Linked Data Query Processing Strategies Technical Report

Günter Ladwig, Thanh Tran

September 14, 2010

Abstract

Recently, processing of queries on linked data has gained attention. We identify and systematically discuss three main strategies: a bottom-up strategy that discovers new sources during query processing by following links between sources, a top-down strategy that relies on complete knowledge about the sources to select and process relevant sources, and a mixed strategy that assumes some incomplete knowledge, and discovers new sources at run-time. We propose an implementation of the mixed strategy. To exploit knowledge discovered at run-time, we propose an additional step, explicitly scheduled during query processing, called *correct source ranking*. Additionally, we propose the adoption of *stream-based query processing* to deal with the unpredictable nature of data access in the distributed Linked Data environment. In experiments, we show that our implementation of the mixed strategy leads to early reporting of results and thus, more responsive query processing, while not requiring complete knowledge.

1 Introduction

The amount of Linked Data on the Web is large and ever increasing. This development is exciting, paving new ways for next generation applications on the Web. We contribute to this development by investigating the problem of how to process queries against Linked Data. Linked Data query processing can be seen as a special case of federated query processing, i.e., to process queries against data that resides in different data sources. However, the highly distributed structure and evolving nature of Linked Data presents unique challenges.

- **Volume of the Source Collection**: According to the Linked Data principles [2], each URI can be dereferenced and the document returned represents a virtual "data source". This dramatically increases the *number of Linked Data sources* that need to be considered for query processing.
- **Dynamic of the Source Collection**: Linked Data sources are added and removed and sources' content changes rapidly over time. Due to this

dynamic, it is no longer safe to assume that information about all sources can be obtained. In particular, sources might be a priori *unknown* and can only be discovered at run-time.

- **Heterogeneity of Sources, Source Descriptions and Access Options**: Sources *vary in size*. There might be large sources, corresponding to Web databases today. Sources could also just comprise several RDF statements obtained via URI lookup. Further, there is *no standard for describing sources* yet. Not all sources are accompanied with a voiD[1] description and even if so, they are often incomplete. Also, the *range of access options is vast*. Sources can be obtained via HTTP lookup, retrieved from SPARQL endpoints or directly loaded from a local repository or cache. Even using the same access method, the time required to obtain the same amount of data might vary greatly due to network latency.

Recently, Harth et al. [5] proposed a probabilistic data structure that aims to improve the efficiency of Linked Data query processing. In order to deal with a large number of sources, rich statistics about them are acquired and stored locally. These statistics are used to determine the sources relevant for the query and to optimize query processing. Hartig et al. [6] proposed a method for dealing with the dynamic aspect of Linked Data query processing. As opposed to [5], the strategy employed here does not assume information about sources to be available. Sources are discovered via lookups of URIs found during query processing. Since data from sources might arrive at different times, this work introduces a *non-blocking iterator* to continue with processing even when data needed for the current operator is not yet available. We follow the direction of this line of work and make the following contributions:

- For Linked Data query processing, we identify the challenges, discuss concrete tasks, and derive three main strategies. There is a *top-down strategy* corresponding to the approach implemented by [5], a *bottom-up strategy* implemented by [6], and a *mixed strategy* that as opposed to [5], does not assume complete but only partial knowledge about the sources and unlike [6], have to discover only some but not all sources at run-time.

- We propose an implementation of the mixed strategy. It is able to use information discovered at run-time for *corrective source selection and ranking*. The proposed ranking scheme can deal with different types of source descriptions containing knowledge at varying levels of granularity.

- As an alternative to the pull-based non-blocking iterator [6], we propose the use of *push-* and *stream-based query processing* where source data is treated as finite streams that can arrive at any time in any order. This approach is better suited for dealing with network latency because it is driven by incoming data (i.e., push instead of pull) and thus does not require temporary rejection of answers as non-blocking iterators do.

[1] http://vocab.deri.ie/void/guide

We implement the proposed approach and perform an evaluation where we compare the mixed strategy with the bottom-up [6] and top-down [5] strategies. The results suggest that the implemented mixed strategy is able to report results much earlier than the bottom-up strategy, while not relying on the assumption that complete knowledge is available, as opposed to the top-down strategy. First results (25% of total results) were on average reported 42% faster than for the bottom-up strategy.

Outline In Section 2 we discuss techniques for Linked Data query processing. In Sections 3 & 4 we present our approach to stream-based query processing and corrective source ranking. Finally, we present related work in Section 5 before the discussion on evaluation results in Section 6 and the conclusions in Section 7.

2 Linked Data Query Processing

We begin with a discussion on Linked Data. For Linked Data query processing, we discuss the tasks (a) source discovery, (b) source ranking and for (c) query evaluation, we discuss the (1) top-down, (2) bottom-up and (3) mixed strategy.

2.1 Linked Data

For this work, we simply conceive Linked Data sources as sets of RDF triples [10].

Definition 1 *A* source *s is a set of RDF triples $\langle s, p, o\rangle \in T^s$ where s is the subject, p the predicate and o the object. It is uniquely identified by an URI and can be retrieved by dereferencing that URI. A source s_i* links *to another source s_j if the URI of s_j appears as the subject or object in at least one triple $t^s \in T^{s_i}$.*

The standard language for querying RDF data is SPARQL [13]. An important part of SPARQL queries are basic graph patterns (BGP). In this work we are concerned with answering BGP queries.

Definition 2 *A* basic graph pattern *is a set of* triple patterns *$\langle s, p, o\rangle \in T^q$ where every s, p and o is either a* variable *or a* constant*. Variables may interact in an arbitrary way such that the triple patterns $t^q \in T^q$ may form a graph.*

An answer to a BGP query is given by μ which maps patterns $t^q \in T^q$ to triples $t^s \in T^s$. By applying such a mapping, each variable in T^q is replaced by the corresponding subject, predicate or object of triples in T^s (called a binding). When processing queries over a set of Linked Data sources, the query is not evaluated on a single source, but on the graph formed by the union of all retrieved sources.

A BGP query is evaluated by performing a series of joins between RDF triples that match the triple patterns in the query. In particular, two triple patterns that share a variable form a *join pattern*.

There are several types of source descriptions the system might be able to obtain for a source: A *metadata description* is like a voiD description of the content. It captures basic information such as the size of the source, the RDF predicate it contains etc. *Statistics* capture detailed information that can be derived from the source data such as triple pattern cardinality, join pattern cardinality, histograms, etc. A representative *sample* of the source data might be available.

2.2 Source Discovery

There are multiple ways for sources to be discovered: Sources can be explicitly set in the *query* using special syntax or can be part of a triple pattern. The query engine can maintain a list of *known sources*. This list can either be entered manually or be compiled from previously executed queries. Sources can be *discovered* during query processing by following links mentioned in the content of retrieved sources.

In the first two cases, sources are known before the actual execution of the query. Compile-time optimization decisions concerning source ranking and query optimization (discussed in the following) are based exclusively on information derived from these sources. In the last case, sources are dynamically added at runtime. New information derived from these sources has an impact on the compile-time optimization plan. This information might render the plan made before query execution no longer optimal. It is used in our work for corrective query optimization.

2.3 Source Ranking

A source is *relevant* if it contains data that can contribute to the final answers. The standard optimization goal is to (1) obtain all results as fast as possible. However, given the volume and dynamic of the Linked Data collection, it is often infeasible to retrieve and process all sources. It is important to rank sources by their relevancy to the query and more fine-grained optimization goals. In particular, it might be desirable to (2) report results as early as possible, (3) to optimize the time for obtaining the first k results, or (4) to maximize the number of total results, given a fixed amount of time.

Source ranking uses available source descriptions that may vary in quality and completeness, i.e., they may lack information important for ranking. This means that it is essential to incorporate not only a priori available knowledge, but also knowledge discovered obtained query execution.

2.4 Query Evaluation Strategies

Top-Down Query Evaluation Linked Data comprises heterogeneous data that comes from different sources. Typically, a federated database system is used to integrate multiple sources and systems into one single federated database.

The goal is to obtain a fully-integrated virtual database that provides transparent access to data of all its constituent sources.

Typically, sources and databases are geographically decentralized in a federated system. However, a system, which discovers, retrieves and stores Linked Data sources centrally, also falls into the category of a federated system. In fact, no matter the physical location (and other characteristics) of the sources, a source is considered if and only if the federated system knows about it. The federated system assumes that *all source descriptions are available* and based on that, compiles a *query evaluation plan* that specifies the relevant sources, and the order for retrieving and processing these sources. Thus, query planning and optimization is a one-off process performed in a top-down fashion based on complete information.

Harth et al. [5] implement this top-down evaluation. The main focus is on using a data structure capturing rich statistics that can be used to improve query planning and optimization. In approaches that fall into this category, source discovery is performed offline and source ranking is not part of the process. In order to deal with the large amount of sources, source ranking based on approximative triple and join pattern cardinality estimation is used to consider only a fixed number of top-ranked sources.

Bottom-Up Query Evaluation As opposed to top-down query processing, this strategy does not assume source descriptions to be available beforehand and computes results in a bottom-up fashion. Without planning and optimization, it directly evaluates the query. During this process, it (1) retrieves the sources that are mentioned in the query, (2) discovers further sources based on source URIs and links found in the data of the retrieved sources, (3) incorporates the content of these discovered sources into query evaluation and (4) terminates when all sources found to be relevant have been processed.

Systems that implement this strategy do not rely on sources or source descriptions being managed centrally but discover and retrieve sources from external locations. *Source discovery* and *retrieval* are an integral part of the online process. These online tasks make this approach to Linked Data query processing different from traditional database approaches. They might be needed due to the Linked Data specific challenges we have discussed. The large volume and the dynamic of the sources and source collection render the traditional top-down approach impracticable. In particular, it cannot be applied when there are sources that are not known beforehand and can only be discovered during online processing.

Another aspect distinct to this approach is *completeness*. As opposed to traditional query processing, it might not be possible to obtain complete knowledge about all sources. In particular, processing queries against Linked Data where sources have to be discovered online might not yield all results. Results to the query cannot be found when they are part of sources that are unknown and cannot be discovered during online processing. This is the case when a link between two sources is only stored in one of the sources, meaning that the link cannot be discovered from the other source.

This bottom-up evaluation is implemented in [6]. Further, the authors mod-

ify the standard iterator-based approach to avoid blocking during query processing due to network delay. This issue is discussed in Section 3.2.

Mixed Strategy Query Evaluation This strategy combines the two other strategies by assuming that knowledge about some sources is available (the sources' data themselves are not necessarily locally available), and more knowledge can be obtained during online query processing. Compared to the top-down strategy, it does not rely on complete knowledge. Similar to the bottom-up strategy, online source discovery is an integral part of query processing. As opposed to that strategy, it makes use of knowledge available beforehand to do query planning and optimization. However, the plan built at compile time might be corrected according to newly acquired knowledge about sources. In particular, the additional optimization tasks that have to be performed online are *corrective source ranking* and *join order optimization*. Source ranking is not a by-product of query optimization [5], but explicitly scheduled as an integral task.

For processing queries on Linked Data, this strategy begins with (1) "best-effort" query planning, and based on this plan, evaluates the query. During this process, (2) sources are retrieved, (3) new sources are discovered, (4) new sources' content are incorporated into evaluation and in a continuous fashion, (5) new sources' descriptions are used for corrective source ranking and optimization. The evaluation proceeds with the continuously refined plan and (6) terminates when all relevant sources have been processed.

This mixed strategy explicitly addresses two of the challenges discussed previously. It uses online discovery to deal with Linked Data *volume* and *dynamic*. Also targeting the aspect of volume, compile-time combined with evaluation-time corrective source ranking and optimization are employed to make processing the large amount of sources affordable.

In the following, we discuss our implementation of this strategy that addresses also the remaining challenges. It features a novel approach for corrective source ranking that is designed to deal with Linked Data heterogeneity by exploiting the *different types of source descriptions* discussed previously. A stream-based query processing is employed to deal with the unpredictable nature of Linked Data resulting from *different source access options*, and to report results early.

3 Stream-based Linked Data Query Processing

We provide an overview of our approach to Linked Data query processing and then discuss stream-based evaluation based on push-based symmetric hash joins.

3.1 Overview of the Process

Query Planning A query plan is constructed during query compilation. We only consider left-deep query plans in this implementation, while in principle, query plans with other shapes such as bushy plans are possible. Depending on available source descriptions, basic information or detailed statistics as discussed

before can be used to plan the order of operators to be executed, and to perform other kinds of database optimizations that might consider indexes, materialized views, or the concrete join implementations [11]. Apart from joins, for Linked Data query processing, the operators we consider additionally include source discovery, source retrieval and source ranking. In this work we do not consider the general case of operator order optimization (and join order optimization) but focus on the specific aspect of corrective source ranking at run-time.

Query Evaluation For evaluating the query according to the query plan, we run each operator in a separate thread. Communication between operators is based on bounded message queues to enable parallel query processing. After query planning, threads for all operators are started. As a first step at run-time, local indexes are probed using the query triple patterns to obtain an initial list of possibly relevant sources, which is then sent to the source ranker.

Source Ranking The source ranker also runs in its own thread and receives source URIs, either obtained through discovery or from local indexes. It ranks the sources according to the methods described in Section 4. Ranking is performed only when necessary. The source ranker checks this continuously, using the parameters given in Section 4.4. If ranking is to be performed, the scores of all sources are calculated and normalized. The source ranker keeps track of the source retrieval threads and assigns them the top-ranked sources.

Source Retrieval Because of network delay it is usually necessary to request data from several sources at once, which is accomplished by running more than one source retrieval threads [6]. They filter the incoming data using the triple patterns of the query and push matching triples to the join operators as soon as they are decoded from the incoming data. This push-based join processing and the join operator are discussed in Section 3.2.

Source Discovery In addition to retrieving sources, the retrieval threads perform discovery of new sources based on the content of the source currently being processed. They notify the source ranker of all sources, which are linked from the source just found.

Termination Several termination conditions can be configured: (1) maximum discovery distance, (2) maximum number of sources to load and (3) number of results to produce. If any of these conditions are reached, the source ranker notifies the join operators so that query execution is terminated as soon

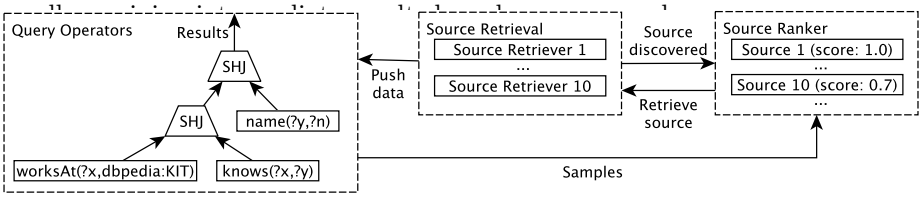

Figure 1: Join, source ranking and retrieval operators.

Example 1 *Fig. 1 shows an example for join, source ranking and source loading operators for a simple query consisting of three triple patterns* $\langle ?x, \text{worksAt}, \text{dbpedia:KIT}\rangle, \langle ?x, \text{name}, ?y\rangle, \langle ?y, \text{name}, ?n\rangle$*, which asks for the names of people known by employees of KIT.*

3.2 Push-based Symmetric Hash Join

Query processing in highly distributed environments, where data is often stored at remote locations, presents unique challenges. These environments require flexible scheduling: operators should not block, so that the query plan can make progress when input is delayed for another part [8]. In query plans using iterator-based ("pull-based") operators, the `next` method blocks until it is able to produce a result. Previously, non-blocking iterators [6] were proposed to address this problem in the context of Linked Data queries. A non-blocking iterator is able to temporarily reject input from iterators lower in the operator tree when it would otherwise block because of unavailable data on the other input. On the next call to its `next` method, the lower operator randomly either returns a new intermediate result or one of the previously rejected results, for which data might now be available in the upper iterator. This ensures that query processing can progress even if data for a particular triple pattern is not yet available. The advantage of this solution is that it can be used in existing query engines. However, while waiting for input to become available the query engine essentially performs busy-waiting in a loop by alternately asking for new results and then rejecting them. Even if no new data arrives the query engine is active, consuming CPU time.

To alleviate this problem, we propose the adoption of a stream-based approach where source data is treated as a (finite) stream that can arrive in any order. To process such streams pipelined operators are required that produce results even before the whole input has been read. Query plans using these operators can be implemented using threads and message queues, taking advantage of multi-core and parallelization capabilities of modern CPUs.

Algorithm 1: SHJ: $push(in, t)$

Input: Operator *in* from which input tuple t was pushed
Data: Hash tables H_1 and H_2; current operator *this*; subsequent operator *out*

```
1 if in is left input then m = 1, n = 2
2 else m = 2, n=1
3 Insert t into hash table H_m
4 Probe H_n with join keys of t
5 forall valid join combinations j do out.push(this,j)
```

One such operator is the symmetric hash join (SHJ), which, in constrast to traditional hash joins, can start reporting results as soon as input tuples arrive in the operator and does not have to wait until one of the input has been completely read [15]. This is achieved by maintaing one hash table on each input. Instead of a pull-based iterator, we employ the SHJ in a push-based mode where operators are driven by their inputs. Instead of a `next` method that is called by operators higher in the operator tree, the join operator has a `push` method for each of its inputs. Algorithm 1 shows the operation of this method in a SHJ operator. First, the arriving tuple t is inserted into the hash table

H_m that corresponds to the input where the tuple belongs to (i.e., $H_m = H_1$ or H_2). Then, t is used to probe hash table H_n for valid join combinations. All such valid join combinations are then immediately reported to subsequent operator *out* by calling its `push` method. Pushing is done from the operators corresponding to the leaf nodes of the operator tree to the root operator. The root operator pushes early results to the caller of the query evaluator. Compared to blocking operators such as the hash join, the SHJ produces results as soon as input tuples are available and input tuples can arrive on all inputs in any order.

4 Corrective Source Ranking

The relevance of a source depends on several factors and is measured based on the current query, any available intermediate results and one of the overall optimization goals as discussed in Section 2.3. In this section, we elaborate on the source features that are taken into account, concrete metrics derived from them, the indexes used to compute the metrics, newly discovered information used to refine and correct previously computed metrics, and how they are incorporated into source ranking.

4.1 Source Features and Metrics

We identify the following features as important for the relevancy of a source:

Triple Pattern Results A source is more relevant if it contains data that contributes to answers of the query. Thus, a source is relevant if it contains triples matching a query triple pattern. The estimation of triple pattern results is based on the metrics triple pattern cardinality and triple pattern specificity.

Definition 3 (Triple Pattern Cardinality) *The* triple pattern cardinality $card(s, t)$ *gives the number of triples in source* s *that match the triple pattern* t*.*

Definition 4 (Triple Pattern Specificity) *The* triple pattern specificity $spec(t)$ *gives the number of constants that occur in the triple pattern* t*.*

Clearly, the higher the cardinality and the more specific the triple pattern, the more relevant is a source matching that pattern. However, these two metrics alone are yet no good indicator for the relevance of a source. Given the power-law distribution of the Web of Data [4], some triple patterns might have a high cardinality for all or many sources. These patterns do not discriminate sources, just like words that frequently occur in all documents of a collection. One example is $\langle ?x, \textit{rdf:type}, ?y \rangle$, which can be found in most Linked Data sources. To alleviate this problem, we adopt the TF-IDF concept to obtain weights for triple patterns (capturing their *importance*). Similarly to words in IR, the importance of a triple pattern positively correlates with how often bindings to this pattern occur in a source as measured by its cardinality, and negatively correlates with how often its bindings occur in all sources of the collection. Higher weight is thus given to discriminative triple patterns.

Definition 5 (TF-ISF) *Given a source s and a triple pattern t, the* triple frequency - inverse source frequency *(TF-ISF) measure is defined as* $\text{TF-ISF}(s,t) = card(s,t) \cdot \log \frac{|S|}{|\{r \in S | card(r,t) > 0\}|}$ *where S is the set of all sources to be ranked.*

Join Pattern Results A source containing data matching larger parts of the query is more relevant. Thus, a source that contributes data matching a join pattern is considered highly relevant. However, not containing data for a join does not render a source irrelevant as its data might be joined with data from other sources. The join pattern cardinality is used for estimating results of a join pattern.

Definition 6 (Join Pattern Cardinality) *Given the join pattern $t_i \bowtie_v t_j$ on the shared variable v, the* join pattern cardinality *of a source s denoted $card(s, t_i, t_j, v)$ gives the number of results a join on the variable v between triples retrieved from s for t_i and t_j produces.*

Links to Results A source containing many links coming from relevant sources is more useful. The relevance of such sources is even higher when these links match query predicates. Note that unlike triple pattern results that can be computed given a source, links can only be discovered by processing several sources. A source at first considered irrelevant based on triple pattern results might become relevant during the process. For measuring links to results, links to other sources are extracted from sources discovered during the process.

Definition 7 (Links to Results) *Let S be the set of sources already processed, $links(s_i, s_j)$ be a function that return all links between a source $s_i \in S$ and the source s_j, the* links to results *of s_j is defined as $links(s_j) = \bigcup_{s_i \in S} links(s_i, s_j)$.*

Retrieval cost Sources are more useful the faster they can be retrieved.

Definition 8 (Retrieval Cost) *The* retrieval cost *of a source s is a monotonic aggregation of the* size *of s and the* bandwidth *of a host h, defined as $cost(s) = Agg(size(s), bandwidth(h))$.*

Source size is available in the source description. Bandwidth is approximately derived for a particular host based on past experiences or, when available, average performance recorded during the process for sources retrieved from this host.

4.2 Metric Computation

In the mixed strategy, some of the source metrics are available locally. We store these metadata in specialized indexes (1) to select relevant sources and (2) to compute cardinalities for these sources.

Indexes for Source Selection Given a triple pattern, these indexes return a set of sources that contain triples matching the pattern. The only "interesting" patterns are those with one or two variables. Patterns with no variables match

only themselves and pattern with no constants match all triples and thus, match all sources. Three indexes are sufficient to support all patterns with one variable. In particular, we create the indexes SP, PO and OS (where S, P, O stand for subject, predicate and object). Each maps the indexed pattern to a set of sources. For example, to find sources for $\langle ?x, \textit{rdf:type}, \textit{foaf:Person} \rangle$, we use the PO index and retrieve relevant sources using the predicate *rdf:type* and the object *foaf:Person*. Using prefix lookup, the same indexes can be used to cover all patterns with two variables.

Index for Cardinality Computation In [5], a probabilistic index structure is used to support triple and join pattern cardinality estimation of individual sources. A different technique based on aggregation indexes is presented in [12]. We adopt this method, but extend it to support lookup of triple pattern cardinalities and estimation of join cardinalities for individual sources. Instead of calculating the statistics and indexes for the whole dataset, we treat each source as its own dataset and create the aggregation indexes accordingly. While we lose the ability to perform selectivity and cardinality estimation over the indexed data as a whole, we can now calculate estimates for individual sources, which is what is necessary for source ranking.

4.3 Metric Correction and Refinement

During query processing as sources are retrieved and their data is processed, more information becomes available to compute new or to refine and correct previously computed metrics. This is especially important in the case of very general "non-discriminative" triple patterns, such as $\langle ?x, \textit{rdf:type}, ?y \rangle$. When such a pattern is joined with another pattern, it is more or less by chance that matching join combinations are found.

When processing queries over data that is stored and indexed locally, this problem can be alleviated by performing index nested-loop joins. An index nested-loop join between two triple patterns t_1, t_2 uses triples that match t_1 to instantiate triple pattern t_2 by replacing variables with bindings of the join variables in triples matching t_1. This creates more specific triple patterns which are then used perform index lookups to retrieve further data that is guaranteed to create join combinations.

In the case where data is not locally available, we cannot perform such joins. However, we employ a similar technique to estimate join pattern cardinalities, taking into account current intermediate results and information in the cardinality indexes. In particular, a triple pattern of a join is instantiated with intermediate results and then used to perform lookups on the triple pattern cardinality indexes to calculate better join cardinality estimates:

Definition 9 (Join Pattern Cardinality Estimate) *Let t_i, t_j be two triple patterns, T_i^s a set of triples in s matching the pattern t_i, and $T_i^s(v)$ denotes the set of bindings to the variable v of the triple pattern t_i. Based on triple pattern cardinalities, a* cardinality estimate *of a join $t_i \bowtie_v t_j$ is calculated as $card(s, t_i, t_j, v) = \sum_{b \in T_i^s(v)} card(s, t_j.inst(v, b))$, where $t_j.inst(v, b)$ denotes the*

instantiation of the variable v of the triple pattern t_j with the binding b.

The SHJ operators used for query evaluation maintain hash tables on both of their inputs, storing data by the join attribute. The data of a source indexed in a hash table is used to instantiate the triple patterns of the join to obtain more specific triple patterns. Then, the cardinality of these more specific patterns is looked up using the index and aggregated to obtain an estimate for the size of the join. In order to reduce the cost of this process, we perform *sampling* to estimate the join cardinality by instantiating the triple pattern with only a random subset of the triples. Sampling has been used in database research to perform estimation of join cardinalities, see Section 5 for related work on this topic.

4.4 Source Ranking at Run-time

In our implementation we prioritized early result reporting, i.e., producing results as early as possible is the optimization goal. First, for every indexed source, we calculate the TF-ISF measure for all query triple patterns. In order to produce early results the join cardinality is important. We employ both methods for join cardinality estimation: using join pattern indexes and sampling from join states obtained during query processing. Less information is available for sources that are not indexed and were only discovered during query processing. No join cardinality estimation is performed for these sources. For all sources, however, the count and type of incoming links are available. In particular, we follow *owl:sameAs* and *rdfs:seeAlso* links as well as links that have a predicate that occurs in a query triple pattern. Links with query predicates receive a higher weight than others as these are more likely to deliver results. Finally, all scores are normalized separately and then combined using a monotonic aggregation function, in this case a weighted summation.

Ranking of sources is not a one-off process but needs to be done continuously during query processing as new sources and more information about already known sources are discovered. However, ranking also represents an overhead, and therefore should be executed when "necessary". It is too expensive to recalculate the scores for all sources as soon as a new source is discovered or a new triple is pushed to the join operators. We define several parameters that are used to influence the behavior and cost of the ranking process: (1) *Invalid Score Threshold*: the score of a source is invalid if it has not been calculated before, or if new information about the source is available. A ranking is performed when the number of invalid scores passes a threshold. (2) *Sample Size*: using larger samples for join size estimation will give better estimates, but are also more costly to use. (3) *Resampling Threshold*: results of previous join size estimates are cached for each indexed source. Only when the corresponding hash table maintained by the join operator grows over a given threshold, join size re-estimation is performed using a new sample.

5 Related Work

Seminal work on Linked Data query processing [6, 5] and some concrete techniques related to our work have been discussed throughout the paper. Here, we summarize the relation between the proposed corrective ranking and steam-based processing techniques to database work on query optimization and processing in an distributed environment.

Query Optimization One main problem of query optimization is finding the optimal join order optimization. To do that, it is necessary to estimate their selectivity. Histograms [14] and more complex probabilistic data structures have been suggested to store and estimate selectivity information of RDF triples. In [12], aggregation indexes are used to improve the accuracy of selectivity estimation for joins between triple patterns. As discussed in Section 4.2, we extend these indexes to estimate the cardinality of joins for individual sources (instead of the entire source collection).

Compared to these approaches, [9] does not perform compile-time join ordering, but optimizes the query at run-time by using chain sampling to estimate the selectivity of joins that were not yet performed. In our work, we use sampling combined with triple pattern cardinality indexes to estimate the cardinality of joins given data in a particular source.

Sideways information passing has been employed to complement compile-time optimization with a run-time decision-making technique for reusing intermediate states from one query part to prune and reduce computation of other parts [12, 8]. The feedback process between query execution and source ranking employed in our approach for metric refinement can be seen as a case of sideways information passing.

Query Processing in Distributed Environments In distributed environment data is often stored in remote locations, causing delays in data access. Much database research has been focused on compensating for these delays. Widely used for this are pipelineable query operators that operate on streams. As discussed in Section 3.2, the symmetric hash join is one such operator. Another aspect of stream-based query processing is adaptivity. Query processing techniques have been proposed to adapt the query plan at run-time to deal with changing characteristics of the data. One technique is to switch among query plans at run-time [7]. Other techniques use special operators, such as Eddies [1] and STAIRs [3] that adaptively route incoming tuples through a series of query operators.

Comparison Our work is the first to provide a systematic overview of Linked Data query processing. The specific techniques proposed extend related work in database research to deal with the specific aspects of Linked Data. In particular, whereas selectivity information has been used for query optimization [12, 14, 5], it is incorporated in this work into a framework for source ranking, a task that is novel and specific to Linked Data query processing. Likewise, the ideas behind stream-based and adaptive processing [7] and sideways information passing techniques [8] are adopted to address the specific challenges, to enable corrective source ranking on Linked Data streams.

6 Evaluation

In the experiments, we systematically compare the three Linked Data strategies and examine the impact of various parameters on corrective source ranking.

Queries and Data We create a set of eight queries that can all be executed using a discovery-only approach (i.e. results can be discovered by exploring from sources mentioned in the query). These queries use popular datasets from the Linked Open Data Project, such as DBPedia, Geonames, DBLP, Semantic Web Dog Food, data.gov, Freebase and others. Overall, during answering these queries, 6200 sources were retrieved containing 500k triples in total.

Systems We compare the approaches proposed in [6] for bottom-up evaluation (BU), [5] for top-down (TD), and our implementation of the mixed (MI) strategies. All approaches were implemented on top of the same stream-based query engine using Scala and Java. We randomly chose 25% of the sources from the complete index of TD to construct a partial index for MI. Note that these indexes are used for obtaining source descriptions, but the actual data used for query processing comes from remote hosts.

Setting To obtain a controlled environment to systematically investigate different aspects of the discussed strategies, we simulate the Linked Data environment by creating a local and configurable cache of relevant sources as follows. We executed all queries against real Linked Data sources available on the Web, recorded all HTTP accesses and their responses (200 OK, 404 Not Found, 30x Redirect), and locally stored the obtained RDF documents. Next, we set up a proxy server on the local network that uses the previously recorded data to respond to requests by the query engine. All requests for unknown URIs were answered with a Not Found response. Because local access has lower latency than remote, we applied a configurable delay to the proxy server. This enables us to control the impact of network latency. For this evaluation we used a latency of 2s, whereas in real condition this can be much higher. The evaluation was executed on a quad-core system with Intel Xeon 2.8GHz CPUs and 8GB memory, 2GB of which were assigned to the Java VM.

6.1 Comparison of Strategies

The strategies under investigation vary w.r.t completeness of results. The bottom-up strategy finds only sources and results that can be discovered by following links, the mixed strategy usually finds some more, and the top-down strategy finds all of them. To make the approaches comparable, we restrict the sources to those that can be considered by all strategies, i.e., those discovered by the BU strategy.

Table 1 shows the results for all queries, capturing the times needed to obtain (some percentage of the) results, and the specific times needed for source selection and ranking. The results show that for all queries, the MI and TD approaches report results earlier than BU. The benefit lies in the use of prior knowledge about sources, which helps to retrieve more relevant sources first. Less expected, MI outperformed TD in some cases (Q1,Q3,Q5,Q6,Q7,Q8) in terms of early reporting. The cause lies in the higher source selection times

	BU	MI	TD	BU	MI	TD	BU	MI	TD
	Q1			Q2			Q3		
25% res.	24810.5	10300.0	11038.0	10464.5	10162.0	8096.5	9207.0	7900.0	11166.0
50% res.	43464.5	40782.0	15787.0	13080.5	17974.5	8327.0	10568.0	8048.5	11391.5
Total	84066.5	86895.5	44323.5	21623.5	23273.0	21428.0	22711.0	21944.0	21733.5
Src. sel.	0.0	853.0	1444.5	0.0	805.0	1280.0	0.0	1211.0	1717.0
Ranking	25.5	2404.0	411.0	32.5	358.0	196.5	32.0	575.5	523.0
#Sources	622.0	612.0	154.0	120.0	120.0	67.0	134.0	134.0	67.0
	Q4			Q5			Q6		
25% res.	56800.5	26025.5	10969.5	16837.5	6580.5	4177.0	8222.5	4743.5	5545.0
50% res.	56804.5	26047.0	13605.0	21578.5	11855.5	9186.0	10961.5	7650.5	5634.0
Total	98129.0	98931.0	91352.0	29562.0	30603.5	20074.0	24086.0	20711.0	16469.0
Src. sel.	0.0	270.0	351.0	0.0	203.0	292.0	0.0	1331.0	1863.5
Ranking	31.0	3173.5	1358.5	25.5	283.5	414.5	23.5	292.5	335.0
#Sources	392.0	390.0	342.0	119.0	117.0	70.0	236.0	92.0	49.0
	Q7			Q8					
25% res.	7164.0	3636.5	3710.5	42029.0	33740.0	14929.0			
50% res.	9578.5	6503.5	3753.0	61726.5	34704.5	14943.0			
Total	24250.0	20630.0	6780.5	91405.5	91093.0	90360.5			
Src. sel.	0.0	287.5	333.0	0.0	1242.0	1821.0			
Ranking	25.0	281.5	181.0	25.0	2751.0	1354.5			
#Sources	119.0	98.0	16.0	368.0	365.0	332.0			

Table 1: Execution times for the evaluation queries. Times in ms.

resulting from the use of a larger index. On average the time to retrieve 25% and 50% of the results was 8.7s and 12.8s for MI and 15.1s and 22.0s for BU, respectively. This is an improvement of about 42% in both cases, which may increase with higher, more realistic latencies where the impact of ranking will be higher.

In terms of total execution time, MI and BU approaches are comparable, while TD is significantly faster in most cases. While TD incurs more overhead for the initial source selection because of the larger index, it enables the exclusion of sources. Due to the high network cost, not retrieving irrelevant sources results in a significant performance gain. Using a partial index covering 25% only, MI is not able to restrict the number of sources that have to be retrieved. This means that in the end MI processes almost the same sources, same data and thus does the same work as BU. The additional overhead incurred by source selection, ranking and sampling lead to execution times worse than BU in some cases (Q1,Q2,Q6). However, Note that MI was able to process more useful sources and results earlier. To better illustrate the behavior of the different approaches, Fig. 2 shows the arrival of results over time for queries Q4 and Q6. For Q4, the first result for TD was produced after less than 10s and all results were reported after 33s. The difference to overall execution time of about 90s given in Table 1 is due to the fact that even after the final result was reported other relevant

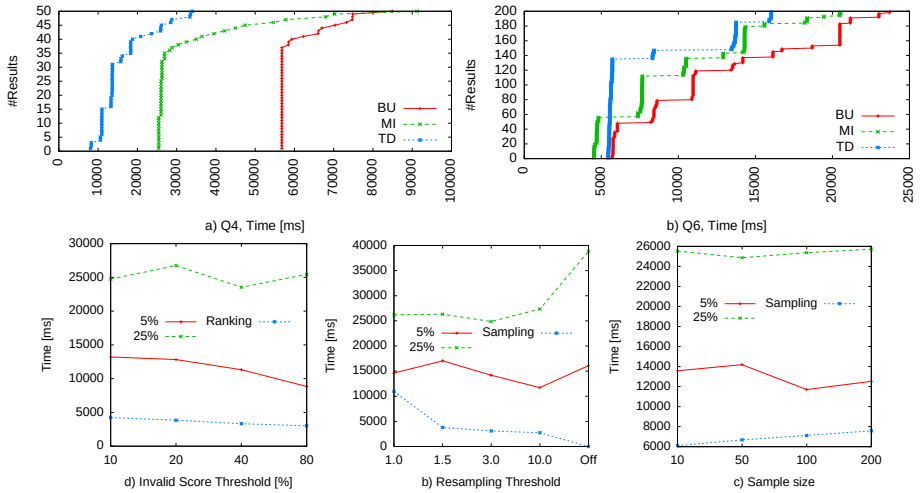

Figure 2: a+b: Result arrival times for two queries Q4 and Q6. d-f: Effects of invalid score threshold, sample size and resampling threshold.

sources had to be processed, but did not contribute to the final result. This indicates that early result reporting resulting in better responsiveness is very important in some cases, where processing all source might be very costly and not needed. Clearly, TD produced results earlier than MI, which was better than BU. In the end, BU caught up and was faster than MI. A similar pattern can be seen for Q6, except that MI started reporting results earlier than TD, because of the lower overhead of the smaller index.

6.2 Corrective Source Ranking

In this part we examine the influence of various parameter configurations on sampling and ranking. To separate the effect of each parameter, we vary one while setting the other parameters to default values 40% for invalid score threshold, 3 for resampling and 50 for sample size.

Invalid Score Threshold Fig. 2d shows average query times for computing 5% and 25% of the results and for sampling at different invalid score thresholds from 10%-80%. With increasing threshold, ranking is performed less often, and correspondingly, times for ranking decreased. The effect of performing ranking less often was positive for computing 5% results, but no clear trend could be observed for 25% results, where the best time was observered for a threshold of 40%. Ranking is beneficial as query execution is more guided and sources that directly contribute to join results are preferred, especially by using join cardinality estimation with sampling.

Resampling Threshold Fig 2e shows that times for sampling decrease with higher resampling thresholds, as sampling is performed less often. Times for 5% and 25% results are best for a threshold of 1.5 and 3, respectively. Clearly, sampling is better than no sampling, because the time to reach 25% of results is the highest when sampling is off.

Sample Size Fig 2f shows that times for sampling increased as the sample grows larger. While sampling creates an overhead, it also provides benefits. Larger sample sizes can lead to more accurate cardinality estimates. Thus, total effect on result computation times varies. While the time for 25% results stayed largely the same, time for 5% results was clearly best for a sample size of 100.

7 Conclusion

We provided a systematic analysis of the challenges and tasks, and discussed concrete strategies for linked data query processing. We proposed a concrete implementation of the mixed strategy that most closely mimics a realistic linked data scenario where some partial knowledge of linked data sources are available. The implementation can exploit different types of knowledge available beforehand, and also, incorporate information discovered during query processing to support corrective source ranking. The scheme proposed for ranking specifies various types of metrics, which can be combined to reach different optimization goals. Besides, a stream-based processing technique is adopted to deal with the unpredictable nature of linked data access. The experiments showed that while not requiring complete knowledge, the proposed implementation leads to early reporting of results and thus, more responsive query processing. On average early results were reported 42% faster than for the bottom-up strategy. In the linked data scenario where response times are very high due to the large number of sources and network latency, the capability to produce early results is essential.

As future work, we aim to use information discovered at run-time not only for source ranking but for optimizing the entire evaluation process. In particular, we will target the problem of run-time corrective query optimization to refine the query plan and the join order determined at compile-time.

Acknowledgements Research reported in this paper was supported by the German Federal Ministry of Education and Research (BMBF) under the CollabCloud project (grant 01IS0937A-E).

References

[1] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. *SIGMOD Rec.*, 29(2):261–272, 2000.

[2] C. Bizer, T. Heath, T. Berners-Lee, T. Heath, M. Hepp, and C. Bizer. Linked data - the story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 2009.

[3] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *Proceedings of the Thirtieth international*

conference on Very large data bases - Volume 30, pages 948–959, Toronto, Canada, 2004. VLDB Endowment.

[4] W. Ge, J. Chen, W. Hu, and Y. Qu. Object link structure in the semantic web. In *The Semantic Web: Research and Applications*, pages 257–271. 2010.

[5] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K. Sattler, and J. Umbrich. Data summaries for on-demand queries over linked data. In *Proceedings of the 19th international conference on World wide web*, pages 411–420, Raleigh, North Carolina, USA, 2010. ACM.

[6] O. Hartig, C. Bizer, and J. Freytag. Executing SPARQL queries over the web of linked data. In *The Semantic Web - ISWC 2009*, pages 293–309. 2009.

[7] Z. G. Ives, A. Y. Halevy, and D. S. Weld. Adapting to source properties in processing data integration queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 395–406, Paris, France, 2004. ACM.

[8] Z. G. Ives and N. E. Taylor. Sideways information passing for Push-Style query processing. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 774–783. IEEE Computer Society, 2008.

[9] R. A. Kader, P. Boncz, S. Manegold, and M. van Keulen. ROX: run-time optimization of XQueries. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 615–626, Providence, Rhode Island, USA, 2009. ACM.

[10] G. Klyne, J. J. Carroll, and B. McBride. Resource description framework (RDF): concepts and abstract syntax, 2004.

[11] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *Proc. VLDB Endow.*, 1(1):647–659, 2008.

[12] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 627–640, Providence, Rhode Island, USA, 2009. ACM.

[13] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, 2008.

[14] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceeding of the 17th international conference on World Wide Web*, pages 595–604, Beijing, China, 2008. ACM.

[15] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, Jan. 1993.