SIHJoin: Querying Remote and Local Linked Data

Günter Ladwig and Thanh Tran

Institute AIFB, Karlsruhe Institute of Technology, Germany
{guenter.ladwig,ducthanh.tran}@kit.edu

Abstract. The amount of Linked Data is increasing steadily. Optimized top-down Linked Data query processing based on complete knowledge about all sources, bottom-up processing based on run-time discovery of sources as well as a mixed strategy that combines them have been proposed. A particular problem with Linked Data processing is that the heterogeneity of the sources and access options lead to varying input latency, rendering the application of blocking join operators infeasible. Previous work partially address this by proposing a non-blocking iterator-based operator and another one based on symmetric-hash join. Here, we propose *detailed cost models* for these two operators to systematically compare them, and to allow for query optimization. Further, we propose a novel operator called the *Symmetric Index Hash Join* to address one open problem of Linked Data query processing: to query not only remote, but also local Linked Data. We perform experiments on real-world datasets to compare our approach against the iterator-based baseline, and create a synthetic dataset to more systematically analyze the impacts of the individual components captured by the proposed cost models.

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 reside in different data sources. However, the highly distributed structure and evolving nature of Linked Data presents unique challenges. In particular, as discussed in [6], the number of Linked Data sources is large (*volume*); sources evolve quickly (*dynamic*); sources vary in size, there is no standard for source descriptions, and access options vary (*heterogeneity*). As source descriptions, Harth et al. [2] proposed a probabilistic data structure to capture and store locally rich statistics about remote sources, used to determine relevant sources and to optimize query processing. Hartig et al. [3] proposed a method for dealing with the dynamic aspect of Linked Data query processing. As opposed to [2], the strategy employed here does not rely on complete knowledge about Linked Data available as source descriptions but is based on run-time source discovery via URI lookups. In previous work [6], we

G. Antoniou et al. (Eds.): ESWC 2011, Part I, LNCS 6643, pp. 139–153, 2011.

proposed a mixed strategy that is able to leverage locally stored source descriptions if they exist, and to discover other sources at run-time.

Partially, our previous work also elaborates on the aspect of *join implementation*. In this setting, standard blocking iterator-based join is not optimal due to possibly very high network latency (as a result of Linked Data heterogeneity). Instead of blocking, Hartig et al. [3] proposed a non-blocking iterator-based join operator (NBIJ) that employs a busy-waiting strategy. Basically, it is a workaround that temporarily rejects inputs when the other inputs needed for the join are not available. In previous work [6], we discussed a conceptually cleaner strategy called push- and stream-based query processing based on the symmetric hash join (SHJ) [15], where source data is treated as finite streams that can arrive at any time in any order. In theory, this strategy is better suited to deal with network latency as it is driven by incoming data (i.e., push- instead of pull-based) and thus, does not require temporary input rejection. However, a systematic comparison of these two strategies at the conceptual level as well as experimental results that can validate possible differences are missing.

Another problem is that so far, joins are performed on remote data, but in practice, Linked Data (can be imported and) might be available locally, giving rise to non-blocking operators capable of *processing both remote and local data*.

Contributions. In this work we focus on join operators:

- We propose a new join operator called Symmetric Index Hash Join (SIHJ) that is non-blocking, pushed-based, stream-based, and in particular, is able to process both remote and local linked data.
- We propose a cost model that can be used to analyze this operator given only remote data, only local data, or a combination of them. Further, we provide a cost model for the proposed NBIJ [3]. These two cost models can be used for query optimization, and allow us to compare the mechanisms underlying these operators in a systematic fashion.
- In an experimental comparison, we evaluate these two approaches on real-world datasets and a synthetic dataset to more systematically analyze the impacts of the individual components captured by the proposed cost models.

Outline. In Section 2 we introduce Linked Data query processing and motivate our approach. Section 3 presents and analyzes the symmetric index hash join operator. We compare the SIHJ operator to the previously proposed NIHJ in Section 4. Finally, we present related work in Section 5, before discussing the evaluation results in Section 6 and the conclusions in Section 7.

2 Preliminaries

Linked Data Query Processing. As usual [3,6], we conceive Linked Data sources as interlinked sets of RDF triples [5]:

Definition 1. *A* source d *is a set of RDF triples* $\langle s^d, p^d, o^d \rangle \in T^d$ *where* s^d *is called the subject,* p^d *the predicate and* o^d *the object. There is a function* ID

which associates a source d with a unique URI. There is a link *between two sources d_i and d_j if the URI of d_i appears as the subject or object in at least one triple of d_j, i.e., $\exists t \in T^{d_j} : s^d(t) = ID(d_i) \vee o^d(t) = ID(d_i)$; or vice versa, i.e., $\exists t \in T^{d_i} : s^d(t) = ID(d_j) \vee o^d(t) = ID(d_j)$ (then d_i and d_j are said to be* interlinked*). The union set of interlinked sources $d_i \in D$ constitutes the* Linked Data $T^D = \{t | t \in T^{d_i}, d_i \in D\}$.

The standard language for querying RDF data is SPARQL [10]. An important part of SPARQL queries are basic graph patterns (BGP). Work on Linked Data query processing so far focused on the task of answering BGP queries. In this work we focus on BGP queries that form a connected graph, i.e., can be answered without cross products:

Definition 2. *A* connected basic graph pattern *q is a set of* triple patterns $\langle s^q, p^q, o^q \rangle \in T^q$ *where every s^q, p^q and o^q is either a* variable *or a* constant*. There are some variables appearing in several patterns $t^q \in T^q$ such that together, the set of patterns T^q forms a connected graph.*

Since Linked Data triples in T^D also form a graph, processing queries in this context amounts to the task of graph pattern matching. In particular, an *answer* (also called a *solution mapping* or *query binding*) to a BGP query is given by μ which maps the variables in query graph pattern T^q to RDF terms in T^D, such that applying the mapping by replacing each variable with its bound RDF term yields a subgraph T_q^D of T^D. We denote the set of solution mappings for BGP query as Ω and the set of partial solution mappings or bindings for a single triple pattern $t^q \in T^q$ as Ω_{t^q}.

A BGP query is evaluated by retrieving triples matching the patterns $t^q \in T^q$, and by performing a series of joins between the bindings Ω_{t_q} created from the triples retrieved. In particular, this is done for every two triple patterns that share a variable, forming a *join pattern* (that variable is referred to as the *join variable*).

In the Linked Data context, triple patterns are not evaluated on a single source, but have to be matched against the union of all sources D. When all sources in D are known, sources needed for processing a given query can thus be determined, retrieved by dereferencing their URIs, and triples obtained from these sources are joined along query join patterns [2]. In contrast to standard federated query processing triples from Linked Data can only be obtained via URI lookups, as opposed to retrieving only those matching a given pattern.

Exploration-based Linked Data Query Processing. In this work, we consider the case more general than in [2], following the direction of *exploration-based* Linked Data query processing, which does not rely on having complete knowledge about all Linked Data sources [3,6]. This line of approaches deal with the case where Linked Data is assumed to be dynamically evolving such that obtaining results might also require run-time discovery of new sources based on link traversal. In [3], no knowledge is available at all, and the query is assumed to contain at least one constant that is a URI. This URI is used for retrieving the first source representing the "entry point" to Linked Data; new sources are then discovered in a bottom-up fashion beginning from links found in that

entry point. The approach recently proposed in [6] combines the two previous approaches [2,3] to discover new sources as well as leveraging known sources.

Here, query processing also relies on performing joins according to a query plan. The difference is that sources have to be discovered and even in the case where some knowledge is available, all source data are assumed to be remote and thus have to be retrieved from the network. Dealing with the network latency resulting from this requires that join operators do not block, such that when input is stalled for one part, progress can be made for other parts of the query plan. Two alternatives have been proposed to deal with this problem: one is *NBIJ* and the other is the *push-based SHJ*. Through a more systematic study of these operators we will show based on detailed cost models that the push-based SHJ is less expensive. Further, it also computes all results from retrieved data, while this completeness guarantee cannot be provided by NBIJ.

Remote and Local Linked Data Query Processing. While all approaches proposed so far assume remote data, in realistic scenarios, some Linked Data may be available locally. Conceptually, local data can be seen as yet another source. Thus, a *basic solution* to integrate locally stored data is to treat them just like a remote source and process them in the same way.

However, the availability of local data makes a great difference in practice, because while remote Linked Data sources have to be retrieved entirely (only URI lookup is available), local data can be accessed more efficiently using specialized indexes. Typically, local data are managed using a triple store, which maintains different indexes to directly retrieve triples that match a given pattern, i.e., relevant bindings Ω_{t^q} of a local source d can be directly obtained for $t^q \in T^Q$.

Given such querying capabilities for local data, we will show in this work that remote and local Linked Data with different access options can be processed using a single join operator. Instead of loading all local data, this operator retrieves only triples matching a given pattern. Further, we observe that there are non-discriminative triple patterns such as $\langle ?x, \textit{rdf:type}, ?y \rangle$, which produce a large number of triples that do not contribute to the final results. To alleviate this problem, we take advantage of the available indexes to further instantiate query triple patterns with data obtained during query processing to load only triples that are guaranteed to produce join results.

3 Symmetric Index Hash Join

We propose to extend the SHJ operator to obtain the index-based SIHJ operator that can also take advantage of local data and indexes. This operation is similar to the index nested-loop join operator, where tuples (i.e., bindings) of one input are used to access indexes available for the other input. As opposed to that, it is a *non-blocking* operator based on SHJ, and is a hybrid one in that it employs both *push- and pull-based* mechanisms.

Query Processing based on SIHJ. Typically, a tree-shaped query plan is employed to determine the order of execution. Using the standard *SHJ* operator, the execution is pull-based in that starting from the root operator, higher-level

operators in the plan invoke (the `next` method of) lower-level operators to obtain their inputs. Instead of pulling from the root, the *push-based SHJ* [6] allows inputs to be pushed to higher level operators (by invoking their `push` methods). The push-based SHJ maintains two hash tables, one for each input. Incoming tuples on either input are first inserted into their respective hash table and then used to probe the other hash table to find valid join combinations, which finally are pushed to subsequent operators. Thereby, results can be produced as soon as input tuples arrive. Without local data, SIHJ is essentially a push-based SHJ. Otherwise, it combines pull and push, i.e., while processing tuples that have been pushed to either one of its inputs, it also supports pulling local data from the index available for one of its inputs using data of the other input.

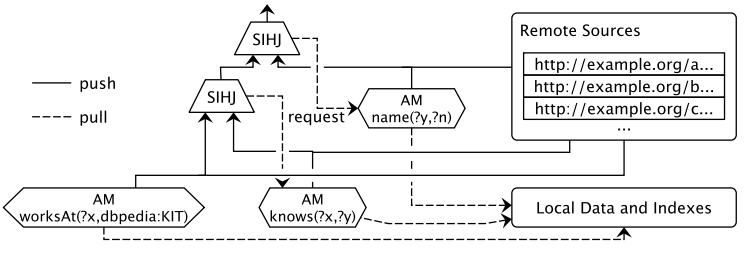

Fig. 1. Query plan with SIHJ operators and access modules (AM)

For a query with three triple patterns, Fig. 1 shows a left-deep query plan consisting of SIHJ operators and access modules for loading data. In a left-deep plan, the left input of all join operators is connected to the output of a join operator lower in the query plan, while the right input is connected to data sources, which in our case, might comprise both remote and local data. The exception is the lowest join operator, whose left input is not connected to another join operator but to data sources. Data arriving from remote sources are retrieved by a dedicated retrieval thread [6] and their data is pushed directly into the corresponding operators, whereas the access modules pull data from local indexes on request and then push them into the join operators.

Algorithm. In particular, we designate the left input of SIHJ as the "driving" input. All bindings that arrive on the left are used to perform lookups on local data to load only bindings into the right input that produce join results. This is achieved by instantiating the triple pattern on the right input with bindings for the join variable obtained from the left input:

Definition 3. *Let t_i^q, t_j^q be two triple patterns of T^q, v the join variable shared by t_i^q and t_j^q and $\Omega_{t_i^q}$ be the set of bindings for t_i^q. The results of the join of t_i^q and t_j^q on v is then calculated as $\Omega_{t_i^q} \bowtie_v \Omega_{t_j^q}$, where $\Omega_{t_j^q} = \bigcup_{u \in \Omega_{t_i^q}(v)} \{b | b \in \Omega_{t_j^q(v,u)}\}$, where $t_j^q(v,u)$ is an instantiated triple pattern obtained by substituting constant u for variable v.*

For local data we use separate *access modules* [11] that encapsulate access to local indexes. The `load` method for the AM is specified in Alg. 1. For every

SIHJ operator, one access module is created and connected to its right input. The access module accepts requests from the join operator *in* for loading the bindings Ω_{t^q} from triples matching a triple pattern t^q using the index I (line 1). All access to local storage is executed asynchronously by the access module so that operations in other parts of the query plan can still progress. Bindings loaded by an access module are pushed into its join operator (line 2).

Algorithm 1. AM: $load(in, t^q)$

Input: Operator *in*, which requests data inputs for pattern t^q

```
1 Ω_{t^q} = I.lookup(t^q);                          // lookup in local index
2 foreach b ∈ Ω_{t^q} do in.push(this,b);     // push bindings to join operator
```

Algorithm 2. SHJ: $push(in, b)$

Input: Operator *in* from which input binding b was pushed

Data: Hash tables H_i and H_j; current operator *this*; subsequent operator *out*; join variable v; t_i^q is the left and t_j^q the right triple pattern

```
1 if in is left input then
2     if b(v) ∉ H_i then AM.load(this,t_j^q(v, b(v)))
3     H_i[b(v)] ← H_i[b(v)] ∪ b
4     J ← H_j[b(v)]
5 else
6     H_j[b(v)] ← H_j[b(v)] ∪ b
7     J ← H_i[b(v)]
8 forall j ∈ J do out.push(this,merge(j,b))
```

This use of local data via the access module is shown in Alg. 2. In particular, loading from the index results in a new triple that is pushed to the SIHJ operator (right input, line 4). All inputs of the "driving" left input are also pushed into this operator. When a binding b arrives on the left input, the corresponding hash table H_i is first probed to determine if it already contains the binding $b(v)$ for the join variable v captured by b (line 2). If this is not the case, i.e., this binding has not been processed before, a request to load triples from the local index using the instantiated triple pattern $t_j^q(v, u)$ is sent to the access module (line 2). Then, b is inserted into the corresponding hash table H_i and H_j of the right input is probed to obtain valid join combinations (line 3 - 4), which are then pushed to operator *out* (line 8). Bindings arriving on the right input (i.e., from remote sources or those pushed from the AM) are processed in a similar manner, except that no requests are sent to the access module (line 6 - 7), which is not necessary as all bindings are stored in hash table H_j and are therefore available when a matching input arrives on the left input.

Note that bindings on the right or left input may be both local or remote data. Both remote and local data may be pushed into the left input. Remote data may also be pushed into the right input, and through explicit pulling using the AM (line 4), this input might also contain local data.

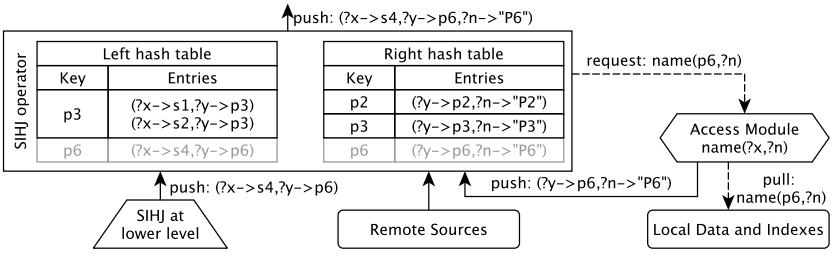

Fig. 2. Processing of the SIHJ operator for data coming from the left input

Fig. 2 illustrates the operation of a SIHJ operator. An input containing bindings for two variables $?x, ?y$ is received and then inserted into the left hash table. Then a request for $\langle p6, name, ?n\rangle$ is sent to the access module. After loading the data from the local index, the binding for $?y, ?n$ is inserted into the right hash table. In combination with the binding in the left hash table a join result is finally created and pushed to the subsequent operator.

Cost Model. We use a unit-time-basis cost model that captures the operator cost in terms of the tuples that are accessed and the cost of the physical operations needed [4]. All costs are defined in an abstract manner, independent from the concrete implementation and data structures being used.

The cost of a SIHJ with two inputs A and B is the sum of three components: the cost for joining tuples arriving on the left and the right input and the cost of the access module: $C_{A \bowtie B} = C_{A \rtimes B} + C_{A \ltimes B} + C_{AM}$

The operation carried out for tuples on the left input are: insertion into hash table for A, probing of hash table for B, creating join results and finally, sending a request to the access module. Accordingly, the cost $C_{A \rtimes B}$ is defined as follows:

$$C_{A \rtimes B} = |A|(I_h + P_h + \varphi \cdot |B| \cdot J \cdot \frac{|A|}{|A| + |B|} + R)$$

with: weight factors I_h, P_h for hash table insert and probe; join selectivityφ; weight factor J for creating result tuples; weight factor R for request to access module; the fraction $\frac{|A|}{|A|+|B|}$ of inputs arriving on the left input

The term $I_h + P_h$ represents the cost of inserting an incoming tuple and then probing the other hash table. Given a join selectivity φ, the number of results for $A \bowtie B$ is $\varphi|A||B|$. Multiplied by the weight factor for creating results, this yields the term $J \cdot \varphi \cdot |A||B|$. Further, it is multiplied with $\frac{|A|}{|A|+|B|}$ to consider join cost only for tuples that actually arrive in A. For each tuple in A, a request is sent to the access module, whose cost is captured by R.

The cost $C_{A \ltimes B}$ for the other input is defined in a similar fashion, except that no requests to the access module are needed:

$$C_{A \ltimes B} = |B|(I_h + P_h + J \cdot |A| \cdot \varphi \cdot \frac{|B|}{|A| + |B|})$$

The cost C_{AM} for the access module is defined as $C_{AM} = |A| \cdot P_l + |B_l| \cdot L_l$, where the input B is split into tuples from remote sources B_r and local tuples

loaded from disk B_l (i.e., $B = B_r \cup B_l$ and $B_r \cap B_l = \emptyset$).The cost for probing the local index, which has to be done for all tuples arriving in A, is represented by $|A| \cdot P_l$. When matching tuples are found, they have to be loaded from disk, the cost of which is given by $|B_l| \cdot L_l$.

Using the Cost Model for Query Optimization. The cost model developed in the previous section abstracts from concrete implementations and hardware by using weight factors. To use the cost model for query optimization these weight factors have to be known. The weight factors can be determined by running the operator on known input and then measuring the CPU time of the operations represented by the individual weight factors. Note that the weight factors are dependent on the characteristics of the data being used, in particular on the input size (both remote and local) and join selectivity. For example, the higher the join selectivity, the higher the relative weight of join result creation. Thus – as always the case of query optimization in practice – weight factors shall be derived from the underlying data.

In particular, measurements shall be taken for different combinations of input size and join selectivity. These measurements shall aim at covering a large space of possible combinations. At query compile-time, the weight factors precomputed for the combination that best fit the input size and join selectivity estimated for the given query are used to estimate join operator cost.

Batching. When an access module receives a request for loading data matching an instantiated triple pattern from local storage, all matching triples will have the same binding for the join variable because it has been used to instantiate the triple pattern in the first place. Sending each binding one by one to the join operator will incur an unnecessary overhead because they all will be inserted into the same hash bucket; and subsequently, the same hash bucket has to be probed several times when using these bindings. It is therefore beneficial to process data loaded from local indexes in *batches*, where the hash tables of the join operator are accessed only once for a batch of bindings.

4 Comparison to Non-blocking Iterator

In [3], NBIJ was proposed to deal with high network latency in the Linked Data context and the resulting issue of blocking. We now study this operator, extending previous work [3] with a completeness analysis and cost model.

Query Processing based on NBIJ. NBIJ is based on a traditional pull-based mechanism, i.e., each operator in the query plan has a `next` method that is called by operators higher in the query plan tree. It is also used in left-deep plans, where all inputs consist only of data from remote sources.

During query processing an in-memory list G of data sources is maintained. Each downloaded source is indexed and then added separately to G. When the next method receives a result from a lower operator on the left input, first the following requirement is checked:

Requirement 1. *Let t_i^q, t_j^q be two triple patterns of T^q, v the join variable shared by t_i^q and t_j^q and $b \in \Omega_{t_i^q}(v)$ a binding received on the left input.*

Then b can only be further processed if the following condition holds: $\forall u \in \{s(t_j^q(v, b(v))), p(t_j^q(v, b(v))), o(t_j^q(v, b(v)))\}$: *if u is an URI then* $ID(u) \in G$.

This requirement ensures that all sources identified by URIs in the instantiated triple pattern have been retrieved and added to the list of in-memory sources. If the requirement is not fulfilled, the sources are marked for asynchronous retrieval, the binding is *rejected* by calling the `reject` method of the lower join operator, and the operator calls `next` again to retrieve further inputs. Otherwise, all sources in G are successively queried for the instantiated triple pattern $t_j^q(v, b(v))$ using in-memory indexes to construct join results.

When the `reject` method of a NBIJ operator is called, the rejected binding is added to a separate list maintained by the operator. On subsequent calls to its `next` method, the operator randomly decides between returning a previously rejected binding from the list or a new one. The rejection mechanism ensures that query processing can proceed even when sources for a particular pattern are not yet available.

Completeness. A disadvantage of NBIJ is that the obtained results are not necessarily complete w.r.t. downloaded data, i.e., it is not guaranteed that all possible results that can be derived from downloaded data are actually computed [3]. While Requirement 1 does ensure that all sources mentioned in an instantiated triple pattern are retrieved before processing the pattern, it is possible that data matching that pattern is contained in other sources *retrieved later* during query processing. This is possible because Linked Data sources can contain arbitrary data and therefore not all data matching a particular triple pattern is necessarily contained in the sources mentioned in the pattern. As the NBIJ works in a pull-based fashion (and not push-based), this data will be disregarded if it is never requested again.

In contrast, a query plan based on SIHJ operators is guaranteed to produce all results. Requirement 1 is not necessary, because the operation of the SIHJ operator is completely symmetrical and push-based, i.e., incoming data can arrive on both inputs and in any order and its operation is driven by the incoming data instead of the final results. When an input tuple arrives on either of its input, the SIHJ operator is able to produce all join results of that tuple with *all previously seen inputs*, because these are kept track of in the hash table of the SIHJ operator. This ensures that it does not matter at which point during query processing a particular input for a triple pattern arrives, the final result is always complete with respect to the data in the sources that were retrieved.

Cost Model. Since the randomness of the rejection mechanism cannot be accurately captured in a cost model, we simply assume that all incoming bindings on the left input are first rejected and then processed on the second try. The cost for the NBIJ operator can then be calculated as follows:

$$C_{A \bowtie_{NBIJ} B} = |A|(P_G + T + |G| \cdot L) + \varphi|A||B| \cdot J$$

with: weight factor P_G for checking Req. 1; number of sources $|G|$; weight L for probing in-memory graph; weight T for tracking rejected bindings

The term P_G gives the cost for checking whether the corresponding sources for a binding have been retrieved. The cost for rejecting a binding is T. Both these operations are performed for all bindings of the left input. For each bindinge from A all available graphs (in the worst case all sources) are consecutively probed for join combinations, yielding the term $|A||G| \cdot L$.

We now compare the cost models of the SIHJ and NBIJ operators. As the NBIJ operator only operates on remote data, we disregard the costs of SIHJ for requests sent to the access module. The cost of SIHJ is then:

$$C_{A \bowtie_{SIHJ} B} = |A|(I_h + P_h) + |B|(I_h + P_h) + \varphi|A||B| \cdot J$$

Assuming that both operators operate on the same inputs (i.e., we disregard the completeness issue discussed earlier), the results produced by both operators are the same and therefore the cost $\varphi|A||B| \cdot J$ for creating results is the same. The SIHJ might incur higher overhead for maintenance of its hash tables as all incoming tuples require insertion into and probing of a hash table. Compared to this, NBIJ incurs cost for checking the requirement, rejecting bindings and maintaining rejected bindings. However, NBIJ further incurs cost for probing all in-memory sources $|A||G| \cdot L$, which depends on the number of available sources. That means that the more sources are retrieved during processing, the higher the cost of the operator, whereas the SIHJ operator incurs no such cost and is independent from the number of retrieved sources.

5 Related Work

Previous work on Linked Data query processing [3,2,6] was discussed throughout the paper. Here, we discuss related database research.

Join Operators. In the database community a lot of research has been done on join operators that can produce results as soon as inputs become available without blocking and are therefore suited to high latency environments and stream processing. The symmetric hash join [15] was the first of a new generation of such operators. To deal with the high memory requirements of the SHJ, the XJoin operator [13] flush tuples to disk if memory becomes scarce (during the arriving phase). During a reactive phase, when inputs are blocked, XJoin uses previously flushed tuples to produce further join results. During the final cleanup phase after all inputs have been consumed, the XJoin operator joins the remaining tuples that were missed during the previous phases. An important observation is that the output rate is heavily influenced by which tuples are flushed to disk, as some tuples might produce more results than others. This lead to the introduction and subsequent improvement of a flushing policy [9,12,1].

The SIHJ operator proposed in this work is also based on the symmetric hash join. The memory consumption of the SIHJ could be addressed using concepts proposed for the XJoin; but this topic was not the focus of this work. Similar to XJoin, SIHJ does access locally stored data, but the purpose is different: SIHJ treats local data as an additional data source whereas XJoin and the mentioned work based on it use the disk as a cache and focus on the problem of how to use it for tuple storage when memory becomes scarce.

Adaptive Query Processing. Access Modules [11] were proposed to be used in conjunction with an Eddy to provide different data access methods (scan, index) and switch between them at run-time. Probe tuples are sent from the Eddy to the access module to request a particular subset of the data. The access module then pushes the data into the Eddy, marking the end with a special tuple. In our work we adopt the notion of an Access Module to provide access to local indexes in an asynchronous fashion.

Stream Databases. Fjords [8] support push- and pull-based operators and combine push-based stream processing with pull-based processing. Fjords provide a bounded queue between operators that buffers tuples between two operators so that push- and pull-based operators can be used in the same query plan. Because the queues are bounded, tuples may have to be discarded. The SIHJ operator also uses push- and pull- based processing, but in a single operator.

In all, some concepts underlying SIHJ overlap with ideas from related database work. However, there is no single operator that can be used for remote and local data where the latter is not considered as cache but an additional independent source – especially in the Linked Data setting. SIHJ fills this gap and presents a means to incorporate local data into Linked Data query processing.

6 Evaluation

The evaluation consists of two parts: first, we use real-world datasets to compare SIHJ with NBIJ; second, we create several synthetic datasets with different characteristics to study the performance based on the proposed cost models. We present a summary and refer to the technical report [7] for more details.

6.1 Overall Performance

Setting. In this part, we first show the benefits of stream-based query processing in comparison to non-blocking iterators. We compare an SHJ-based implementation (*SQ*) with the implementation of the NBIJ-based query processing (*NBI*) in SQUIN[1]. Both systems do not use local data and run without query optimization, and thus are comparable. Second, we compare three implementations of stream-based query processing over local and remote data to study the push- and pull-based mechanism. One is the baseline, which is a configuration of SIHJ that does not pull from the local data indexes but simply pushes all relevant data into the query plan (*SQ-L*), i.e., this corresponds to the basic solution described in Section 2. This is compared with the configuration using indexes as proposed in this work, where *SQ-I* ran without and *SQ-IB* ran with batching.

All experiments were run on a server with two Intel Xeon 2.8GHz Dual-Core CPUs and 8GB of main memory. SQUIN is a Java implementation of NBIJ, whereas the SQ systems are implemented in Scala. Both systems employ multithreading and were configured to use five threads to retrieve sources.

Dataset. The data consists of several popular Linked Data datasets, among them DBpedia, Geonames, New York Times, Semantic Web Dog Food and several life

[1] `http://www.squin.org`

science datasets. In total, the data consists of ca. 166 million triples. For the experiments with approaches using local data, the dataset was split into remote and local data, where the randomly chosen local data accounted for 10% of the total dataset. Remote data were deployed on a CumulusRDF[2] Linked Data server on the local network so that data can be accessed using URI lookup, whereas local data were indexed using our triple store [14].

Queries. We created 10 BGP queries that cover different complexities w.r.t. query size and the number of sources retrieved during query processing. For example, Q1 retrieves the names of authors of demo papers at ISWC 2008:

```
SELECT * WHERE { ?p sw:isPartOf <http://data.semanticweb.org
      /conference/iswc/2008/poster_demo_proceedings> .
  ?p swrc:author ?a .  ?a rdfs:label ?n . }
```

Results. Fig. 3a shows query times of the SQ and NBI systems for all ten queries. The SIHJ-based system was faster for all queries, in some cases up to an order of magnitude. On average, queries took 9699.18ms for SQ and 41704.27ms for NBI, corresponding to an improvement of 77%.

Query times for SQ-I, SQ-IB and SQ-L are presented in Fig 3b. In all cases, SQ-I and SQ-IB outperformed SQ-L and also here, improvements were up to an order of magnitude in some cases. Note that for Q8, SQ-L ran out of memory because the amount of local data to be loaded was too large. On average, query times were 9366.39ms for SQ-IB, 9396.18ms for SQ-I and 28448.7.98ms for SQ-L. This yielded an improvement of 67% of SQ-IB over SQ-L, clearly showing that using locally available indexes is beneficial. It reduced the amount of data that is loaded from disk, especially for queries with less selective triple patterns. The improvement achieved through batching could also be observed, and will be examined in more detail in the next section.

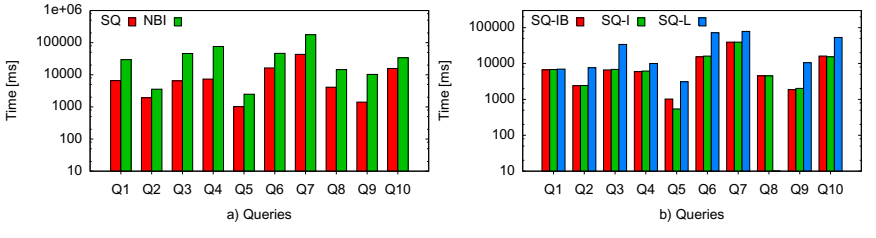

Fig. 3. Overall query times for a) SQ and NBI and b) SQ-IB, SQ-I and SQ-L

6.2 Join Operator Performance in Detail

Setting. Previously, the operators were incorporated into plans for processing entire BGP queries. Here, we focus on *join processing* using SIHJ and NBIJ. Synthetic datasets that have known characteristics are used to examine the performance of these operators in detail. We evaluated three SIHJ-approaches:

[2] `http://code.google.com/p/cumulusrdf/`

SQ-IB, SQ-I and SQ. For the NBIJ operator, we used our own implementation in order to instrument the code with detailed measurement points.

Datasets. The synthetic datasets for these experiments consist of separate sets of triples for the left and right input. The right input is split into local and remote parts, where the remote part is distributed among a number of sources. Here, we want to focus on the weight factors of the cost models and therefore keep "remote" data in memory and push it into the operator, instead of performing network access, which might lead to inconsistencies in the performance measurements. The data were generated with the following parameters: the size of the left and right input is given by a, b, respectively; ρ is the fraction of the right input that is local data; φ is the join selectivity; the number of sources for the remote part of the right input is s (the source sizes follow a normal distribution). We create several sets of datasets, where one of the parameters is changed while the others are fixed in order to examine the influence of each parameter.

Results. We examine the parameters' effect on the weights of the cost model:

Join Selectivity. Fig. 4a shows the influence of join selectivity on the different weight factors of the SIHJ cost model in terms of their relative fraction of total time measured for SQ. For joins with high selectivity ($\varphi = 0.0005$), i.e., only a small number of input tuples match other tuples to form join results, loading of local data took the largest part of total processing time. For low selectivity joins ($\varphi = 0.5$), the creation of result tuples dominated query processing. Using the cost model, this can be explained by the observation that join selectivity has impact only on the term $J \cdot \varphi |A||B|$, meaning that only the weight of result creation increases with lower join selectivities.

Number of Sources. Fig. 4b shows processing times for various number of sources $|G|$ for SQ and NBI. Overall, times for SQ were largely the same for all source counts, whereas the times for NBI increased with larger a number of sources. The times for NBI were split into times for checking the requirement (cr), loading data from the in-memory graphs (load) and creating result tuples (join). Clearly, results show that both cr and load times were dependent on $|G|$. This is accounted for by the cost model, i.e., the term $|A|(P_G + |G| \cdot L)$ indicates that cost depends on P_G and $|G|$. Join times were the same because the number of results does not change with $|G|$.

Input Size. Fig. 4c presents the effect of input size (on the right input) on processing times of SQ-I and SQ-IB. We can see that for larger inputs, the relative time spent on loading local data decreased and the relative weights of hash table insertion and increased. This is probably due to the larger hash tables that were required for larger input sizes, introducing more overhead for rehashing when the hash tables need to be expanded.

Local Data Fraction. We examined processing times for various local data sizes. The overall number of inputs on the right input was the same, only the ratio between remote and local data changed. A value of $\rho = 0.1$ means 10% of the data is local data. Fig. 4d shows processing times for SQ-I and SQ-IB. With higher local data fractions, the impact of loading on total processing times is more pronounced. Whereas for a local fraction of 0.1 loading accounted for

about 22% of total time, at 0.8 it accounted for over 60%. This is because with more local data, more effort was spent on using the local indexes to find triples that produce join results. Thus, less effort was needed for join, probe as well as insert. Note that as remote data were actually in-memory data, access to local data was slower than for "remote" data. Thus, loading here essentially means loading local data. In the standard setting, network access is usually slower than disk access. This means that loading would have an even larger impact.

This experiment also shows the benefits of batching, which are more pronounced for larger amounts of local data, as reflected by the smaller amounts of time spent on inserting and probing hash tables.

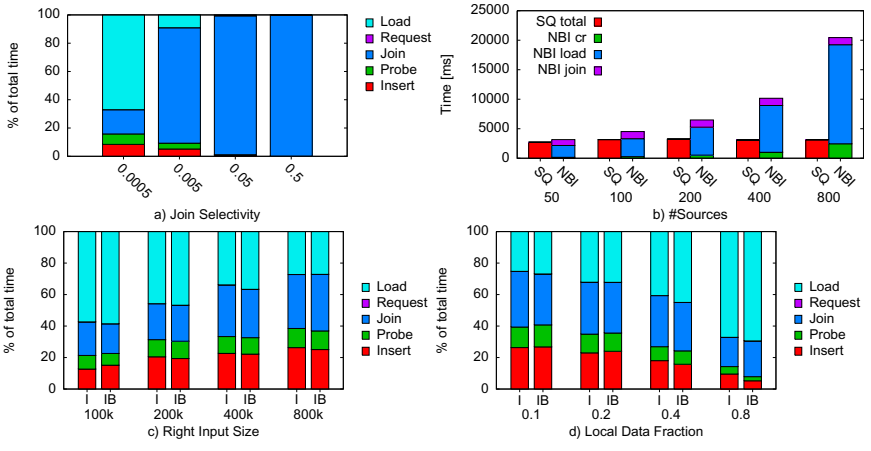

Fig. 4. a) Join selectivity ($b = 10000, \rho = 0.2, s = 200$), b) Number of sources ($b = 500000, \varphi = 0.0002, \rho = 0.2$), c) Input size ($\varphi = 0.2, \rho = 0.2, s = 200$), d) Local part ($b = 500000, \varphi = 0.0002, s = 200$) ($a = 10000$ for all)

7 Conclusion

We propose a new operator, the Symmetric Index Hash Join (SIHJ) for processing queries over local and remote Linked Data in a stream-based and non-blocking fashion. We provide cost models for SIHJ and the Non-Blocking Iterator (NBIJ) previously proposed for dealing with remote Linked Data. A detailed comparison shows that while SIHJ might have larger overhead for accessing its hash tables, its cost does not depend on the number of data sources processed. The number of sources however has a large impact on the performance of NBIJ. Further, as opposed to NBIJ, SIHJ guarantees complete results w.r.t. the data retrieved during query processing. We performed an evaluation of both operators on a real-world dataset and several synthetic datasets. We show that stream-based query processing using push-based SHJ performs on average 77% better than NBIJ-based query processing w.r.t. to overall query execution time. The experiments show that using available indexes to access local data is beneficial, resulting in an average improvement of 67% compared to a baseline that

simply loads all data matching query triple patterns. Detailed analyses using the synthetic datasets further shed light on the weights of the proposed cost models.

Acknowledgements. The authors would like to thank Olaf Hartig for providing an updated implementation of SQUIN and Andreas Harth for discussions. Research reported in this paper was supported by the German Federal Ministry of Education and Research (BMBF) in the CollabCloud project (01IS0937A-E).

References

1. Bornea, M., Vassalos, V., Kotidis, Y., Deligiannakis, A.: Double index NEsted-Loop reactive join for result rate optimization. In: IEEE 25th International Conference on Data Engineering, ICDE 2009, pp. 481–492 (2009)
2. Harth, A., Hose, K., Karnstedt, M., Polleres, A., Sattler, K., Umbrich, J.: Data Summaries for On-Demand Queries over Linked Data. In: Proceedings of the 19th International Conference on World Wide Web, pp. 411–420. ACM, New York (2010)
3. Hartig, O., Bizer, C., Freytag, J.-C.: Executing SPARQL queries over the web of linked data. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 293–309. Springer, Heidelberg (2009)
4. Jeffrey, J.K., Naughton, J.F., Viglas, S.D.: Evaluating window joins over unbounded streams. In: ICDE, pp. 341—352 (2003)
5. Klyne, G., Carroll, J.J., McBride, B.: Resource description framework (RDF): concepts and abstract syntax (2004)
6. Ladwig, G., Tran, T.: Linked data query processing strategies. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 453–469. Springer, Heidelberg (2010)
7. Ladwig, G., Tran, T.: SIHJoin: Querying Remote and Local Linked Data. Technical report (2010), `http://people.aifb.kit.edu/gla/tr/sq_report.pdf`
8. Madden, S., Franklin, M.J.: Fjording the stream: An architecture for queries over streaming sensor data. In: International Conference on Data Engineering, p. 0555. IEEE Computer Society, Los Alamitos (2002)
9. Mokbel, M., Lu, M., Aref, W.: Hash-merge join: a non-blocking join algorithm for producing fast and early join results. In: Proceedings of 20th International Conference on Data Engineering, pp. 251–262 (2004)
10. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation (2008)
11. Raman, V., Deshpande, A., Hellerstein, J.: Using state modules for adaptive query processing. In: Proceedings of 19th International Conference on Data Engineering, pp. 353–364 (2003)
12. Tao, Y., Yiu, M.L., Papadias, D., Hadjieleftheriou, M., Mamoulis, N.: Rpj: Producing fast join results on streams through rate-based optimization. In: SIGMOD Conference, pp. 371–382 (2005)
13. University, T.U., Urhan, T., Franklin, M.J.: XJoin: a Reactively-Scheduled Pipelined Join Operator (2000)
14. Wang, H., Liu, Q., Penin, T., Fu, L., Zhang, L., Tran, T., Yu, Y., Pan, Y.: Semplore: A scalable ir approach to search the web of data. J. Web Sem. 7(3), 177–188 (2009)
15. Wilschut, A.N., Apers, P.M.G.: Dataflow query execution in a parallel main-memory environment. Distributed and Parallel Databases 1(1), 103–128 (1993)