LHD: Optimising Linked Data Query Processing Using Parallelisation

Xin Wang
Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ
xw4g08@ecs.soton.ac.uk

Thanassis Tiropanis
Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ
tt2@ecs.soton.ac.uk

Hugh C. Davis
Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ
hcd@ecs.soton.ac.uk

ABSTRACT

In the past few years as large volume of Linked Data is being publishing, processing distributed SPARQL queries over the Linked Data cloud is becoming increasingly challenging. The high data traffic cost and response time significantly affect the performance of distributed SPARQL queries as the number of SPARQL end point and the volume of data at each endpoint increase. In this context, parallelisation is promising to fully exploit the potential of connections to SPARQL endpoints and thus improve the efficiency of querying Linked Data. We propose LHD, a distributed SPARQL engine that is built on a highly parallel infrastructure and able to minimise query response time, and we evaluate its performance using a BSBM based environment.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed databases*; H.3.3 [**Information Storage And Retrieval**]: Systems and Software-Performance Evaluation

General Terms

Design, Measurement, Performance, Experimentation

Keywords

SPARQL, Linked Data, distributed query processing

1. INTRODUCTION

Over the years a large amount of Linked Data has been published and forms a global data space [11]. This data space is structured and thus enables sophisticated processing. Meanwhile querying Linked Data on a large scale usually causes a large amount of network traffic and has high response time. In this context, to efficiently query the Linked Data cloud becomes an increasingly significant challenge.

Linked Data consists of RDF [10] data, which can be remotely queried by SPARQL [21] when the data is stored in SPARQL endpoints[1].The details of querying distributed RDF data are taken care of by query engines. For example, support of distributed queries is not explicitly mentioned in SPARQL 1.0 which is the current version of SPARQL specification. In the SPARQL working draft, SPARQL 1.1 [20], although multiple data services can be queried using SERVICE keyword to indicate their URIs, it is the job of researchers and developers to explore techniques that can improve the performance of query engines.

A large number of techniques have been developed for query processing in distributed database management systems (DBMS). Although distributed SPARQL queries share some characteristics with queries of distributed DBMS, their differences make it not straightforward to use well-established DBMS techniques in distributed SPARQL query engines. For example, in the Linked Data cloud, fewer statistics are available and data services are connected over the Web. The extremely large scale of the Linked Data cloud, and limitations such as network bandwidth and data services' capability bring more challenges into the research of distributed SPARQL query optimisation.

In this work, we present LHD[2] which is a parallelism-based distributed SPARQL engine. LHD adopts

- A selectivity-based cost model that is able to estimate the response time of parallel query plans;
- An optimisation algorithm that quickly produces parallel query plans having minimum communication cost using heuristics and exhaustive search;
- A parallel execution system that exploits streaming and parallelism to minimise the execution time.

We evaluate the performance of LHD using a distributed extension of the Berlin SPARQL Benchmark (BSBM) [4] presented in [30].

More specifically, in this paper we start by summarising related work of distributed SPARQL optimisation in section 2.

[1]In this paper, we only consider the scenario in which the RDF data are stored in SPARQL endpoints.
[2]Large scale High speed Distributed engine

After that we provide details of our optimisation techniques, and the implementation of LHD in section 3. We compare the performance of LHD with two existing engines FedX [25] and SPLENDID [5] in section 4. Finally we discuss the outcome of these experiments, summarise conclusions and outline future work in section 5.

2. RELATED WORK

The Linked Data cloud and distributed DBMS have many characteristics in common. They both consist of many data sets that can be accessed through network. They also aim to reduce both the amount of data being transferred and the query execution time. Many techniques have been developed to improve performance and scalability of distributed DBMS, which can be adapted to querying the Linked Data cloud. To process a distributed query, a query plan, which consists of operations to execute the query, is firstly generated. A query usually can be executed using more than one query plans, and query optimisation refers to the process of searching for the optimal query plan(s). A widely used technique of query optimisation in distributed DBMS is exhaustive search, which first builds all possible query plans and then identifies the optimum one according to a cost model [14]. A representative approach in this class is dynamic programming [26] which is adopted in most of distributed DBMS such as System R* [18]. Dynamic programming approaches can provide the optimal query plan but also take more time than other approaches. There are cases in which time is more important than the quality of query plans. As a result, greedy algorithms that give less optimal query plans in a short time have been proposed. Between dynamic programming and greedy algorithms lies the family of iterative dynamic programming (IDP), which balance time and quality [14]. All of the three approaches have been adopted in distributed SPARQL engines; dynamic programming is adopted in SPLENDID, IDP is adopted in DARQ [22], and greedy algorithms are adopted in FedX and DSP [30]. The costs of query plans are estimated by cost models, that provide an abstraction of the query environment [19]. When querying Linked Data, the accuracy of cost models are limited by the information we know about endpoints in the Linked Data cloud. Since information such as network connection speed is difficult to obtain, most of the distributed SPARQL engines' cost models consider only the size of data transferred on the Web. Operations contained in query plans are usually joins (and operations to access remote date sources). For instance, loop joins (e.g. nested loop join, hash join, indexed join) are widely used to aggregate query results from pieces. Semijoin uses existing results to filter out invalid results before they are returned. Furthermore, bind join [6], which replaces variables of a subquery using values of existing results, can dramatically reduce intermediate results under certain circumstances.

Approaches of querying Linked Data using SPARQL can be roughly divided into two categories. One is called traversal-based query execution [9]. This class of approaches discovers related data services by resolving the URIs in queries and intermediate results, and then extends the intermediate results incrementally by querying newly discovered data. The final results are achieved by applying the two steps iteratively. One representative approach in this class is the work presented in [8], which is improved in [7] by using some heuristics. Similar work can be found in [15, 16], which propose a non-block iterator to improve query performance.

The second category contains those approaches that require certain knowledge about the target data sources. For example, NetworkedGraph [24] and FedX require the URIs of data sources to be queried. Most approaches in this category work with statistics, or service descriptions, of data services. DARQ, SemWIQ [17] and DSP, for instance, maintain files containing the number of triples and the selectivity of predicates, subjects and objects, while SPLENDID uses the Vocabulary of Interlinked Datasets (VoID) [3] which is widely used to describe RDF data. Due to the contents of service descriptions being used, these approaches use different strategies to assign triple patterns to data sources. In FedX, for each triple pattern it sends *ASK* queries to all data services to identify the data services that have the potential to return results. Some approaches such as DARQ, SemWIQ and DSP select for each triple pattern those data services that contain the same predicate of this triple pattern. SPLENDID adopts a mixture of the above two strategies plus matching the class types of the resources of triple patterns. Furthermore, a sophisticated approach containing 12 rules has been proposed in [2] which intensively exploits VoID for source selection. Service descriptions also affect the way costs are estimated in each approach. FedX uses only heuristics to estimate the cost of triple patterns, while others use more sophisticated and potentially more accurate cost models. Approaches of this category regard the Linked Data cloud as a loosely coupled distributed DBMS, and therefore adopt many of the aforementioned distributed DBMS techniques. Minimum spanning tree (MST) algorithms, which belong to greedy algorithms, have been proposed in [27, 29] and adopted in DSP. DARQ and SPLENDID adopt iterative dynamic programming and dynamic programming respectively. In query execution phase, loop joins are used by most engines discussed here. In NetworkedGraph, a SPARQL version of Semijoin is implemented using FILTER. Bind join is widely applied in various engines, such as DARQ, FedX, DSP and SPLENDID. DARQ and DSP use intermediate bindings to replace variables of triple patterns, and then execute the bound triple patterns. In FedX and SPLENDID, intermediate bindings are wrapped in *UNION* clauses and therefore many bindings can be sent in one query request. Parallelism is also applied in query execution to improve efficiency. For example, ANAPSID [1] proposed an extension of Symmetric Hash Join [23] to adapt to bursty connections on the Web. FedX proposed a parallel task scheduler that uses multiple thread to execute queries.

3. TECHNIQUES AND IMPLEMENTATION OF LHD

Efficiently querying Linked Data on a large scale requires a combination of techniques. To achieve this goal, LHD aims to minimise the size of data transfered through the network as well as to maximise the data transfer rate. LHD exploits exhaustive search algorithms along with heuristics to reduce the size of network traffic, and parallelisation techniques to increase bandwidth usage. In the following sections we first describe data source selection techniques and a response time cost model that are used by LHD. Then we focus on an optimisation algorithm that quickly produces optimal query plans for parallel execution, and a parallel

```
1: d a void:Dataset ;
2:       ...
3: # simple statistics:
4:       void:triples "t_d" ;
5:       void:distinctSubjects "s_d" ;
6:       void:distinctObjects "o_d" ;
7: # statistics per predicate:
8:       void:propertyPartition [
9:            void:property p ;
10:           void:triples "t_d.p" ;
11:           void:distinctSubjects "s_d.p" ;
12:           void:distinctObjects "o_d.p" ;
13:      ], [
14:      ...
15:      ].
```

Figure 1: The statistics contained in a VoID file. Other information is omitted for simplicity.

execution system that fully exploits bandwidth and capability of data sources.

The optimisation of LHD focuses on basic graph patterns (BGPs), and therefore, for queries containing multiple BGPs we optimise each BGP separately. LHD is implemented using Jena[3]. However, Jena does not use parallelisation to combine results of different BGPs, and therefore limits the performance of LHD on certain queries. We will show the limitation in the evaluation (section 4).

3.1 Data Source Selection

In LHD, metadata of data sources are obtained from VoID files, which are provided by more and more data sources. Data source selection eliminates irrelevant data sources at an early stage and thus increases the accuracy of cost estimation. LHD analyses the predicate partition information in VoID files and identifies data sources having the same predicate as relevant candidates to a triple pattern. Then *ASK* queries enclosing the triple pattern are sent to these candidates to refine selected sources. It worth mentioning that although we use similar techniques as in existing approaches (e.g. FedX, SPLENDID and the work of 2]) to select relevant data sources, we aim to increases the accuracy of cost estimation rather than saving communication cost (since an *ASK* query has similar cost to a *SELECT* query with no results).

3.2 Cost Estimation

In this section we describe a selectivity-based response time cost model. We assume that the response time of a query request is proportional to the number of bindings transferred through the Web. Using the statistics of VoID files, we first estimate the cardinality of each operation, and then estimate the response time of query plans.

Cardinality estimation

Given the VoID file (shown in figure 1) of a data source d, we have the total number of triples t_d, distinct subjects s_d and objects o_d in d, and we have the number of triples $t_{d.p}$, distinct subjects $s_{d.p}$ and objects $o_{d.p}$ in the partition of predicate p. We assume that subjects and objects are uniformly distributed in data sources. Given a triple pattern $T : \{S\ P\ O\}$[4], we use $S(T)$ to denote a function that gives the set of relevant data sources of T, and $sel_T(x)$ to denote the selectivity of x with respect to $S(T)$, and $card_T(x)$ to denote the cardinality of x with respect to $S(T)$.

Single triple pattern

For $T = \{S\ P\ O\}$, the selectivity of each part is estimated as follows:

$$sel_T(S) = \begin{cases} \dfrac{\sum_{d \in S(T)} t_d / s_d}{\sum_{d \in S(T)} s_d} & \text{if } var(P) \wedge \neg var(S), \\ \dfrac{\sum_{d \in S(T)} t_{d.p} / s_{d.p}}{\sum_{d \in S(T)} s_{d.p}} & \text{if } P = p \wedge \neg var(S), \\ 1 & \text{if } var(S). \end{cases} \quad (1)$$

$$sel_T(P) = \begin{cases} \dfrac{\sum_{d \in S(T)} t_{d.p}}{\sum_{d \in S(T)} t_d} & \text{if } P = p, \\ 1 & \text{if } var(P). \end{cases} \quad (2)$$

$$sel_T(O) = \begin{cases} \dfrac{\sum_{d \in S(T)} t_d / o_d}{\sum_{d \in S(T)} o_d} & \text{if } var(P) \wedge \neg var(O), \\ \dfrac{\sum_{d \in S(T)} t_{d.p} / o_{d.p}}{\sum_{d \in S(T)} o_{d.p}} & \text{if } P = p \wedge \neg var(O), \\ 1 & \text{if } var(O). \end{cases} \quad (3)$$

where $var(X)$ is a function that returns *true* if X is a variable. Assuming that $sel_T(S)$, $sel_T(P)$, and $sel_T(O)$ are statistically independent, the cardinality of the triple pattern T is estimated as

$$card(T) = t \cdot sel_T(S) \cdot sel_T(P) \cdot sel_T(O) \quad (4)$$

Since we consider only the relevant data sources of T (rather than the "global graph" constructed as the union of all data sources), better source selection can increase the accuracy of the cost model.

Joined triple patterns

Consider three triple patterns $T_1 : \{?x\ p_1\ o_1\}$, $T_2 : \{?x\ p_2\ o_2\}$ and $T_3 : \{?x\ p_3\ ?y\}$, if the domain of p_1 is a subset or a superset of the domain of p_2, then $card(T_1 \bowtie T_2)$ can be estimated as $min(card(T_1), card(T_2))$. If the domain of p_1 is a subset of the domain of p_3, then $card(T_1 \bowtie T_3)$ can be estimated as $card(T_1) \cdot card(T_3)$. However, these conditions do not always hold in reality.

[3] `http://jena.apache.org/`

[4] In this section we use a question marked letter (e.g. ?x) to denote a variable, a lower-case letter (e.g. s) to denote a concrete value, and a upper-case letter (e.g. O) to denote either a variable or a concrete values.

Given two triple patterns $T_1 : \{S_1\ p_1\ O_1\}$ and $T_2 : \{S_2\ p_2\ O_2\}$ (we use concrete predicates p_1 and p_2 for simplicity), the join selectivity $sel(T_1 \bowtie T_2)$ positively correlates to the number of distinct values of the joined variable in the partitions of p_1 and p_2, and negatively correlates to the total number of distinct values of the joined variable. Assume that the sets of subjects (or objects) of different datasets are pairwise disjointed, we estimate the join selectivity as[5]

$$sel(T_1 \bowtie T_2) = \begin{cases} \dfrac{\sum_{d \in S(T_1)} s_{d.p1} \cdot \sum_{d \in S(T_2)} s_{d.p2}}{\sum_{d \in S(T_1)} s_d \cdot \sum_{d \in S(T_2)} s_d} & \text{if joined on } S_1 = S_2, \\ \dfrac{\sum_{d \in S(T_1)} o_{d.p1} \cdot \sum_{d \in S(T_2)} o_{d.p2}}{\sum_{d \in S(T_1)} o_d \cdot \sum_{d \in S(T_2)} o_d} & \text{if joined on } O_1 = O_2, \\ \dfrac{\sum_{d \in S(T_1)} s_{d.p1} \cdot \sum_{d \in S(T_2)} o_{d.p2}}{\sum_{d \in S(T_1)} s_d \cdot \sum_{d \in S(T_2)} o_d} & \text{if joined on } S_1 = O_2 \\ 1 & \text{if no shared variables.} \end{cases} \quad (5)$$

which is the possibility that a value of the join variable falls into predicate partitions of both triple patterns. For triple patterns that having variable predicates, we sum up the selectivity for each possible value of predicates. When more than two triple patterns are contained in a join, we accordingly calculate the join selectivity as the possibility that a value of the join variable falls into predicate partitions of all triple patterns. The cardinality of joined triple patterns is estimated as

$$\begin{aligned} &card(T_1 \bowtie T_2 \bowtie \ldots \bowtie T_n) \\ &= \prod_{i=1}^{n} card(T_i) \cdot sel(T_1 \bowtie T_2 \bowtie \ldots \bowtie T_n) \end{aligned} \quad (6)$$

Response time cost model

As a result of the intensive use of parallelism, LHD adopts a response time cost model rather than a network traffic model. To estimate a query plan we distinguish the execution of joins that require pre-computed bindings (e.g. bind join, Semijoin, denoted as $(q \bowtie_B t)$, where q is a join or a triple pattern and t is a triple pattern) from those do not need input bindings (e.g. hash join, nested loop join, denoted as $(q \bowtie p)$, where q and p are joins or triple patterns). Two triple patterns involved in a hash join can be executed in parallel while in a bind join they have to be executed in sequence. We say it is a *plain access plan* of a triple pattern that executes the triple pattern directly, and a *dependent access plan* if intermediate bindings are used to execute the triple pattern (it should be noticed that the execution of a dependent access plan also produces the results of a bind join). We denote a plain access plan of t as $acc(t)$, and a dependent access plan with bindings of q as $acc(q, t)$. We assume the response time of a query is proportional to the number of bindings sent to and returned from a data source, and the response time of a query plan is estimated using the following equations:

$$cost(q \bowtie p) = max(cost(q), cost(p)) \quad (7)$$

$$cost(q \bowtie_B t) = cost(q) + cost(acc(card(q), t) \quad (8)$$

$$cost(acc(t)) = rt_q + card(t) \cdot rt_t \quad (9)$$

$$cost(acc(q, t) = card(q) \cdot rt_q + card(q \bowtie t) \cdot rt_t \quad (10)$$

where rt_q is the time of sending a triple pattern or a precomputed result to a data source, and rt_t is the time of receiving a result.

3.3 Optimisation Algorithm

When processing distributed SPARQL queries, most network traffic and processing time are effected by the execution of triple patterns, and query optimisation aims to find out the optimal execution order and access plans of triple patterns. If in a query plan that all triple patterns are executed sequentially using plain access plans, any order of execution produces the same amount of network traffic and has the same processing time. Therefore, the choice that which triple patterns are executed using dependent access plans and whose bindings are required by these dependent access plans (i.e. the number and dependency of dependent access plans in a query plan) becomes essential in query optimisation. In LHD the parallel query plan is generated in two steps. We first find the query plan with minimum response time. This plan provides the information that how triple patterns are joined, from which the number and dependency of dependent access plans are decided. Then we determine the actual order to execute all triple patterns in a parallel fashion.

A query plan contains the order in which operations (e.g. access plans, joins) are executed, and the methods to execute each operation (e.g. plain access plan, dependent access plan, hash join). It should be noticed that given an execution order, the cardinality of a series of joins is independent of the concrete methods of these joins (i.e. whether a join is executed as hash join or bind join will not affect the number of results of this join). Furthermore, given the number of input bindings, we can determined whether a plain access plan or a dependent access plan generates less network traffic. Therefore, we can determine the order of execution and the methods of operations separately to reduce the search space of query plans. The methods to execute operations can also be determined during query execution using real-time statistics. If the estimated cardinality deviants too much from the actual cardinality, we can re-compute the network traffic of certain access plans and use different access plans if necessary.

Dynamic programming is used by LHD to find the minimum-response-time query plan, thus all possible execution orders are examined and for a specific order the best method to execute each operation is determined. However, during investigation of the performance of other engines, we found that a pure dynamic programming approach significantly increased the processing time on BGPs with many triple patterns. In the implementation of LHD, triple patterns that have a

[5]For unusual joins (e.g. predicate-predicate or subject-predicate), we manually choose a small number (0.00001 in our implementation) as the join selectivity.

concrete subject or object (here we call them partial-bound triple patterns for simplicity) are executed using plain access plans. We take advantage of this to improve the efficiency of processing large BGPs without compromising the quality of query plans. Given a query plan, executing a plain access plan earlier than it should will not effect the amount of network traffic of this plan (i.e. a plain access plan can be executed at any time before its results are needed by other dependent access plans). Therefore, LHD firstly scans out all triple patterns that will be executed using plain access plans (i.e. triple patterns that are partial-bound), and uses dynamic programming to decide access plans and join order for remaining triple patterns. The optimal plan found by dynamic programming is appended to the plain access plans generated before. The above procedure is shown in algorithm 1.

Algorithm 1: Generation of the optimal query plan

input : A BGP b
output: A query plan *optimal*

```
1  initialPlan ← ∅ ;
2  foreach triple pattern t of b do ;        // Generate the
   initial plan that contains plain access plans of
   all partial-bound patterns
3
4      if isPartialBound(t) then
5          remains ← b\t ;
6          initialPlan ← initialPlan ∪ plainAccPlan(t) ;
7      end
8  end
9  plan ← dynamicProgramming(remains) ;        // Use
   dynamic programming to find the optimal plan for
   remaining patterns
10 optimal ← append(initialPlan, plan)
```

Once the optimal query plans is generated, LHD determines the execution order of each operation of the query plan. In LHD, access plan starts to execute as soon as the intermediate results that it requires are ready. If a dependent access plan depends on the results of another access plan, we say the former has an execution order number which is one more than the execution order of the latter. For example, a plain access plan always has execution order 0 since it does not require any input bindings. A dependent access plan can have execution order 1 (when depending on a plain access plan) or more (when depending on another dependent access plan). LHD determines the execution order of all access plans as follows: 1) scan the given query plan and add all plain access plans to the set of execution order 0. Mark the access plans of the current execution order as bound, and increase the execution order by 1; 2) scan remaining access plans (all are dependent access plans now) and mark those dependent access plans, whose requirements are met, with the current execution order. Increase the execution order by 1 and go to step 2 if there are still access plans remaining, end the procedure otherwise. The above producer is shown as algorithm 2.

From an abstractive point of view, a query plan produced by our algorithm is a partial-directed (i.e. some edges are directed while others are not) graph as shown in figure 2. The

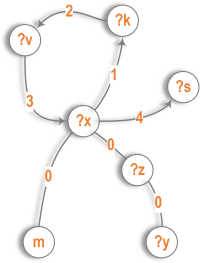
Algorithm 2: Determining the execution order

input : A query plan p
output: A query plan with execution orders *parallel*

```
1  executionOrder ← 0 ;
2  bound ← ∅ ;
3  while p ≠ ∅ do
4      foreach access plan j in p do
5          dep ← getDependingPlans(j) ; // Get the access
           plans whose results are required by j
6          if dep ⊂ bound then ;    // Add j to the current
           execution order if required plans are ready
7
8              parallel[executionOrder] ←
               parallel[executionOrder] ∪ j ;
9              bound ← bound ∪ j ;          // Add j to bound
10             p ← p − j ;
11         end
12     end
13     executionOrder ← executionOrder + 1 ;
14 end
```

t_1 : m p_1 ?x, 0
t_2 : ?z p_2 ?y, 0
t_3 : ?z p_3 ?x, 0
t_4 : ?x p_4 ?k, 1
t_5 : ?v p_5 ?k, 2
t_6 : ?v p_6 ?x, 3
t_7 : ?s p_7 ?x, 4

(a) Triple patterns with execution order



(b) A corresponding query plan

Figure 2: An example query and its execution plan

nodes of the graph are subjects or objects and the edges are triple patterns. A undirected edge represents a plain access plan of a triple pattern, while a directed edge represents a dependent access plan that consumes bindings from its starting node. Each edge has a execution order number, that edges with smaller order numbers are executed earlier. Execution order numbers are used to determine the execution order for edges connected to the same node, but not edges connected to different nodes (i.e. two edges of different nodes are executed in a random order).

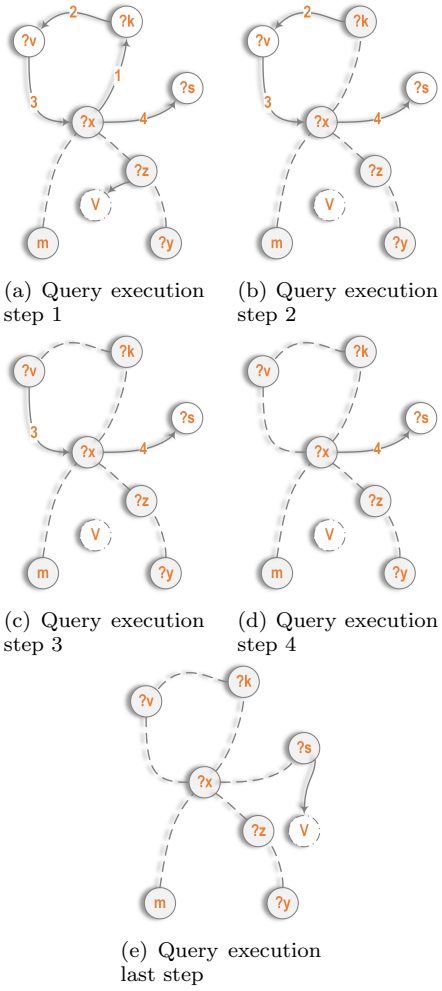
3.4 Parallel Query Execution

To improve the efficiency of query execution, LHD adopts a sophisticated concurrent control system to fully exploit the bandwidth and data sources. We separate execution of query plans from communicating with (i.e. sending queries to or receiving results from) data sources. The former is controlled by the query plan executor, and the latter is managed by the communication manager. The query plan executor submits query execution tasks to the communication manager, and the communication manager determine when to execute certain tasks. This design enables us to independently control the traffic to each data source and thus all data sources can work at full strength.

Query plan executor

For the query plan executor, each edge is regarded as a data stream which contains the results of executing the triple pattern of this edge. We use a quadruple $\{t, n, s, E\}$ to denote a stream of evaluating a triple pattern t that has an execution order n, a starting node s, and a set of end nodes E where the stream goes into. For a stream corresponding to a plain access plan, s is ϕ and E contains all variables of t that are used as join variables. In case of a dependent access plan, s is the node providing bindings and E contains the other node of t. A stream consumes bindings of s (if not empty) and pushes the results of evaluating t to the nodes in E. Streams are joined at nodes. A node v is denoted as a pair $\{In, Out\}$ where In denotes a list of incoming streams and Out a list of outgoing streams, ordering by their execution order number from small to large. A node accepts results of incoming streams, joins the results and triggers certain outgoing streams. After execution starts, all streams of the smallest execution order (i.e. 0) start. At a node v, incoming streams that have not been used are joined. An outgoing streams starts as soon as all incoming streams, whose execution order is larger than this outgoing stream, have started. Once the results of a incoming stream is being consumed by a outgoing stream, the incoming stream is marked as "consumed" and will not be involved in future join or passed to later outgoing streams (thus to make sure the stream will not be joined twice). In case no outgoing streams exist for new incoming streams (i.e. all out going streams have smaller execution order than the new incoming streams), these streams are redirected to a virtual node at where the final results of the query are produced.

Figure 3 shows a step-by-step example of executing the query plan shown in figure 2. At the beginning (figure 3a) three streams of execution order 0, $(t_1, 0, \phi, \{?x\})$, $(t_2, 0, \phi, \{?z\})$ and $(t_3, 0, \phi, \{?x, ?z\})$ start. Since $?y$ is not used as a join variable, no streams provide data to it. The streams of t_2



(a) Query execution step 1

(b) Query execution step 2

(c) Query execution step 3

(d) Query execution step 4

(e) Query execution last step

Figure 3: Execution of a query plan

and t_3 are joined and marked as consumed at node $?z$. Join results are pushed to the virtual node V. In the meantime, the streams of t_1 and t_3 are joined at $?x$ (but not marked as consumed yet). In the next step (figure 3b, stream $(t_4, 1, ?x, \{?k\})$ consumes the join results of streams of t_1 and t_3, both of which are marked as consumed at node $?x$, and executes t_4 using a dependent access plan. The stream of t_7 keeps waiting since the stream of t_6, whose execution order is smaller than t_7, has not started yet. Step 3 (figure 3c) is similar to step 2 that executes t_5 using a dependent access plan. In step 4 (figure 3d the stream of t_6 goes back into $?x$ but is not joined with any stream (since all other incoming streams of $?x$ are marked as consumed). In the final step (figure 3e) only the stream of t_6 is passed to the stream of t_7. The results of executing t_7 that go into node $?s$ are passed to the virtual node V. All results (as streams) at V are joined to produce the final results of a query.

Communication manager

The actual execution of a query is managed by the communication manager. For each data source the communication manager maintains several worker threads that send query requests to and receive responses from the data source, and a

queue that stores tasks submitted to this data source. The number of threads of each data source is set with respect to the capability of and the connection to the data source. Once the query plan executor invokes a stream, one or several query execution requests are submitted to the communication manager. A plain access plan generates only one request. For a dependent access plan more than one requests are possible since the input bindings of the dependent access plan can be partitioned into multiple segments and then executed in parallel (i.e. using horizontal partition [13] to achieve intra-operation parallelism [12]). For example, a dependent access plan that has ten input bindings can be executed in parallel as two dependent access plans each with five input bindings each, or even ten dependent access plans each with one input binding. For each request from the query plan executor, the communication manager dispatches tasks to all relevant data sources of the triple pattern of the request. A task first goes into the task queue, waiting if all worker threads are busy, being executed otherwise. Once the task queue becomes empty, all worker threads are suspended until new tasks coming in.

The main advantage using this communication manager is to control communication to different data sources independently, and thus ensures that all data sources work at their strength without being over flooded. Furthermore, separating plan executor from communication manager enables query plan execution to proceed without waiting for actual query execution (as long as some data sources are providing result streams), and query tasks are continuously submitted to communication manager (to keep as many worker threads working as possible).

3.5 Implementation of LHD

The execution system of LHD is built using pipelined parallelism such as Double-pipelined Hash Join [23] and XJoin [28]. Instead of using two hash tables like in a Double-pipelined Hash Join, we maintain multiple hash tables at a node to enable joining more than one streams simultaneously. A result coming from one stream is stored in the hash table of this stream, and at the same time probed against the hash tables of other streams. For example, at a time three streams a, b and c are joined at a node $?x$, and three hash tables H_a, H_b and H_c are maintained respectively. Once a result comes from a, it is stored to the hash table H_a under the key of the value of $?x$, and probed against H_b and H_c on the same value of $?x$. A join result is produced as soon as matching records are found in both H_b and H_c, and given to outgoing streams that consume the result. This multiple hash join enables a node to execute several incoming streams as well as outgoing streams in parallel. The execution will not be delayed unless all data sources stop providing results.

When executing a dependent access plan there could be duplicated values of the depended variable (e.g. considering two input bindings $(?x \rightarrow x_1, ?y \rightarrow y_1), (?x \rightarrow x_1, ?y \rightarrow y_2)$ for triple pattern $\{?x\ p\ ?z\}$, only one value $(? \rightarrow x_1)$ is required by the dependent access plan of the triple pattern). To eliminate unnecessary network traffic, we propose the Hash Dependent Access (HDA) operator, that partitions the input bindings using a hash table on the values of the depended variable(s). Therefore we only use distinct values to execute a dependent access plan, and the returned results of a specific input value are joined with bindings of the same value in the hash table. When only one binding is given for a dependent access plan, variables in the triple pattern of this access plan are replaced by values of the given binding (i.e. the implementation of bind join in DARQ and DSP). Otherwise the input bindings are attached as inline data using the *VALUES*[6] syntax in the dependent access plan. For example, to execute $\{?x\ p\ ?z\}$ with input bindings $(?x \rightarrow x_1, ?y \rightarrow y_1), (?x \rightarrow x_1, ?y \rightarrow y_2)$, firstly a hash table $x_1 \rightarrow \{(?x \rightarrow x_1, ?y \rightarrow y_1), (?x \rightarrow x_1, ?y \rightarrow y_2)\}$ is built. Then $\{x_1\ p\ ?z\}$ is evaluated against relevant data sources. The results $(?z \rightarrow z_1), (?z \rightarrow z_2)$ are joined with $(?x \rightarrow x_1, ?y \rightarrow y_1), (?x \rightarrow x_1, ?y \rightarrow y_2)$ to produce the complete results.

4. EVALUATION

In our previous work we have proposed a benchmark which extends BSBM to distributed SPARQL evaluation [30]. In our experiments, we generate about 40 million triples using BSBM tools. All the triples are distributed over 10 SPARQL endpoints which are deployed on 10 remote virtual machines, following a power law distribution. All SPARQL endpoints are set up using Sesame 2.4.0 and Apache Tomcat 6 with default settings. The engines under testing are run independently on a machine with an Intel Xeon W3520 2.67 GHz processor and 12 GB memory.

4.1 Experiment Settings

In our experiment we use all queries of BSBM except query 6, 9 and 12. Query 6 contains only two triple patterns which are too few to test query optimisation techniques (basically there are only two possible order of execution, the choice of which is depends on chance more than optimisation techniques). Query 9 is a DESCRIBE query, whose results depend on the implementation of SPARQL endpoints rather than query engines. Query 12 is a CONSTRUCT query. Although, from a query optimisation point of view, few differences exist between CONSTRUCT queries and SELECT queries, we prefer not to change the BSBM's query and exclude query 12. Since query 11 has only one execution plan, we keep it to demonstrate the performance of query execution of each engine. Each query of BSBM is actually a template, from which many instances are randomly generated in a test. We prefer those query instances that have more intermediate results. Therefore, for each query, we firstly generate thousands of instances, and then select the top 20 instance that have more intermediate results than others. This refinement of query instances significantly increase the number of intermediate results on query 1 and 5. However, for a few queries (query 2, 4, and 10), none of their thousands of instances had many intermediate results, which is probably due to the design of BSBM.

We evaluate each engine with 5 warm up runs and 20 test runs (each run uses a different instance of query). After the execution of each query instance, we set a 10 seconds interval for remote SPARQL endpoints to clear their job. Time out of query execution is set to 120 seconds.

The metrics being used include the number of queries executed per second (QPS), the size of outgoing (mainly queries

[6] `http://www.w3.org/TR/sparql11-query/#inline-data`

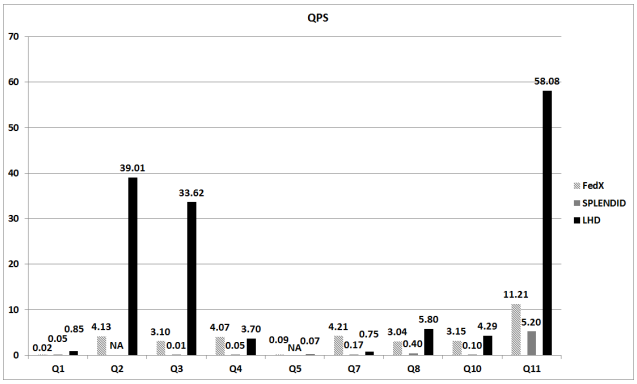


Figure 4: QPS of the three engines per query. NA means time out.

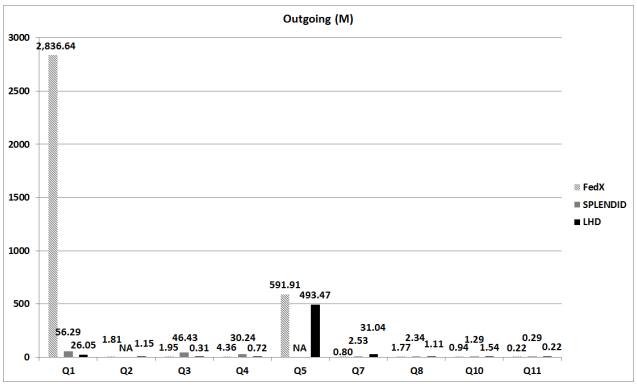


Figure 5: Outgoing data of the three engines per query. NA means time out.

being sent out) and incoming (mainly query results coming back) data, average CPU usage and memory usage. We also confirm that all engines return the same number of results for the same query.

4.2 Evaluation Results

We present the results of the three engines as the following figures: QPS is shown in figure 4; outgoing data (in megabyte) is presented in figure 5; incoming data (in megabyte) is presented in figure 6; average CPU usage (in percentage) is shown in figure 7, and average memory usage (in megabyte) is shown in figure 8.

4.3 Results Analysis

From an overview of figure 4, we can see LHD improves query efficiency for most queries. Meanwhile we notice that LHD has higher memory and CPU usage than FedX (figure 8 and 7). This is because LHD maintains a number of threads for each data service separately, and therefore more threads are used than in FedX. Since these threads are active only when the task queues are not empty, the CPU and memory usage of LHD are related to both the number of threads and intermediate bindings (more bindings consume more memory and possibly generate more execution tasks). Query 1 confirms this argument that LHD improves query performance with lower network flow and memory usage. Some may argue that it is possible to improve the performance of

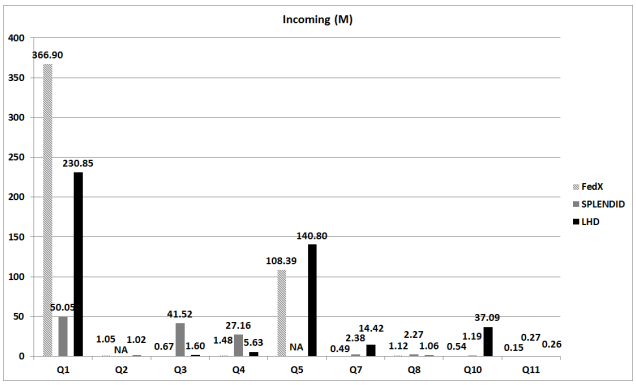


Figure 6: Incoming data of the three engines per query. NA means time out.

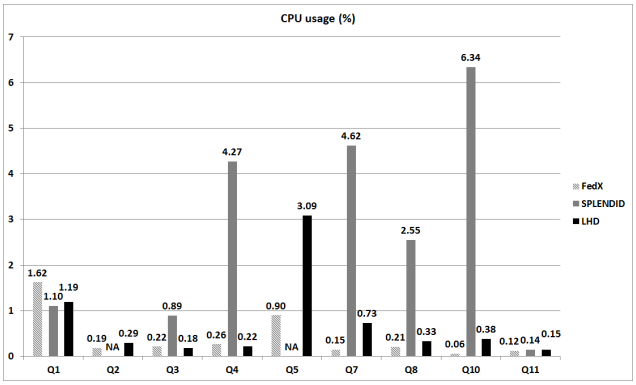


Figure 7: CPU usage of the three engines per query. NA means time out.

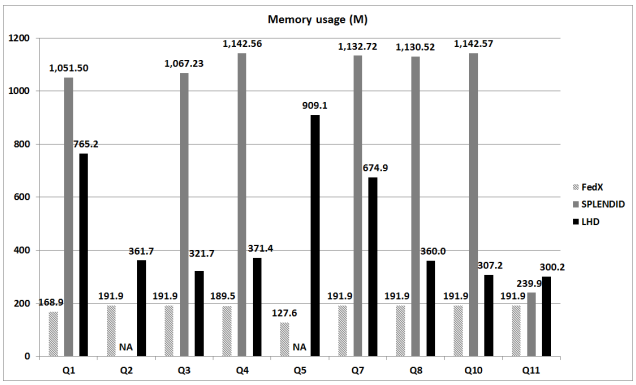


Figure 8: Memory usage of the three engines per query. NA means time out.

FedX by increasing the number of threads. However, the number of concurrent threads used in FedX is limited by SPARQL endpoints that have least computational power. Those endpoints will be flooded by more threads and no results will return. By comparing the incoming and outgoing data (figure 6 and 5), we notice that there is no significant difference between the total amount of network flow of LHD and FedX for most queries. This further confirms that our execution system optimises bandwidth use on average.

LHD is built on Jena, in which BGPs of OPTIONAL queries are pipelined and bindings are passed one by one. This slows down LHD's query execution since the bindings passed between BGPs cannot be merged into one query request. It is expected that queries with many small BGPs (i.e. BGPs containing a few triple patterns) are affected more than those have one big BGP and a few smaller BGPs. Query 7 contains several small BGPs while query 8 has one big BGP and a few small ones, and therefore LHD shows a lower QPS than other engines on query 7 but a higher QPS on query 8.

SPLENDID is able to generate the best query plans but does not optimise its query execution. It shows lower QPS even when its network flow is lower than others (query 1). The downside of SPLENDID's query execution can be confirmed by query 11. Query 11 has only one query plan and thus its execution time is mainly determined by query execution of each engine. Furthermore, the query optimisation of SPLENDID takes a large amount of time on complex queries, which leads the time our on query 2 and query 5 (the large number of results of query 5 is also part of the reason). The significant optimisation time on complex queries does not occur in LHD as a result of using heuristics in the optimisation algorithm.

The network traffic charts also open questions worthy of further investigation. For example, considering the query optimisation techniques used by SPLENDID, FedX and LHD, it is excepted that SPLENDID has the best query plan and thus has the least amount of network traffic while FedX has the most. However, our experimentation results show that for most queries, SPLENDID has the most network traffic while FedX and LHD are comparable. The reason of this contradiction could be that although SPLENDID generates the best query plans, its query execution strategy generates more traffic than FedX and LHD. Another possibility is that the estimated cost (a result of either statistics or cost models) of SPLENDID or LHD are not accurate enough and thus lead to pseudo-optimal query plans. Exploration on either reason will lead to better understanding and further improvement of Lined Data queries.

5. CONCLUSION

In this paper we explored parallel techniques to improve the efficiency of distributed SPARQL query processing. We proposed LHD, a distributed SPARQL engine that adopts a response time cost model, an optimisation algorithm using both heuristics and exhaustive search to find optimal parallel query plans and a sophisticated parallel query execution system. We evaluated our approach in a BSBM-based environment and demonstrated that our techniques were effective at reducing communication cost and response time of query processing.

Based on our experiment we observed that, although parallel techniques improve efficiency, they can lead to the consumption of more resources in terms of CPU, memory and bandwidth, and to potentially over-flood SPARQL endpoints. We also noticed that query optimisation based on pure exhaustive search algorithms can significantly increase query processing time on queries with many triple patterns. Finally, our experiment revealed higher network traffic than expected which could be a result of certain query execution strategy or inaccurate cost estimation.

In the future we plan to explore methods that can reduce resource consumption of parallel query execution approaches. In addition, the optimisation accross BGPs (i.e. optimising BPGs of a query together ranther than separately) is worthy of further investigation. Furthermore, more experiments will be performed to find out potential issues in existing techniques used in distributed SPARQL queries.

6. REFERENCES

[1] M. Acosta, M. Vidal, and T. Lampo. ANAPSID: an adaptive query processing engine for SPARQL endpoints. In *ISWC2011*, 2011.

[2] Z. Akar, T. G. Halaç, and E. E. Ekinci. Querying the Web of Interlinked Datasets using VOID Descriptions. In *Linked Data on the Web (LDOW2012)*, 2012.

[3] K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. Describing Linked Datasets On the Design and Usage of voiD , the âĂIJ Vocabulary Of Interlinked Datasets âĂİ. *Linked Data on the Web Workshop (LDOW 09), in conjunction with 18th International World Wide Web Conference (WWW 09)*, 2009.

[4] C. Bizer and A. Schultz. The Berlin SPARQL Benchmark. *International Journal On Semantic Web and Information Systems-Special Issue on Scalability and Performance of Semantic Web Systems*, 2009.

[5] O. Görlitz and S. Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *Proceedings of the 2nd Workshop on Consuming Linked Data (COLD2011)*, 2011.

[6] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proceedings of The International Conference on Very Large Data Bases*, pages 276–285. Citeseer, 1997.

[7] O. Hartig. Zero-Knowledge Query Planning for an Iterator Implementation of Link Traversal Based Query Execution. *ESWC (2011)*, 6643:154–169, 2011.

[8] O. Hartig and C. Bizer. Executing SPARQL Queries over the Web of Linked Data. *The Semantic Web-ISWC 2009*, pages 293–309, 2009.

[9] O. Hartig and J.-C. Freytag. Foundations of traversal based query execution over linked data. In *Proceedings of the 23rd ACM conference on Hypertext and social media - HT '12*, page 43, New York, New York, USA, June 2012. ACM Press.

[10] P. Hayes and B. McBride. RDF semantics, 2004.

[11] T. Heath and C. Bizer. Linked Data: Evolving the Web into a Global Data Space. *Synthesis Lectures on the Semantic Web: Theory and Technology*, 1(1):1–136, Feb. 2011.

[12] W. Hong and M. Stonebraker. Optimization of parallel

query execution plans in XPRS. *Distributed and Parallel Databases*, pages 218–225, 1991.

[13] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.

[14] D. Kossmann and K. Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Transactions on Database Systems*, 25(1):43–82, Mar. 2000.

[15] G. Ladwig and T. Tran. Linked Data Query Processing Strategies. *The Semantic WebâĂŞISWC 2010*, pages 453–469, 2010.

[16] G. Ladwig and T. Tran. SIHJoin: Querying Remote and Local Linked Data. *The Semantic Web: Research and Applications*, 6643:139–153, 2011.

[17] A. Langegger, W. Wöß, and M. Blöchl. A Semantic Web Middleware for Virtual Data Integration on the Web. *The Semantic Web Research and Applications*, 5021:493–507, 2008.

[18] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. *ACM SIGMOD Record*, 17(3):18–27, June 1988.

[19] M. Özsu and P. Valduriez. *Principles of distributed database systems*. Prentice Hall, 1999.

[20] E. Prud'hommeaux. SPARQL 1.1 Federation Extensions. W3C Working Draft (1 June 2010), 2010.

[21] E. Prud'Hommeaux and A. Seaborne. SPARQL query language for RDF, 2008.

[22] B. Quilitz. Querying distributed RDF data sources with SPARQL. *The Semantic Web: Research and Applications*, pages 524–538, 2008.

[23] L. Raschid and S. Y. W. Su. A Parallel Processing Strategy for Evaluating Recursive Queries. pages 412–419, Aug. 1986.

[24] S. Schenk. Sesame RDF repository extensions for remote querying. *ZNALOSTI Conf*, pages 375–378, 2008.

[25] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *Proceedings of the 10th International Semantic Web Conference, Bonn, Germany*, 2011.

[26] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. *Access path selection in a relational database management system*. ACM Press, New York, New York, USA, May 1979.

[27] 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. ACM, 2008.

[28] T. Urhan and M. Franklin. XJoin: Getting fast answers from slow and bursty networks. *University of Maryland Technical Report CS-TR-3994 (Feb.)*, 1999.

[29] B. P. Vandervalk, E. L. McCarthy, and M. D. Wilkinson. Optimization of Distributed SPARQL Queries Using Edmonds' Algorithm and Prim's Algorithm. In *2009 International Conference on Computational Science and Engineering*, volume 1, pages 330–337. IEEE, 2009.

[30] X. Wang, T. Tiropanis, and H. C. Davis. Evaluating Graph Traversal Algorithms for Distributed SPARQL Query Optimization. In *JIST2011*, 2011.