A SPARQL Engine for Streaming RDF Data

Sven Groppe, Jinghua Groppe, Dirk Kukulenz and Volker Linnemann
Institute of Information Systems (IFIS), University of Lübeck, Lübeck, Germany
{groppe, jinghua.groppe, kukulenz, linnemann} @ ifis.uni-luebeck.de

Abstract

The basic data format of the Semantic Web is RDF. SPARQL, which has been developed by the W3C, is the upcoming standard for RDF query languages. Typical engines for processing SPARQL queries on RDF data first read all RDF data, may build indices of the complete read data and afterwards evaluate SPARQL queries. Such engines cannot operate on streaming RDF data. Streaming query engines operating on streams of data can (a) discard irrelevant input as early as possible, and thus save processing costs and space costs, (b) build indices only on those parts of the data, which are needed for the evaluation of the query, and (c) determine partial results of a query as early as possible, and thus evaluate queries more efficiently. We propose such a streaming SPARQL engine, which is the first streaming SPARQL engine to the best of our knowledge.

1. Introduction

The Semantic Web [21] uses the Resource Description Framework (RDF) [5] as its basic data format, which aims to represent information about resources. The W3C has developed SPARQL [31] as a query language for RDF data. SPARQL recently became a W3C Candidate Recommendation, which is one step before a W3C Recommendation. Furthermore, in recent years RDF storage systems, which support or plan to support SPARQL, have been developed like e.g. Jena [32], Sesame [4], rdfDB [17], Redland [3], Kowari [27], RDF Suite [1] and Allegro [13]. Thus, SPARQL is increasingly important as the upcoming standard for RDF query languages.

SPARQL [31] supports querying by triple patterns, conjunctions, disjunctions, and optional patterns, and constraining queries by source RDF graph and extensible value testing. Results of SPARQL queries can be ordered, limited and offset in number.

RDF data consists of a set of triples of the form (s, p, o), where s is called the subject, p is called the predicate and o is called the object of the triple. The triples of *RDF data streams* arrive continuously and are processed using a single sequential scan.

Current SPARQL engines (e.g. [32], [4], [17], [3], [27], [1] and [13]) do not operate on streams, i.e. these engines first read all the RDF data, may build indices of all RDF data and afterwards evaluate SPARQL queries on the read RDF data.

Use of streams, as opposed to querying against materialized RDF data, minimizes the runtime memory footprint of the query engine. At runtime, each runtime operator consumes as its input one triple or environment with bindings of variables at a time, and any input data not required is eventually discarded. Furthermore, only those indices are built of only useful parts of the input data (and not of all input data) at run-time, which optimize the evaluation of the given SPARQL query.

Stream-based processing enables more efficient evaluation not only in local scenarios, where the data is stored and the SPARQL evaluator runs on the same computer, but also

- to integrate data over networks like the Internet, in particular from slow sources, it is desirable to progressively process the input before all the data is retrieved,
- for selective dissemination of information, where RDF data has to be filtered according to requirements given in a SPARQL query before being distributed to subscribers, and
- for pipelined processing, where the input is sent through a chain of processors the input of each of which is the output of the preceding processor.

We propose a streaming SPARQL engine, which evaluates SPARQL queries on streams of RDF data, and can discard irrelevant input as soon as possible, build indices only on the data needed for the evaluation of the query, and determine partial results of a query as soon as they are available, in order to save processing costs and space costs and to enable more efficient processing of queries.

The contributions of our paper are

- a well-defined formal streaming SPARQL algebra, such that the correctness of following optimization steps can be checked,
- the transformation from a SPARQL query into an operator graph of the proposed streaming SPARQL algebra, such that we concentrate on

core functionalities of SPARQL and abstract from equivalent language constructs,

- a logical and physical optimization framework for the operations in the operator graph in order to speed up the execution times,
- a streaming SPARQL engine, which computes results as soon as possible, i.e. partial results are determined even before all the data is read, and
- a performance analysis, which demonstrates that stream-based SPARQL processing is approximately 1.8 times faster than using the non-streaming SPARQL evaluator Jena for our used benchmark queries.

2. RDF and SPARQL

We briefly introduce RDF and SPARQL in this section. The RDF data of Figure 1 contains 13 triples, the first of which (in line (2)) associates the subject pub:book1 with the predicate pub:title and the object "Twenty Two". In this paper, we do not resolve the prefix pub, which is defined in line (1) of Figure 1, due to simplicity of presentation.

```
(1)  @prefix pub:  <http://uni-luebeck.de/publications#> .
(2)  pub:book1  pub:title  "Twenty Two" .
(3)  pub:book1  pub:price  22 .
(4)  pub:book1  pub:cite   pub:book2 .
(5)  pub:book1  pub:author "The Unknown" .
(6)  pub:book1  pub:rate   6.0 .
(7)  pub:book1  pub:shop   <http://www.my-bookshop.com/> .
(8)  pub:book2  pub:title  "Fourty Two" .
(9)  pub:book2  pub:price  42 .
(10) pub:book2  pub:cite   pub:book1 .
(11) pub:book2  pub:author "The Unknown" .
(12) pub:book3  pub:title  "The Beginners" .
(13) pub:book3  pub:price  23 .
(14) pub:book3  pub:author "Adam Adams" .
```

Figure 1. RDF data

```
(1) PREFIX  pub:  <http://uni-luebeck.de/publications#>
(2) SELECT DISTINCT ?title ?author ?shop
(3) WHERE  {
(4)   ?x pub:cite ?y .
(5)   ?x pub:author ?author .
(6)   ?y pub:author ?author .
(7)   ?x pub:title ?title .
(8)   OPTIONAL {
(9)       ?x pub:price ?price .
(10)      ?x pub:rate  ?rate .
(11)      FILTER (?price<30 && ?rate>5) .
(12)      ?x pub:shop ?shop .              } }
```

Figure 2. A SPARQL query Q

Figure 2 shows a SPARQL query Q, which returns the titles and authors of books containing a self-citation, i.e. a citation to another book with a common author. Furthermore, the query Q returns additionally a shop for cheap books (price is below 30) with a good rate (above 5). The query starts with the PREFIX clause with declaration of the prefix pub in line (1). The SELECT clause in line (2) identifies the variables ?title, ?author and ?shop to appear in the query results. In general, the result of a SPARQL query is a *bag* (also called *multiset*), i.e. its unordered elements can appear more than once, of *environments* (see Definition 2) consisting of *bindings* of variables (see Definition 1). The environments of the result of a SPARQL query must be distinct if the SELECT clause contains the optional DISTINCT keyword as in Figure 2, such that the result (see Figure 3) is actually a *set* of environments.

Definition 1: A *binding* of a variable is a tuple (n, v), where n represents the name of the variable and v its current value.

Definition 2: An environment E is a set of bindings of variables, where each variable of the bindings in E has exactly one assigned value, i.e. $\forall$ (n, v_1)$\in$ E: $\forall$ (n, v_2)$\in$ E: $v_1=v_2$.

{{ (title, "Twenty Two"), (author, "The Unknown"), (shop, <http://www.my-bookshop.com/>) }, { (title, "Fourty Two"), (author, "The Unknown") }}

Figure 3. Result set of environments of the SPARQL query Q of Figure 2 with input RDF data of Figure 1

The WHERE clause (see lines (3) to (12)) has four triple patterns (see lines (4) to (7)) and an OPTIONAL pattern (in lines (8) to (12)), which consists of three triple patterns (lines (9), (10) and (12)) and a FILTER expression (line (11)). The first, second and third position (e.g. ?x, pub:cite and ?y in line (4)) in the triple pattern represents the constraints or bindings to variables for the subjects, predicates and objects in the RDF data. Joins of the query are expressed by using the same variable in triple patterns. FILTER clauses (e.g. line (11)) discard those environments of bindings of variables, where the expression of the FILTER clause is evaluated to false. For example, the FILTER clause in line (11) remains those environments, where the variable price is bound to a value less than 30 and the variable rate is bound to a value greater than 5.

The constraints in OPTIONAL clauses do not have necessarily to be fulfilled. However, bindings to variables, which have to be bound in OPTIONAL clauses, do not appear in the resultant environments whenever the whole OPTIONAL pattern is not fulfilled (e.g. environment { (title, "Fourty Two"), (author, "The Unknown") }, where the binding of the variable shop is missing, in the result of Q in Figure 3).

There are further constructs e.g. built-in functions, set operations like the UNION operator, and CONSTRUCT queries to generate RDF graphs. We refer the interested reader to [31].

3. Streaming SPARQL Algebra

In the following paragraphs, we enumerate the operators of the proposed streaming SPARQL algebra. The proposed operators provide core functionalities for SPARQL processing, which abstract from SPARQL syntax and equivalent language constructs, and are the basis for logical and physical optimization. The proposed operators for the streaming SPARQL engine are event-based. For every incoming triple of the RDF data stream the Stream operator triggers an *event* at its succeeding operators, which may trigger events at their succeeding operators as well. All operators, their succeeding and preceding operators span an *operator graph*.

For each operator, we describe the actions and the resulting events of the operators after receiving an event t_i from the i-th preceding operator in pseudo code. Note that we describe the resulting event in a declarative way, i.e. we only describe what the resulting events look like and not how to compute the resulting events. We describe different implementations of the non-trivial operators in Section 5 as part of the physical optimization.

In the definitions of the operators, we describe the optional initialization after the keyword "Init", the events to receive after the keyword "Input", the actions to be done for each received event after the keyword "Actions" and the actions to be done after the RDF data stream has been closed, i.e. the RDF data stream has no incoming triples any more, after the keyword "Final" (optional) in pseudo code. While t_i represents an event from the i-th preceding operator, we use t to represent an event if there is only one preceding operand or if the events of the preceding operators do not need to be distinguished. $t_i.E$ represents an attached environment and $t_i.S$ an attached triple of the event t_i. num represents the concrete number of operands of an operator. The instruction trigger_p E triggers an event with attached environment/statement E at the operator p and trigger E triggers at all succeeding operators.

We enumerate the operators in the following items:

1. Every operator graph has a Stream operator as root node. The Stream operator receives the incoming triples of the RDF data stream and transmits these triples as events to its succeeding operators (which typically consist of one operator, the pattern matcher operator).

Operator	Stream
Input:	RDF data stream with incoming triple s
Actions:	trigger s;

2. The pattern matcher operator Match_{Pats} triggers all triple pattern operators Pats with an incoming event with attached triple. Note that implementations of the pattern matcher may choose matching triple pattern operators in a more intelligent way for speeding up processing of SPARQL queries (see Section 5.1) as part of the physical optimization.

Operator	Match_{Pats}
Input:	Event t with attached triple
Actions:	$\forall\ p \in Pats$: trigger_p t.S;

3. The triple pattern operator $\text{Pat}_{(p1,\ p2,\ p3)}$ represents a triple pattern (p1, p2, p3) (e.g. $\text{Pat}_{(?x,pub:cite,?y)}$ for line (4) of Figure 2). The resulting event of $\text{Pat}_{(p1,\ p2,\ p3)}$ is triggered when the attached triple of the received event matches the triple pattern, i.e. all literals in the triple pattern are also in the attached triple at the same position and the variables are only bound to one value. An environment is attached to the resulting event, where the variables of the triple pattern are bound to the corresponding values of the considered triple. For example, $\text{Pat}_{(?x,pub:cite,?y)}$ triggers an event with attached environment {(x, pub:book1),(y, pub:book2)} for the triple (pub:book1, pub:cite, pub:book2), but triggers no event for the triple (pub:book1,pub:price,22).

Operator	$\text{Pat}_{(p1,\ p2,\ p3)}$
Input:	Event t with attached triple $t.S=(s_1, s_2, s_3)$
Actions:	$E=\{(x,v) \mid i \in \{1,2,3\} \wedge x=p_i \wedge p_i \text{ is a variable} \wedge v=s_i\}$;
	if $((\forall\ j \in \{1, 2, 3\}: (p_j \text{ is a variable}) \vee (p_j=s_j)) \wedge$
	$\forall\ (n, v_1) \in E: \forall\ (n, v_2) \in E: v_1=v_2)$
	then trigger E;

4. The join operator Join represents a join of environments of received events of several operands (e.g. one join operator for the triple pattern operators representing the triple patterns from line (4) to (7) in Figure 2). An environment of a received event of one operand is joined with all environments of previously received events of the other operands. The Join operator triggers events with all joined environments of the received environment and all previously received environments of the other operands whenever the join condition is fulfilled. The join condition requires that variables with the same name (which are called *join partners*) are bound to the same value.

Operator	Join
Init:	$\forall\ i \in \{1,\ldots,num\}: SE_i=\{\}$;
Input:	Event t_i with attached environment, where $i \in \{1,\ldots,num\}$
Actions:	$SE_i = SE_i \cup t_i.E$;
	$\forall\ E \in \{ s_1 \cup \ldots \cup s_{num} \mid \forall\ j \in \{1,\ldots,num\}-\{i\}: s_j \in SE_j \wedge$
	$s_i = t_i.E \wedge \forall\ (n, v_1) \in s_1 \cup \ldots \cup s_{num}:$
	$\forall\ (n, v_2) \in s_1 \cup \ldots \cup s_{num}: v_1=v_2\}$:
	trigger E;

5. The filter operator $\text{Sel}_{expression}$ evaluates a filter expression expression based on the environment of the received event in order to transmit the environment to its succeeding operators or to discard this environment. For example, $\text{Sel}_{?price<30\ \&\&\ ?rate>5}$

represents the filter expression in line (11) of Figure 2.

Operator	$Sel_{expression}$
Input:	Event t with attached environment
Actions:	if(expression(t.E)) then trigger t.E;

6. The Optional operator joins the environments, which are attached to the events of its two operands (e.g. the Optional operator with two operands representing line (4) to (7) and line (9) to (12) in Figure 2), where the second operand is transformed from an optional graph pattern, and triggers the joined environments afterwards. After the RDF data stream has been closed, the Optional operator triggers the environments received from events of the first operand, which have not been joined so far with the environments from the second operand.

Operator	Optional
Init:	SE_1={}; SE_2={}; SE_{joined}={};
Input:	Event t_i with attached environment, where $i \in \{1,2\}$
Actions:	$SE_i = SE_i \cup t_i.E$;
	$\forall (e_1, e_2) \in \{ (s_1,s_2) \mid \forall j \in \{1,2\}-\{i\}: s_j \in SE_j \wedge$
	$s_i=t_i.E \wedge \forall (n,v_1) \in s_1 \cup s_2: \forall (n,v_2) \in s_1 \cup s_2: v_1=v_2\}:$
	$\{ SE_{joined} = SE_{joined} \cup e_1$; trigger $e_1 \cup e_2$; $\}$
Final:	$\forall E \in SE_1-SE_{joined}$: trigger E;

7. The projection operator $Projection_V$ excludes those bindings of an environment of a received event, which are not contained in V. For example, we use $Projection_{\{title,author,shop\}}$ for the SELECT clause (except DISTINCT for which we have an own operator) in line (2) of Figure 2.

Operator	$Projection_V$
Input:	Event t with attached environment
Actions:	trigger $\{ (n, v) \mid (n, v) \in t.E \wedge n \in V \}$;

8. The distinct operator Distinct represents the option DISTINCT in SELECT clauses (as in line (2) of Figure 2). The distinct operator triggers the succeeding operator(s) with the environments of received events, if the environment is different from environments of previously received events.

Operator	Distinct
Init:	$E_{previous}$={};
Input:	Event t with attached environment
Actions:	if($t.E \notin E_{previous}$) then { $E_{previous}=E_{previous} \cup t.E$; trigger t.E;}

9. The union operator Union triggers the unchanged environments of each received event of different operands.

Operator	Union
Input:	Event t with attached environment
Actions:	trigger t.E;

10. Every operator graph has an Output operator as leaf. The events for the output operator Output contain the results of the operator graph, such that the operator Output triggers the application with the environments of each received event in order to transmit the resultant bindings of the operator graph.

Operator	Output
Input:	Event t with attached environment
Actions:	trigger t.E;

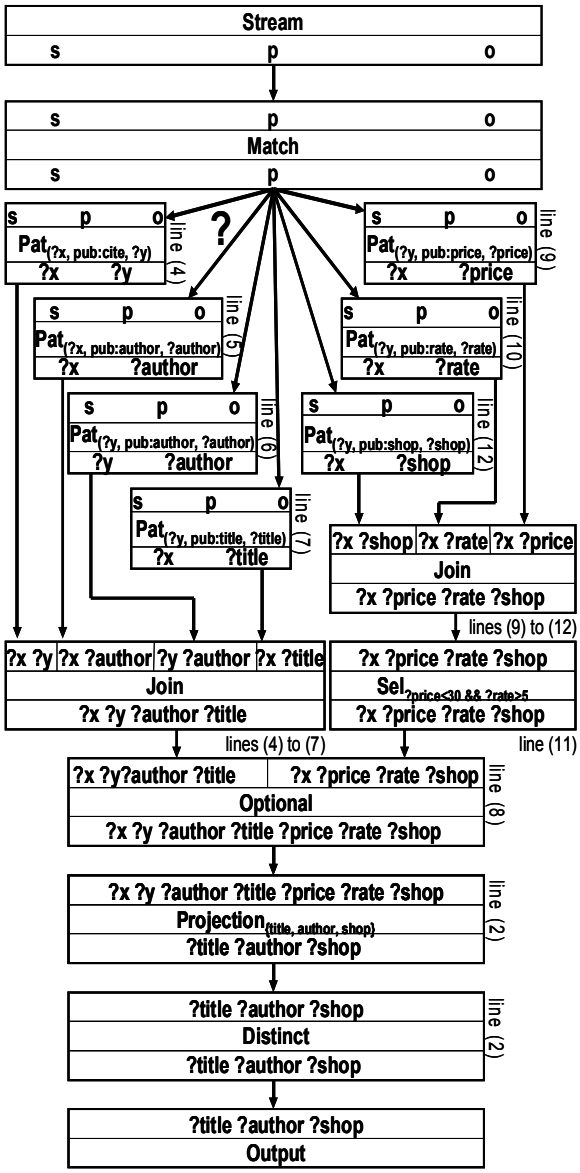

Figure 4. Operator graph of the SPARQL query of Figure 2. The line numbers refer to the corresponding subexpressions of the SPARQL query of Figure 2.

Due to space limitations, we do not represent operators for filter expressions here. Note that filter expressions only need to be evaluated to a Boolean value based on a given environment of bindings of variables for the decision of inclusion or exclusion in the result of the SPARQL query.

We require that the *operator graph of a SPARQL query* returns the same result for any input as the SPARQL query. For example, Figure 4 contains the operator graph of the SPARQL query of Figure 2. Here, the Stream operator triggers the Match operator with all received triples of the RDF data stream. The Match operator triggers those $Pat_{(p1, p2, p3)}$ operators,

which match the current triple. The $Pat_{(p1, p2, p3)}$ operators trigger their succeeding Join operator with environments of bound variables according to their triple pattern. In Figure 4, we have two Join operators, joining the results of the triple patterns outside (see lines (4) to (7)) and inside (see lines (9), (10) and (12)) the Optional construct in Figure 2. The $Sel_{?price<30 \&\& ?rate>5}$ operator represents the filter expression in line (11) of Figure 2 and constraints the result of the Join operator for the Optional construct further. The Optional operator combines the results of the joins and filter expressions outside and inside the Optional construct of Figure 2. The $Projection_{\{title,author,shop\}}$ operator projects the environments to the variables title, author and shop and the Distinct operator eliminates duplicates according to the SELECT clause of Figure 2. Finally the Output operator triggers the resultant environments of the whole SPARQL query of Figure 2.

4. Logical Optimization

Logical optimization aims to reorganize the operator graph into an equivalent operator graph, which generates the same output for any input as the original operator graph, in order to optimize the execution time of query evaluation. In addition to [28] and [30], we present two important logical optimization rules for optimizing the join order and for shifting filter upward in the operator graph.

4.1. Join Order

A join combines each environment of its operands with each other environment of its operands, i.e. let SE_i be the bag of environments of the i-th operand, then the join operator has to combine $|SE_1|*...*|SE_{num}|$ environments in the worst case. However, the result of the join is often much less than $|SE_1|*...*|SE_{num}|$ environments and changing the order of joins can speed up the evaluation. Thus we have to transform a join $Join(op_1, ..., op_n)$ with n operands $op_1, \ldots, op_n$ into binary joins with two operands, where the join order has been optimized.

The join order can be optimized by joining those joins, the result size of which is the smallest, before the other joins. Note that the result size of the joins is only known in few real-world cases before joining and thus the result sizes (or at least the relative result sizes) must be estimated. In our implementation, we evaluate those joins, where the operands have the most join partners, before the other joins. For an example, see Figure 5, where the join order of the operator graph of Figure 4 has been optimized, e.g. the bindings of ?x and ?title (with the join partner ?x) are joined at last with the result of previous joins, as the previous join has two join partners ?y and ?author.

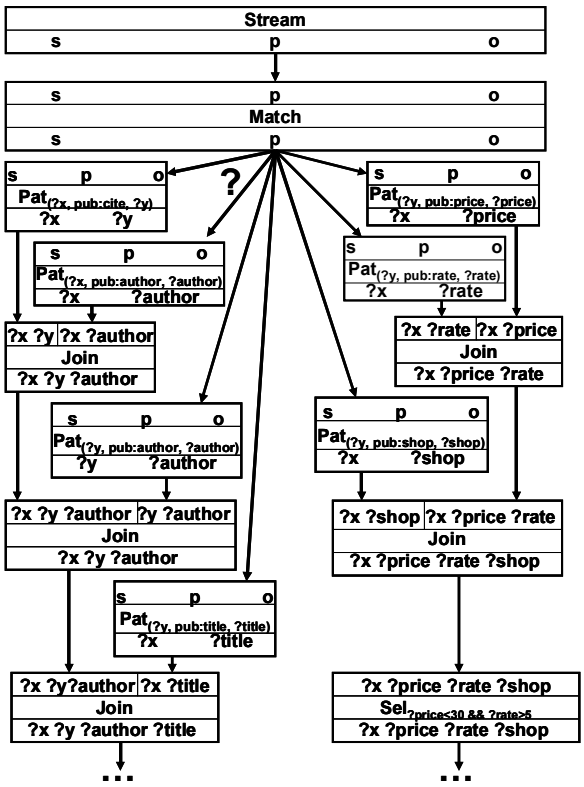

Figure 5. Operator graph of Figure 4 with optimized join order

4.2. Filter

We can push filter upward in the operator graph after those operators in the operator graph, where *all* variables of the filter expression have already been bound. This reduces the space used for intermediary results. Furthermore, $Sel_{e1 \&\& e2}(op)$ is equivalent to $Sel_{e1}(Sel_{e2}(op))$ (which is equivalent to $Sel_{e2}(Sel_{e1}(op))$, where e1 and e2 are filter expressions and op is an operand. Thus, we can split an *and*-combined filter expression e1 && e2 for pushing upward the single filters Sel_{e1} and Sel_{e2} more to the top of the operator graph. For an example, the filter has been pushed upward in Figure 6.

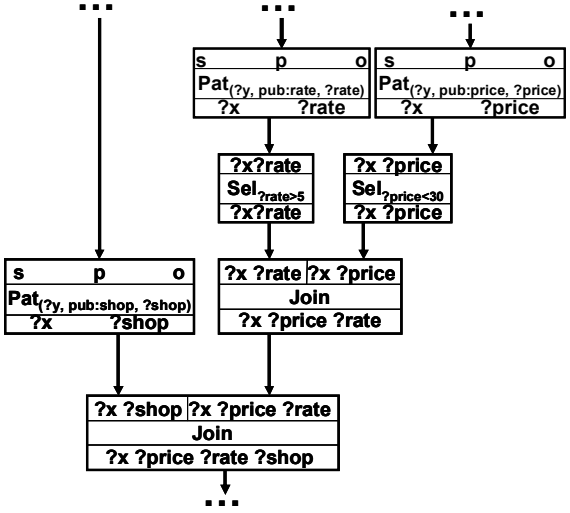

Figure 6. Operator graph of Figure 5 with pushed upward filter

5. Physical Optimization

Physical optimization aims to choose the algorithm with the best estimated execution times in the context of the operator for each operator in the operator graph. We do not present an exhaustive list of physical optimizations for the operators due to space limitation, but present optimized implementations for two important operators, for the pattern matcher and for joins.

5.1. Pattern Matcher

The *simple pattern matcher* tries for each incoming triple of the RDF data stream every triple pattern operator, which is fast whenever there are few triple patterns in the SPARQL query.

For many triple patterns in the SPARQL query, we propose to use the *hash pattern matcher*, which determines the set of matching triple patterns for an incoming triple in time linear to the length of its literals. The hash pattern matcher uses *hash maps* to access the matching triple patterns by keys. Hash maps store the matching triple pattern operators at the location of the hash value of the key, i.e. except of computing the hash value of the key, which is linear to the length of the key, the matching triple pattern operators can be accessed in constant time. The hash pattern matcher uses the sequence of literals in a triple pattern as key for the hash map, as variables in a triple pattern can be bound to any value and thus has to be neglected for building the key. For example, the triple pattern ?x pub:cite ?y has the key |<http://uni-luebeck.de/publications#cite>|, where the prefix pub has been resolved and where | represents a unique symbol to distinguish the subject, predicate and object position of the triple pattern. The used positions for the key of the triple pattern must be accordingly used to build the key of an incoming triple, e.g. for the triple pattern ?x pub:cite ?y, the key for the matching triple (pub:book2, pub:cite, pub:book1) is |<http://uni-luebeck.de/publications#cite>| and the key for the non-matching triple (pub:book3, pub:price, 23) is |<http://uni-luebeck.de/publications#price>|. As each position in a triple pattern can be a literal or a variable, there must be 2*2*2=8 different hash maps for the different types of triple patterns. For each incoming triple, the hash pattern matcher determines 8 different keys of the incoming triple for the 8 different hash maps. The hash pattern matcher then triggers all matching triple pattern operators, which are the results of looking up the 8 different hash maps using the corresponding keys.

5.2. Join Operator

The nested-loop join combines the environment from the received event of one operand with all environments of all other operands and checks the join condition, i.e. let SE_i be the bag of environments of the i-th operand, then the nested-loop join operator checks $|SE_1|*...*|SE_{num}|$ environments for joining for every possible input.

In comparison with the nested-loop join, the hash join determines the environments, which fulfill the join condition in one step. For this purpose, the hash join uses hash maps for the environments of each operand. The keys of the hash maps are the sequence of values of the join partners. For example, the key of the environment { (title, "Twenty Two"), (author, "The Unknown"), (shop, <http://www.my-bookshop.com/>) } with the join partners title and author is "Twenty Two"|"The Unknown", where | represents a unique symbol to distinguish the values of the different join partners. For each environment from the received event of one operand, the hash join builds the key and looks up the hash maps of the other operands for determining the environments to join in time linear to the length of the key.

6. Performance Analysis

We compare the average of ten execution times for full and partial results of the 14 queries of the Lehigh University Benchmark (LUBM) [18], which provides a data generator for data modelling a university.

The test system uses an Intel Pentium 4 processor with 2.66 Gigahertz, 1 Gigabytes main memory, Windows XP Professional Version 2002 and Java 1.6.

We use Jena version 2.0 [32] as the reference SPARQL processing engine since it supports the current SPARQL version [31], which is not supported by many other SPARQL processing engines. We use ARP version 2.0 [22] for parsing RDF data in our streaming SPARQL engine. We have used the parameters 1 to 5 for the data generator of LUBM [18] for generating RDF data with file sizes from 8.4 Megabytes to 52.75 Megabytes.

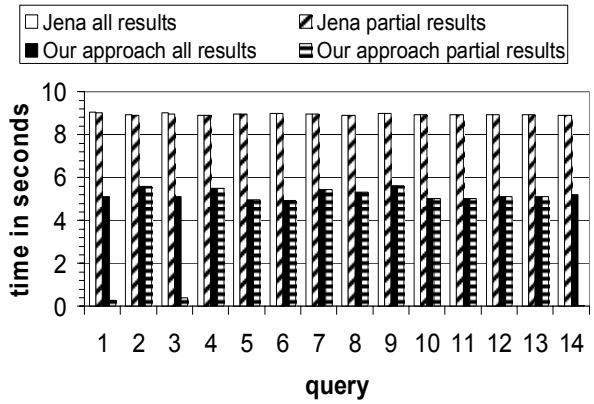

Figure 7. LUBM 1

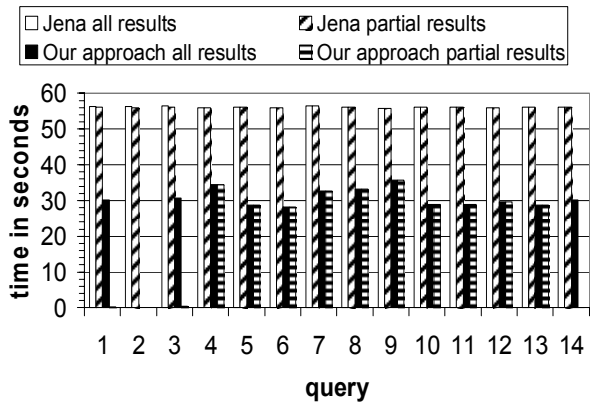

Figure 8. LUBM 5

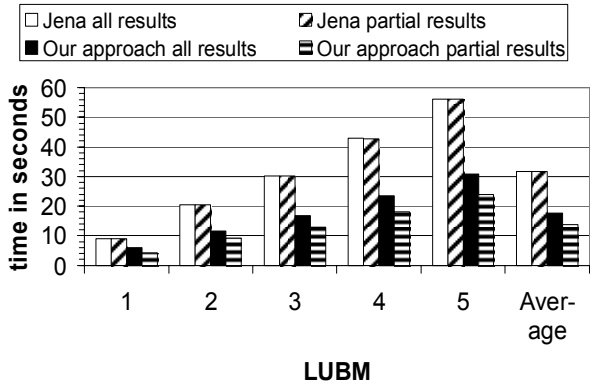

Figure 9. Average query execution times for all 14 queries of the LUBM benchmark for LUBM 1 to LUBM 5

We have similar results for all 14 queries of the LUBM benchmark (see Figure 7 for a file size of 8.4 Megabytes and see Figure 8 for a file size of 52.75 Megabytes), except that for the queries 1, 3 and 14 partial results can be computed very fast (<0.36 seconds) for our streaming SPARQL evaluator. Our streaming SPARQL evaluator transforms all SPARQL queries into operator graphs in less than 0.023 seconds. Jena uses most of its time to load the RDF data (and to build indices of the load RDF data). The streaming SPARQL evaluator is in average approximately 1.8 times faster for all considered data sets (see Figure 9).

7. Related Work

Besides the specification of SPARQL [31], there are other contributions to define the semantic of SPARQL, e.g. [11] and [12], which define the result of SPARQL basic patterns as a set of partial functions. The formal definition of semantics and algebras of SPARQL are often partial contributions of other contributions like [15] ([30] respectively), which show the translation from a SPARQL query into XQuery/XSLT (rules respectively), or [28], which investigates the complexity of the evaluation of SPARQL queries.

[26] presents an approach for predicting the number of results of a triple pattern, which could be used for optimizing the order of joins.

[8] and [14] proposes two algebras for SPARQL, which are adapted from the relational algebra. Logical transformations are possible based on these algebras for optimization. [7] proposes operators for a RDF query language. [20] proposes a first approach for a query optimization system of SPARQL queries. [16] presents an approach for the optimization of joins by indexing pre-computed joins based on *all* input RDF triples and thus cannot be used for streaming SPARQL engines.

Streaming query engines have been proposed for other data models like the relational and XML data model and query languages like SQL and XPath/XQuery.

[24], [23], and [6] give an overview of logical transformation techniques and of physical evaluation methods for database queries using the framework of the relational algebra ([23] and [6]) or of the (tuple) relational calculus ([24]). First the relational query is transformed into a relational algebra tree ([23] and [6]) or into an object graph ([24]), on which logical transformation rules are applied in order to optimize the query evaluation. After that, depending on cost estimations, real physical operators are chosen for the logical operators from a set of different implementations for evaluating the query in the estimated fastest way.

[2] extends the relational SQL query language for registering continuous queries against streams and stored relations and presents its operators.

Ludäscher et al. [25] describe a stream-based implementation of a subset of XQuery using transducers, and Diao et al. [9] show how information filters defined as XPath expressions can be implemented using Finite State Automatons. [19] and [29] demonstrate implementation techniques for XQuery/XPath for streaming XML data.

In comparison to relational query languages and XML query languages, the basic concept of the graph query language SPARQL is the triple pattern. In comparison to all other contributions, we propose a *streaming SPARQL engine* for streams of RDF data and present an algebra and a framework for logical and physical optimization for streaming SPARQL engines dealing with special features like the triple patterns of graph query languages like SPARQL.

8. Summary and Conclusions

The triples of RDF data streams arrive continuously and are processed using a single sequential scan. Streaming query engines operate on such streams of data in order (a) to determine partial results of a query as early as possible, and (b) to discard irrelevant input as early as possible in order to save processing costs and space costs, and thus speed up query processing. We have proposed an architecture of a streaming SPARQL engine including a framework for logical and physical optimizations based on a proposed SPARQL algebra.

The experimental results of our prototype show that streaming SPARQL engines can significantly optimize the evaluation of SPARQL queries and even determines partial results before the whole RDF data is read. Furthermore, irrelevant triples of the input RDF data are discarded as early as possible such that our proposed streaming SPARQL engine is faster than non-streaming SPARQL engines and needs less memory (main memory and indexed data on disk).

9. References

[1] Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., and Tolle, K.. The rdfsuite: Managing voluminous rdf description bases. In *SemWeb'01 in conjunction with WWW*, Hongkong, 2001.

[2] A. Arasu, S. Babu, J. Widom, The CQL continuous query language: semantic foundations and query execution. *VLDB Journal,* 15(2): 121-142, 2006.

[3] Beckett, D., The design and implementation of the Redland RDF application framework. *Computer Networks,* 39(5):577-588, 2002.

[4] Broekstra, J., Kampman, A., van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. *ISWC*, Springer, Sardinia, 2002.

[5] Carroll J. J., Klyne G., Resource Description Framework: Concepts and Abstract Syntax, *W3C Recommendation*, 2004.

[6] Chaudhuri, S., An Overview of Query Optimization in Relational Systems, In *ACM PODS98*, Seattle, USA, 1998.

[7] L. Chen, A. Gupta and M. E. Kurul. A Semantic-aware RDF Query Algebra. In *COMAD* 2005, Hyderabad, India, 2005.

[8] Cyganiak, R.: A relational algebra for sparql. *Technical Report* HPL-2005-170, HP Laboratories Bristol, 2005.

[9] Diao,Y., Altinel, M., Franklin, M.J., Zhang, H., Fischer, P., Path sharing and predicate evaluation for high-performance XML filtering, *ACM ToDS*, 28(4): 467–516, 2003.

[10] M. Droop, M. Flarer, J. Groppe, S. Groppe, V. Linnemann, J. Pinggera, F. Santner, M. Schier, F. Schöpf, H. Staffler and S. Zugal, Translating XPath Queries into SPARQL Queries, *ODBASE 2007*, Vilamoura, Algarve, Portugal, 2007.

[11] E. Franconi and S. Tessaris. The Sematics of SPARQL. *Working Draft*, 2 November 2005. http://www.inf.unibz.it/krdb/w3c/sparql/.

[12] Enrico Franconi, Sergio Tessaris: The logic of RDF and SPARQL: a tutorial. *PODS 2006,* Chicago, Illinois, USA, 2006.

[13] Franz Inc., AllegroGraph 64-bit RDFStore, http://www.franz.com/products/allegrograph, 2007.

[14] Frasincar, F., Houben, G.J., Vdovjak, R., Barna, P.: RAL: an Algebra for Querying RDF. In *WWW2004*, New York, NY, USA, ACM Press, 2004.

[15] Sven Groppe, Jinghua Groppe, Volker Linneman, Dirk Kukulenz, Nils Hoeller and Christoph Reinke: Embedding SPARQL into XQuery / XSLT, *ACM SAC 2008*, Fortaleza, Ceara, Brasilien, 2008.

[16] Sven Groppe, Jinghua Groppe, Volker Linnemann, Using an Index of Precomputed Joins in order to Speed Up SPARQL Processing, *ICEIS 2007*, Funchal, Portugal, 2007.

[17] Guha, R., rdfDB: An RDF Database, http://www.guha.com/rdfdb, 2007.

[18] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Journal of Web Semantics* 3(2), 2005.

[19] Gupta, A., Suciu, D., Stream processing of XPath queries. *ACM SIGMOD*, San Diego, June 2003.

[20] Ralf Heese, Query Graph Model for SPARQL, *ER Workshops 2006* Tucson, Arizona, USA, 2006.

[21] Herman, I., Semantic Web Activity, W3C, http://www.w3.org/2001/sw/, 2007.

[22] Hewlett-Packard Company, ARP: Another RDF Parser, available at http://www.hpl.hp.com/personal/jjc/arp/, 2001.

[23] Ioannidis, Y. E., Query optimization, In *ACM Computing Surveys*, Vol. 28, No. 1, 1996.

[24] Jarke, M., and Koch, J., Query Optimization in Database Systems, In *ACM Computing Surveys*, Vol. 16, No. 2, 1984.

[25] Ludäscher, B., Mukhopadhyay, P., Papakonstantinou, Y., A transducer-based XML query processor. In *VLDB*, Hong Kong, August 2002.

[26] A. Maduko, K. Anyanwu, A. Sheth, P. Schliekelman, Estimating the Cardinality of RDF Graph Patterns, *WWW2007*, Banff, Canada, 2007.

[27] Northrop Grumman Corporation. Kowari, http://www.kowari.org, 2006.

[28] Jorge Pérez, Marcelo Arenas, Claudio Gutierrez: Semantics and Complexity of SPARQL. *ISWC 2006*, Athens, GA, USA, 2006.

[29] Peng, F., and Chawathe, S., XPath queries on streaming data. *ACM SIGMOD*, San Diego, June 2003.

[30] Axel Polleres. From SPARQL to rules (and back). In *WWW2007*, Banff, Canada, May 2007.

[31] Prud'hommeaux, E., Seaborne, A., SPARQL Query Language for RDF, *W3C Candidate Recommendation*, 2007.

[32] Wilkinson, K., Sayers, C., Kuno, H. A., Reynolds, D., Efficient RDF Storage and Retrieval in Jena2. *SWDB'03*, Berlin, 2003.