blocks during the SQ-Index join stage – where we keep materializing the results obtained from NQP-Join until retrieving k results. SQ-Index join then processes these materialized results together. After k results have been retrieved, SQ-Index join processes together aggregate buckets passed by S-Index join stage to NQP-Join. Thus block-wise access helps in amortizing cost of index scans over a range of results.

8. EVALUATION FRAMEWORK

Quark-X is implemented in C++, compiled with g++-4.8 with -O3 optimization flag. All experiments were conducted on a Dell R620 server with Intel Xeon E5-2640 processor @ 2.5GHz, 64GB main-memory, RAID-5 hard-disk with 3TB effective size. In our experimental evaluation, we report cold-cache timings after dropping filesystem caches using: `echo 3>/proc/sys/vm/drop_caches`, and warm-cache numbers by repeatedly running the query processor with the same query 5 times, and taking the average of last 3 runs.

We evaluate against two research prototypes – RDF-3X and SPARQL-RANK, and two state-of-the-art commercial systems – Jena-TDB-2.13.0 and Virtuoso 7.2. Among the research prototypical competitors we use, the inability of RDF-3X to compute only the top-k results, and the higher number of random seeks incurred by the query processing algorithm of SPARQL-RANK make them overall much slower in comparison to Quark-X. Note that the overheads due to random seeks are further exacerbated in the straight forward extension of these algorithms to quads due to the additional joins with metadata like confidence values. On the other hand, Quark-X outperforms the commercial systems which natively support quads by smartly encoding the ids using soft-schema embedded in the data to improve locality of reference in its index scans.

	Yago2S	**DBpedia-RF**
Size of input files (in ttl)	46 GB	74 GB
# of quads	473,271,482	668,867,020
# of numerical quads[2] containing only confidence	236,635,830	334,433,512
# of numerical quads without confidence	569,558	197,530

Table 1: Sizes of Datasets and Databases (RF: Reified Form)

8.1 Datasets

Despite the abundance of a number of performance benchmarks for RDF/SPARQL query processing [8, 4, 28], our evaluation could not use them since all of them are designed primarily for triple-stores, with no queries using reification and named-graphs in their query set. Therefore, we decided to work with a suite of top-k queries which we designed over two large real-world datasets which extensively use named-graphs or reification. The first dataset we use is DBpedia 3.7 [3], which contains facts extracted from Wikipedia, with provenance expressed in the form of named graph. The second dataset, Yago2S [11], contains facts extracted from Wikipedia and combined with GeoNames and WordNet. It encodes additional information –e.g. confidence score, time, spatial location, and provenance– with each fact using fact ids encoded as a comment before each triple in Turtle format.

To simulate the situation where a confidence score (a real-number between 0 and 1.0) is associated with each fact, we assigned confidence values using an exponential distribution to all facts in the original dataset, and then we translated them into quads.We observed similar trend in results with Uniform distribution, but due to lack of space, we report results only with exponential distribution in the paper. Thus, Yago2S contains a total of $237,180,265$ numerical quads – with $236,610,707$ quads containing only the confidence predicate and $569,558$ containing the remaining quantifiable predicates. Similarly for DBpedia, out of $668,867,020$ quads, we have $415,131,628$ numerical quads with $396,144,979$ containing only confidence values that we assigned. Table 1 summarizes key statistics of the dataset and the size of the resulting database in Quark-X.

8.2 Benchmark Query Workloads

Our benchmark query set consists of a set of 11 top-k ranking queries each for DBpedia and Yago2S. These queries are designed keeping in mind the following SPARQL query features that have been found to be quite important [2]: **structural** features – in which we consider (a) the number of triple patterns (**TP**), (b) the count and degree of joins, and (c) the type of joins; and **statistical** features – in which we consider their (a) result cardinality and (b) filtered result selectivity. For top-k SPARQL query workloads, in addition to the above set of features, we also use the *quantifiable triple pattern count* that has been suggested earlier [38].

Apart from these, we observed the following two features are also important in determining the complexity of query:

Shape: Based on the overall shape of the SPARQL query, we classify them as either *Star* or *Complex*. Queries that belong to *Star* class have only one non-quantifiable triple pattern, connected to other quantifiable triple patterns. On the other hand, queries which contain more than one non-quantifiable triple pattern connected to other quantifiable triple patterns belong to *Complex* class. Although this is a high-level classification, we found them to be sufficiently useful to highlight the differences in the query performance.

Non-Quantifiable Triple Patterns (Non-Quant TP): The number of non-quantifiable triple patterns in the query highlights the efficiency of a top-k for processing non-quantifiable patterns alongside quantifiable query patterns.

Table 2 summarizes some of the important features of all 11 queries for each dataset we have considered in this evaluation. As

[2]quads containing quantifiable predicates

Query id	Shape	# QuantTP	# NonQuantTP	Res.Card.
1	Star	3	1	111,425
2	Star	4	1	108,911
3	Star	3	1	298
4	Star	3	1	396
5	Complex	2	3	700
6	Star	2	1	3,902
7	Star	2	1	15,380
8	Complex	2	3	44,063
9	Complex	2	4	12,909
10	Star	2	1	42,186
11	Complex	2	2	2,639

(a) Yago2S

Query id	Shape	# QuantTP	# NonQuantTP	Res.Card.
1	Complex	3	2	7,763
2	Star	3	1	26,631
3	Complex	3	3	313
4	Star	3	1	861
5	Star	8	1	167
6	Complex	2	3	21
7	Complex	2	2	69,282
8	Complex	1	2	85,968
9	Complex	2	2	293
10	Complex	3	2	293
11	Simple	1	2	182

(b) DBpedia

Table 2: Summarized Characteristics of Benchmark Queries

shown, the queries are designed so as to provide a broad coverage of all the key features. Due to lack of space, we point to the website (https://www.iiitd.edu.in/~jyotil/quarkx-benchmark.html/BenchmarkQueries.docx) for further details of the queries used, including the query listing in SPARQL. It is worth mentioning that although our framework is aimed at convex monotonic ranking functions, for simplicity we use linear ranking functions in evaluation.

9. EXPERIMENTAL RESULTS

In this section, we present the results of our performance evaluation of Quark-X against the baseline systems we have considered. We also discuss the impact of individual components of Quark-X on the overall performance of top-k queries. Unless stated explicitly otherwise, the results correspond to the setting $k = 50$.

9.1 Loading of Data and Size of Database

Framework	DBpedia		Yago2S	
	Time	Size	Time	Size
Quark-X	13.28 hours	249 GB	7.59 hours	175 GB
Virtuoso	2.37 hours	66 GB	1.53 hours	43 GB
Jena-TDB	5 days	296 GB	3 days	132 GB
RDF-3X	14 hours	156 GB	8.75 hours	96 GB
SPARQL-RANK	18 days	326 GB	10 days	221 GB

Table 3: Data Load Performance of Various Frameworks

We start by discussing the loading time – summarized in Table 3 – of various frameworks. Among the systems compared, SPARQL-RANK and Jena-TDB took the longest time to load. SPARQL-RANK operates on an older version of Jena (ARQ 2.8.9) and it took more than 2 weeks to load DBpedia, whereas Jena-TDB-2.13.0 needed about 5 days. Both RDF-3X and Quark-X have almost the same loading time, the least among all row stores.

The size of the database created by Quark-X was smaller than that by SPARQL-RANK and comparable to the one created by Jena-TDB. However, it was larger than that of RDF-3X since Quark-X uses RQ-RDF-3X as the underlying framework which creates many more clustered indexes than RDF-3X and has to build an additional Q-index. Apart from the Q-Index, Quark-X also creates S-Indexes, but that has a comparatively smaller memory footprint. The size of S-index for the two datasets YAGO2S and DBpedia is 904 MB and 1.7 GB respectively, about 2% of the size of the raw data, despite the fact that more than 50 percent of facts in two large real-world datasets which we have used for experimentation (YAGO2S and DBpedia) are quantitative. Further, the cost of construction of Q-Index can be amortized by removing the quantitative facts stored in POS and PSO indexes of the underlying RDF store (RQ-RDF-3X in our case), as this information is already present in S and Q-Indexes.

From the results in Table 3, the performance of Virtuoso is noticeably better with respect to loading time (using buffer size of 48 GB) and size of database created. This is not very surprising because unlike RQ-RDF-3X (and RDF-3X) which takes an exhaustive indexing approach, Virtuoso builds only two default indexes (PSOG and POSG), plus 3 distinct projections (SP,OP,GS) [5]. We would like to emphasize that the ideas introduced in this paper can be applied to other RDF quad-stores. Choosing RQ-RDF-3X is merely to demonstrate the effectiveness of our approach. Note that Quark-X uses only a small amount of storage (about 6% of the size of raw data for both S- and Q-indexes together), rest of the overhead is due to the underlying engine RQ-RDF-3X.

9.2 Query Execution Performance

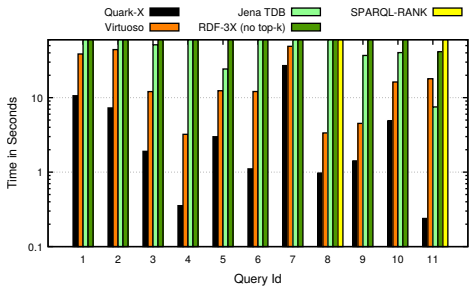
Now we turn our attention to the query processing performance in answering top-k queries, by first presenting the cold-cache performance followed by the warm-cache performance. In our discussion, we use aggregated *speedup* values computed as the geometric mean of individual query speedups for each system considered. Thus, $\text{speedup}_{(X,Y)} = \left(\prod_{i=1}^{n} \frac{Y_{Q_i}}{X_{Q_i}}\right)^{\frac{1}{n}}$, where X_{Q_i} denotes the time taken by the system X for evaluating the query Q_i and n is the total number of queries in the benchmark. Workload-average benchmarks like TPC frequently use geometric mean, since it normalizes the values being averaged against outliers.

Our primary comparison is against the columnar-store Virtuoso, which has been shown to have superior performance in comparison to other RDF storage engines [5]. It is embellished with many optimizations, of which vectorization and cache-consciousness are the most relevant to our experiments. In contrast, the current implementation of Quark-X runs in single threaded mode and does not have cache-conscious features as well as vectorized execution modes. We believe that the use of these optimizations will significantly help in further improving the performance of Quark-X.

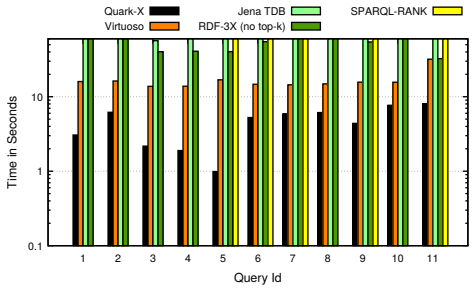
Cold-cache Performance

The performance of each system on all benchmark queries is summarized in Figure 5, which plots in logarithmic scale the average time taken, in seconds, after running the query in cold-cache 5 times. Note that we set the time-out for queries as 30 minutes, hence, the Y-axis is limited to $1,800$.

The first observation we can make from these numbers is that Quark-X outperforms all other systems under consideration by a large margin. We also observe that SPARQL-RANK is many orders of magnitude slower than even RDF-3X which does not do any top-k processing at all, and instead materializes all the results of the query. As we already discussed in Section 3, this is primarily due to the query processing algorithm of SPARQL-RANK which ends up using many random accesses over indexes. It is worth mentioning that SPARQL-RANK returned incorrect results for all queries except Q_8, Q_{11} of DBpedia, and Q_5, Q_6, Q_7, Q_9, Q_{11} of YAGO2S, hence the results for the incorrect queries have not been shown in Figure 5. For queries returning correct results, Quark-X outperformed SPARQL-RANK over both DBpedia and YAGO, with all benchmark queries timing out on SPARQL-RANK. In subsequent experiments, we will not report the results over SPARQL-RANK.



(a) DBpedia



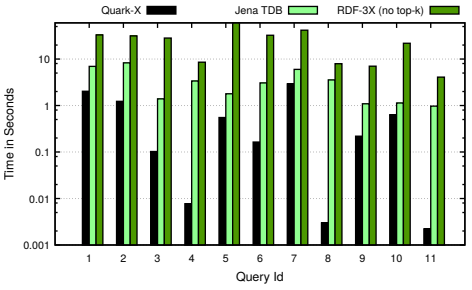
(b) Yago2S

Figure 5: Cold-cache Query Processing Performance

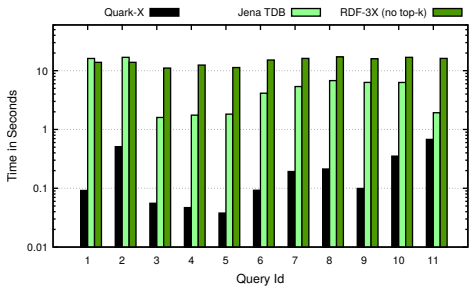
Quark-X outperforms RDF-3X for all queries by 1–2 orders of magnitude. The reason for poor performance of RDF-3X is: its inability to retrieve just the top-k results.

We can also see that Virtuoso is by far the closest in performance to Quark-X cold-cache setting. Quark-X outperforms it by a speedup factor of 3.9 over Yago2S and 7 for DBpedia. Of all the queries, Quark-X is significantly faster for Query Q_{11} over DBpedia by almost two orders of magnitude over Virtuoso, demonstrating the power of S-index. S-indexes help Quark-X skip over large portions during query evaluation.

Warm-cache Performance

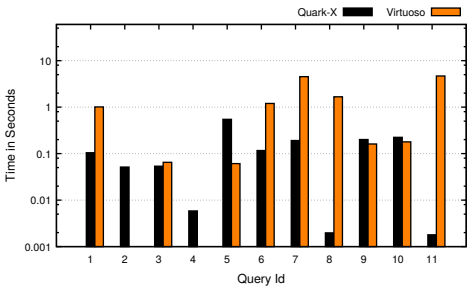


(a) DBpedia

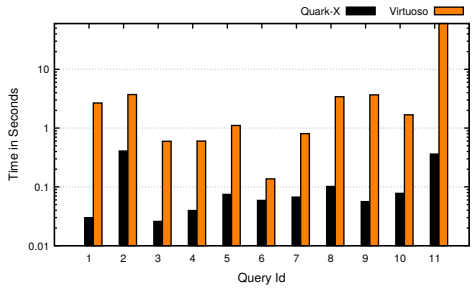


(b) Yago2S

Figure 6: Warm-cache Query Processing Performance (Mode1)



(a) DBpedia



(b) Yago2S

Figure 7: Warm-cache Query Processing Performance (Mode2)

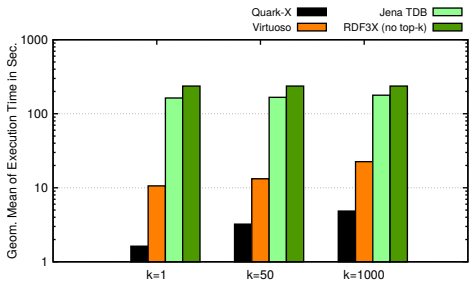
We now turn our attention to warm-cache query processing performance of systems under comparison. Jena and RDF-3X make use of only the operating system cache, whereas Virtuoso explicitly manages its own (somewhat large) cache. Therefore, in order to have a fair comparison, we present warm-cache results in two parts:

Mode 1: Figure 6 reports the results of comparison amongst Quark-X, Jena, and RDF-3X where all the systems use only O.S. caches. It can be seen that Quark-X outperforms Jena and RDF-3X significantly. Specifically, over DBpedia, Quark-X outperforms Jena by a speedup factor of about 22 and RDF-3X by a factor of 206.

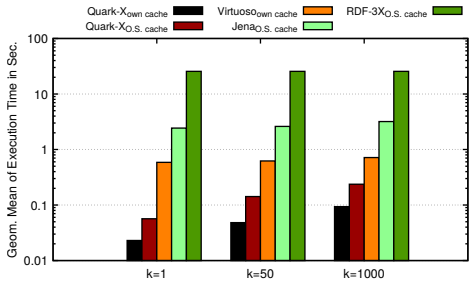
Mode 2: In this mode, we compared Quark-X and Virtuoso, both using their own internal caches. We suitably modified Quark-X also to cache the working set of the database, similar to Virtuoso. Since Quark-X does not yet use vectorization, the vector size of Virtuoso is set to 1. The results of this comparison are shown in Figure 7, where we find that Quark-X continues to outperform Virtuoso – by a speedup factor of 24.2 for Yago2S and 13.6 for DBpedia. Note that we do not report numbers for Virtuoso for Q_2 and Q_4 of DBpedia, because, somewhat surprisingly, it returned incorrect results.

Quark-X is slower than Virtuoso for queries Q_5, Q_9 and Q_{10} over DBpedia. On further analysis, we discovered that unlike Yago2S, DBpedia is highly unstructured leading to a huge number of characteristic sets, many of which occur only once. Specifically, DBpedia contains 197,530 characteristic sets and Yago2S contains 86 characteristic sets of quantifiable predicates. Large number of characteristic sets leads to fragmentation in id space – which naturally leads to increased random accesses. For a given set of quantitative predicates of Q_5, Q_9 and Q_{10} the *Non-Quantifiable Predicate Join (NQP-Join)* stage (cf. Section 5), of Quark-X's query processing engine has to look through many different characteristic set space – which is bound to cause a significant overhead. We can mitigate this by generating characteristic sets with approximate overlap.

Finally, Quark-X is more than an order of magnitude superior to all the other systems even when it retrieves all the required results e.g. for Q_6 of DBpedia. The good performance of Quark-X in comparison to Virtuoso and Jena is attributed to its efficient use of S-Indexes and reduced memory and index access due to increased data-locality induced by its novel semantically encoded identifiers.



(a) Cold-cache



(b) Warm-cache

Figure 8: Performance over DBpedia for Varying k

We also observe that the consistent poor performance of Jena in both cold as well as warm cache is due to its exclusive use of index-based joins leading to random access patterns during query processing.

9.3 Impact of Varying k

Next, we consider the impact of varying k. Figure 8 shows the geometric mean of all queries for each value of $k = \{1, 50, 1000\}$ in both cold- and warm-cache (the subscript denotes, whether the system uses its own cache or O.S. cache). Due to lack of space, we limit showing results only over DBpedia (results over Yago2S follow a similar trend).

One of the immediate effects that can be observed from these plots is that on increasing the value of k, systems which translate URIs and strings in RDF data into integer ids and back during final result generation are significantly affected. If the dictionary used for this translation is inefficient, for large values of k, the system can spend significant amount of time in its dictionary lookups during result generation. Due to this reason even though Virtuoso and Jena evaluate and output all results still they show slight variation with increasing value of k. Quark-X, in contrast, does not evaluate all results, thus, the variation of Quark-X's query processing performance is much greater. Additionally, it is noteworthy that Quark-X's performance in warm cache using its own cache is better than its performance using O.S. caches. This performance improvement is attributed to the elimination of decompression cost in Quark-X when it uses its own cache, as decompression is known to be beneficial for cold cache but is an unnecessary overhead for warm cache (when caching is performed by the O.S.).

Upon incrementing k, the use of block-wise processing in Quark-X also comes into play as follows: we process k fraction of buckets at a time until all k results are obtained. As we increase k, if all top results are obtained from the initial fraction of buckets, then we can see significant performance speedups. Only in specific queries which include many non-quantifiable facts, this feature does not play a crucial role.

10. CONCLUSION

This paper presents Quark-X, an efficient top-k query processing framework for RDF quad stores. The salient features of Quark-X include its indexes, viz., *S-Index* and *Q-Index*; the *Rank-Hash Join* query execution algorithm to adaptively choose the best index for joins; and the *Semantic Encoding* strategy used for increasing data locality. Through our experiments, Quark-X was shown to outperform existing frameworks by 1-2 orders of magnitude. As part of the future work, we plan to implement Quark-X enabled top-k RDF-graph reasoner.

11. REFERENCES

[1] D. J. Abadi et al. Scalable Semantic Web Data Management using vertical partitioning. In *Proc. of VLDB*, 2007.

[2] G. Aluç et al. Diversified Stress Testing of RDF Data Management Systems. In *ISWC*. 2014.

[3] C. Bizer et al. DBpedia - A Crystallization Point for the Web of Data. *J. Web Sem.*, 7(3), 2009.

[4] C. Bizer and A. Schultz. The Berlin SPARQL Benchmark. *Int. Journal of Semantic Web Inf. Syst.*, 5(2), 2009.

[5] P. Boncz et al. Advances in Large-Scale RDF Data Management. In *Linked Open Data–Creating Knowledge Out of Interlinked Data*. 2014.

[6] A. Deshpande et al. Adaptive Query Processing. *Foundations and Trends in Databases*, 1(1), 2007.

[7] S. Ding and T. Suel. Faster Top-k Document Retrieval using Block-max Indexes. In *SIGIR*, 2011.

[8] Y. Guo et al. LUBM: A Benchmark for OWL Knowledge Base Systems. *J. Web Sem.*, 3(2), 2005.

[9] P. Haas et al. Ripple Joins for Online Aggregation. In *SIGMOD*, 1999.

[10] S. Harris et al. SPARQL 1.1 Query Language. *W3C Recommendation*, 2013.

[11] J. Hoffart et al. YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia. *Artificial Intelligence*, 194, 2013.

[12] I. Ilyas et al. Rank-Aware Query Optimization. In *SIGMOD*, 2004.

[13] I. Ilyas et al. Supporting Top-k Join Queries in Relational Databases. *PVLDB*, 13(3), 2004.

[14] I. Ilyas et al. A Survey of Top-k Query Processing Techniques in Relational Database Systems. *CSUR*, 40(4), 2008.

[15] I. F. Ilyas et al. Adaptive Rank-Aware Query Optimization in Relational Databases. *TODS*, 31(4), 2006.

[16] Apache Jena-TDB 2.13 documentation. http://jena.apache.org/documentation/tdb/index.html, Jan 2016.

[17] J. Leeka and S. Bedathur. RQ-RDF-3X: Going beyond Triplestores. In *ICDEW - DESWEB*, 2014.

[18] J. Levandoski and M. Mokbel. RDF Data-Centric Storage. In *ICWS*, 2009.

[19] S. Magliacane et al. Efficient Execution of Top-k SPARQL Queries. In *ISWC*. 2012.

[20] F. Manola et al. RDF Primer. *W3C recommendation*, 10(1-107), 2004.

[21] S. Michel et al. KLEE: a Framework for Distributed Top-k Query Algorithms. In *Proc. of VLDB*, 2005.

[22] T. Neumann and G. Moerkotte. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In *ICDE*, 2011.

[23] T. Neumann and G. Weikum. RDF-3X: a RISC-style Engine for RDF. *PVLDB*, 1(1), 2008.

[24] T. Neumann and G. Weikum. Scalable Join Processing on very Large RDF Graphs. In *SIGMOD*, 2009.

[25] V. Nguyen et al. Don't like RDF Reification?: making Statements about Statements using Singleton Property. In *WWW*, 2014.

[26] H. Pang et al. Efficient Processing of Exact Top-k Queries over Disk-Resident Sorted Lists. *VLDB Journal*, 19(3), 2010.

[27] Rdf primer. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/, Feb 2004.

[28] M. Schmidt et al. SP^{2}Bench: A SPARQL Performance Benchmark. In *ICDE*, 2009.

[29] L. Sidirourgos et al. Column-Store support for RDF Data Management: not all Swans are White. *PVLDB*, 1(2), 2008.

[30] M. Sintek et al. RDFBroker: A Signature-based High-Performance RDF Store. *ESWC*, 2006.

[31] J. Urbani et al. KOGNAC: Efficient Encoding of Large Knowledge Graphs. In *IJCAI*, 2016.

[32] Virtuoso 7.2. http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtTipsAndTricksSPARQL11FeaturesExamplesCollection, May 2016.

[33] A. Wagner et al. Top-k Linked Data Query Processing. In *ESWC*. 2012.

[34] D. Wang et al. Top-k Queries on RDF Graphs. *Information Sciences*, 316, 2015.

[35] C. Weiss et al. Hexastore: Sextuple Indexing for Semantic Web Data Management. *PVLDB*, 1(1), 2008.

[36] K. Wilkinson. Jena Property Table Implementation. *SSWS*, 2006.

[37] D. Xin et al. Progressive and Selective Merge: Computing Top-k with ad-hoc Ranking Functions. In *SIGMOD*, 2007.

[38] S. Zahmatkesh. Retrieval of the most relevant Combinations of Data Published in Heterogeneous Distributed Datasets on the Web. *ISWC-DC 2014*.