Open Access

Open Journal of Semantic Web (OJSW)
Volume 2, Issue 1, 2015

http://www.ronpub.com/ojsw
ISSN 2199-336X

Distributed Join Approaches for W3C-Conform SPARQL Endpoints

Sven Groppe, Dennis Heinrich, Stefan Werner

Institute of Information Systems (IFIS), University of Lübeck, Ratzeburger Allee 160, D-23562 Lübeck, Germany,
{groppe, heinrich, werner}@ifis.uni-luebeck.de

Abstract

Currently many SPARQL endpoints are freely available and accessible without any costs to users: Everyone can submit SPARQL queries to SPARQL endpoints via a standardized protocol, where the queries are processed on the datasets of the SPARQL endpoints and the query results are sent back to the user in a standardized format. As these distributed execution environments for semantic big data (as intersection of semantic data and big data) are freely accessible, the Semantic Web is an ideal playground for big data research. However, when utilizing these distributed execution environments, questions about the performance arise. Especially when several datasets (locally and those residing in SPARQL endpoints) need to be combined, distributed joins need to be computed. In this work we give an overview of the various possibilities of distributed join processing in SPARQL endpoints, which follow the SPARQL specification and hence are "W3C conform". We also introduce new distributed join approaches as variants of the Bitvector-Join and combination of the Semi- and Bitvector-Join. Finally we compare all the existing and newly proposed distributed join approaches for W3C conform SPARQL endpoints in an extensive experimental evaluation.

Type of Paper and Keywords

Regular research paper: *Semantic Web, SPARQL endpoint, distributed join, W3C-conform, SPARQL, query processing, query optimization*

1 Introduction

The current World Wide Web enables an easy, instant access to a vast amount of online information. However, the content in the Web is typically for human consumption, and is not tailored for machine processing. The Semantic Web [50] is hence intended to establish a machine-understandable web.

The World Wide Web Consortium (W3C) [43] developed numerous standards and approaches around the Semantic Web vision. Among them is the Resource Description Framework (RDF) [44], which is used as the data model of the Semantic Web. The W3C also defined SPARQL [14] as RDF query language, and the ontology languages RDFS [9] and OWL [38] to express knowledge.

There are masses of Semantic Web data freely available to the public – thanks to the efforts of the linked data initiative [26]. In 2011 the data contained over 30 billion triples in nearly 300 datasets with over 500 million links between these datasets. These numbers still grow rapidly: In 2014 these numbers have already been more than doubled [36]. Many of these freely available datasets are additionally accessible via SPARQL query servers, called *SPARQL endpoints*. Anyone can submit SPARQL queries to SPARQL endpoints via a standardized protocol [46], where the query is processed on the dataset of the SPARQL endpoint and the query result is returned in one of the standardized formats XML [49], JSON [48] or TSV/CSV [47]. In this way one does not

need to set up an own SPARQL database and import the data into it, but can use these SPARQL endpoints for data access.

The RDF query language SPARQL [45] in its current version 1.1 [14] took a further step in federated query processing over distributed SPARQL endpoints by introducing the SERVICE clause. With the SERVICE clause one can express subqueries to be processed on a remote SPARQL endpoint. In this way the results of a remote SPARQL endpoint can be combined with local data. Even several SERVICE clauses can be used in one query, such that the data residing in several SPARQL endpoints can be easily combined. This is one of the most powerful features of SPARQL 1.1: Even the mature relational query language SQL [20] does not standardize any comparable language features, which combine the data of remote database servers or simply combine the data of several local databases within one query. Proprietary extensions like OPENROWSET in SQL Server [29] and the federated storage engine in MySQL [32] are not integrated in the SQL standard.

By filtering out irrelevant data already at the SPARQL endpoint the performance is increased and the communication costs are decreased, whenever data of a number of SPARQL endpoints or local data and the data of SPARQL endpoints is combined. Distributed join approaches [13] in distributed databases are designed to achieve this goal in the relational world in a homogeneous environment, where only one distributed database management system is running.

1.1 Utilizing Third-Party SPARQL Endpoints

When a user has an own SPARQL endpoint for freely available linked data, she/he must update datasets in the endpoint on a regular basis in order to keep them up-to-date. Furthermore, the user must reserve hardware for the SPARQL endpoint: This is wasting her/his own resources, regarding the fact that many organizations offer SPARQL endpoints with the linked data the user needs for her/his applications. We are hence interested in the following scenarios: Users do *not* run their own SPARQL endpoint. Intead, they utilize freely available and accessible third-party SPARQL endpoints in order to save own (computing and storage) resources and to operate on the latest data.

Typically a user does not have any influence on the configuration and setup of the SPARQL endpoints freely available and accessible via the Internet. Different SPARQL endpoints might use different SPARQL query evaluators and Semantic Web database management systems with varying Quality-of-Service parameters [11, 5]. The only standardized way of communicating with a SPARQL endpoint is using the standardized protocols, query languages and result formats specified by the W3C. We call a SPARQL endpoint a *W3C-conform SPARQL endpoint*, if the SPARQL endpoint offers only the standardized way of communication specified by W3C. Most freely available SPARQL endpoints are W3C-conform SPARQL endpoints, such that users rely on these standards instead of advanced (but proprietary) prootocols. In this heterogeneous environment, distributed joins can hence only be expressed by using the language features of SPARQL. Or in other words: The distributed join approaches have no way to set up requests to SPARQL endpoints other than to send SPARQL queries. We will hence focus on and investigate only distributed join approaches, which generate SPARQL queries for their requests to SPARQL endpoints.

1.2 Our Contributions

Our contributions are:

- describing different distributed join approaches, which utilize only SPARQL language features to filter out irrelevant results already at the remote SPARQL endpoint,
- presenting a set of new distributed join approaches corresponding to the Bitvector-Join [28] in distributed databases, which use only SPARQL language features and adapt the Btvector-Join concept (having the best performance in our real-world scenario),
- proposing a variant, which is a combination of the Semi- [51] and Bitvector-Join approach, being usually faster than any of the two approaches, because it not only avoids costly hash operations, but also reduces the communication costs dramatically,
- showing that an additional built-in function in the SPARQL specification would allow Bitvector-Joins with superior performance, and
- a comprehensive experimental evaluation showing the advantages and disadvantages of the proposed distributed join approaches for W3C-conform SPARQL endpoints.

1.3 Organization of Paper

The remainder of this paper is organized as follows: Section 2 introduces the technologies of Semantic Web, which are used in this work, and further related work. In Section 3, we discuss the different distributed join approaches, including existing ones and ones proposed in this work, using concrete examples. Section 3 first

gives an overview of these approaches and then classify them according the important properites like how the client must process the result of remote queries. Section 4 presents two types of experiments running on synthetic and on real-world data. We also provide an extensive analysis of the experimental results and discuss the reasons for observed runtime characteristics in the experiments. Finally, we summarize the proposed approaches for distributed joins for SPARQL endpoints and conclude our paper in Section 5.

2 Basics of Semantic Web and Remote Queries

In this section, we introduce the technologies of Semantic Web, which are used in this work, and other further related work. We introduce the data model of the Semantic Web in Section 2.1 and its query language in Section 2.2. Section 2.3 contains further related work, where we especially focus on federation over SPARQL endpoints and distributed join approaches for the Semantic Web.

2.1 Data Format RDF

The *Resource Description Framework (RDF)* [44] is a language originally designed to describe (web) resources, but can be used to describe any other information. RDF data consists of a set of triples. Following the grammar of a simple sentence in natural language, the first component s of a triple $(s,\ p,\ o)$ is called the subject, p the predicate and o the object. More formally:

Definition (RDF triple): Assume there are pairwise disjoint infinite sets I, B and L, where I represents the set of Internationalized Resource Identifiers (IRI), B the set of blank nodes and L the set of literals. We call a triple $(s,p,o) \in (I \cup B) \times I \times (I \cup B \cup L)$ an RDF triple, where s represents the subject, p the predicate and o the object of the RDF triple. We call an element of $I \cup B \cup L$ an RDF term.

In visualizations of the RDF data, the subjects and objects become (unique) nodes, and the predicates directed labeled edges from their subjects to their objects. The resulting graph is called *RDF graph*.

Listing 1 shows an example of RDF data consisting of three triples, which describe a book published by `Publisher` with the title `"SPARQL"` from the author `"Ghostwriter"`, in the serialization format N3 [8].

```
@prefix ex: <http://www.ifis.uni-
    luebeck.de/example/>.
ex:book ex:publishedBy ex:Publisher .
ex:book ex:title "SPARQL" .
ex:book ex:author "Ghostwriter" .
```

Listing 1: Example of RDF data

2.2 Query Language SPARQL

The World Wide Web Consortium proposed the RDF query language SPARQL [45, 14] for searching in RDF datasets. Listing 2 presents an example SPARQL query.

The structure of SPARQL queries is similar to that of SQL queries for relational databases. The most important part of a SPARQL query is the *WHERE*-clause. A WHERE-clause consists of triple patterns, which are used for matching triples. Known components of a triple are directly given in a triple pattern, unknown ones are left as variables (starting with a ?). If the known components in a triple pattern matches the corresponding ones in the triple, the variables in the triple pattern are bound to the corresponding RDF terms in the matching triple. The result of a triple pattern contains an entry (with bound variables) for each matching triple. The results of several triple patterns are joined over common variables. All variables given between the keywords *SELECT* and *WHERE* appear in the final result, all others are left out. Besides *SELECT*-queries, also *CONSTRUCT*- and *DESCRIBE*-queries are available to return RDF data. Furthermore, *ASK*-queries can be used to check for the existence of results indicated by a boolean value. Analogous to N3, SPARQL queries may declare prefixes after the keyword *PREFIX*.

Listing 2 presents an example SPARQL query, the result of which is {(?title="SPARQL", ?author="Ghostwriter")} when applied to the RDF data in Listing 1.

```
PREFIX ex: <http://www.ifis.uni-
    luebeck.de/example/>
SELECT ?title ?author WHERE {
 ?book ex:publishedBy ex:Publisher .
 ?book ex:title ?title .
 ?book ex:author ?author .
}
```

Listing 2: Example of a SPARQL query

Besides this basic structure, SPARQL offers several other features like *FILTER* clauses to express filter conditions, *UNION* to unify results and *OPTIONAL* for a (left) outer join.

SPARQL in its new version 1.1 [14] additionally sup-

ports enhanced features like update queries, paths and remote queries.

2.2.1 Remote Queries

Listing 3 presents the schema for queries containing a *SERVICE*-clause. Here, `A` and `B` are placeholders for any language constructs of SPARQL: While the demands expressed in `A` are processed on the local data, the demands expressed in `B` are sent to a SPARQL endpoint residing at `endpoint-url`, which are there processed and the result of `B` is sent back to the client. The demands in `B` are encapsulated in a whole SPARQL query, which we call *remote query*, before sending it to the corresponding SPARQL endpoint. Afterwards, the results of `A` and `B` are joined to form the final result of the query. `A` and `B` themselves could contain again *SERVICE*-clauses, and thus SPAQL endpoints may be accessed several times within a single query.

Listing 4 contains an extended version of the query in Listing 2: The prices of the determined books in the local data are retrieved from a SPARQL endpoint. The query is an example for scenarios with static data stored locally, which is seldom updated, and dynamic data with frequent updates residing at a server. Remote queries can be further used to combine datasets residing at different servers in federated scenarios.

```
SELECT * WHERE {
  A
  SERVICE <endpoint-url> {
   B
  }
}
```

Listing 3: Schema of a SERVICE clause

2.2.2 Operator Graph

Basic operations of relational queries [12, 13] and also SPARQL queries [42] can be expressed in terms of (nestable) operators of the relational algebra. Relational expressions can be visualized by an operator tree or by its more general form, an operator graph. An additional operator for SPARQL queries (in comparison to relational queries) is the triple pattern scan, which is a special form of an index scan operator, yielding the result of a triple pattern. Figure 1 presents the operator graph of the SPARQL query in Listing 2. The operators at the bottom are the triple pattern scan operators of the three triple patterns contained in the SPARQL query. The results of the triple patterns are combined by a join $\bowtie$ over their common variable `?book`. Finally the projection operator

```
PREFIX ex: <http://www.ifis.uni-
    luebeck.de/example/>
SELECT ?title ?author ?price
WHERE {
 ?book ex:publishedBy ex:Publisher .
 ?book ex:title ?title .
 ?book ex:author ?author .
 SERVICE ex:sparql {
   ?book ex:price ?price .
 }
}
```

Listing 4: Example of a SPARQL query with SERVICE clause

π is applied to the result of the join: Only the bound values of the variables `?title` and `?author` remain in the final result.

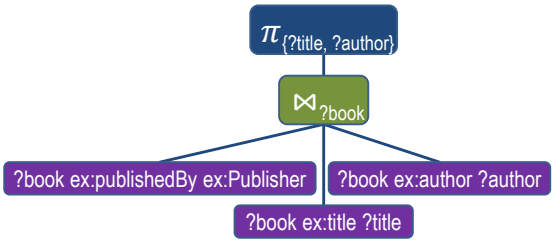

Figure 1: Operator graph of the SPARQL query in Listing 2

Furthermore, for remote queries in SERVICE clauses of SPARQL queries, we can introduce an additional SERVICE operator. Depending on the used algorithm for distributed joins, we have to generate different variants of operator graphs in context of the SERVICE operator. We will discuss these different variants in Section 3.

2.3 Further Related Work

Table 1 provides an overview of the different federation frameworks and their distributed join approaches. Some just use the distributed join approaches of underlying engines (like [18, 3, 25, 24, 23]). Other frameworks have own (often non-blocking) implementations of distributed joins (e.g. [27, 39, 22, 21, 2]). A third group implements distributed joins by generating corresponding SPARQL queries (for example [37, 31, 51], Jena [41] and Sesame [10]). We especially focus on approaches of this third group, extensively describe possibilities for expressing distributed joins with SPARQL 1.1 and propose some

new distributed join approaches for this group, i.e. the Value and Value (X chars) approaches and variants of the Bitvector-Join approaches. [34, 35] contain surveys over federation frameworks over SPARQL endpoints.

Most of the federation systems in Table 1 provide such features that users do not need to know where data is resided, but just formulate one query the triple patterns of which are automatically matched against a number of sources. In order to avoid querying many sources, which do not contain any triples matching the triple patterns in the query, the federation systems typically apply sophisticated source selection approaches. After selecting relevant sources the federation systems need optimized distributed join approaches for increasing the overall query processing, which is the focus of this paper.

DARQ [33] creates an index called service description to select the relevant sources. Each service description contains statistical information about distinct predicates in the data of a SPARQL endpoint. These distinct predicates are simply matched against predicates in the triple patterns of the considered query. If a variable at the predicate position of a triple pattern is bound with a value by a previous operation (the result of which will be joined with that of the triple pattern), then we say that the *predicate of the triple pattern is bound*. If a predicate in a triple pattern is neither bound nor is a constant IRI, the source selection approach of DARQ fails. Besides service descriptions, DARQ offers query rewriting based on a cost-based optimization to further reduce the query processing time and the bandwidth usage.

In comparison to DARQ, LHD [39] and ADERIS [27] additionally support triple patterns with unbound predicates and simply select all data sources in such cases. LHD uses a symmetric hash join to send subqueries and integrate the results in parallel.

Vocabulary of Interlinked Datasets (VoiD) [4] is published as W3C Semantic Web Interest Group note[1] as meta data format for describing LOD datasets, such that datasets relevant to answer a given query can be selected effectively in an automated manner.

The Web of Data Query Analyzer (WoDQA) [3] eliminates (not relevant) datasets by analyzing a given query with respect to the VoiD descriptions of datasets.

In addition to utilizing VoiD descriptions, SPLENDID [15] forwards SPARQL ASK queries to all data sources when any of the subjects or objects of the currently considered triple pattern is bound, and selects only those sources, which successfully pass this test. SPLENDID optimizes the join order by a dynamic programming strategy.

FedX [37] selects its sources only by sending SPARQL ASK queries and utilizing a cache of the recent results of these SPARQL ASK queries. FedSearch [31] extends FedX by supporting keyword search clauses and introduces optimizations for reducing the communication costs for top-k hybrid searches across multiple data sources.

ANAPSID [2] adapts its query plans to the source availability and runtime conditions of SPARQL endpoints. For this purpose, physical pipelining operators are used to dynamically detect blocked sources and traffic (even if they are not blocked at the beginning and become blocked after some time). ANAPSID utilizes both a catalog and ASK queries and apply heuristics [30] to select the sources.

Graph Distributed SPARQL (GDS) [40] uses the Minimum Spanning Tree (MST) algorithm by exploiting Kruskal algorithm to optimize the execution order of triple patterns and joins. As distributed join approaches, either Semi-Join or Bind Join is applied with a cache to reduce traffic costs.

Avalanche [7] first collects on-line statistical information about the data distribution as well as bandwidth availability. Based on these and other qualitative statistical information it optimizes a given query for quickly providing first answers and executes the query in a distributed manner.

The goal of LDQPS [22] is also to early report results by ranking the sources.

[21] proposes the non-blocking, pushed-based and stream-based Symmetric Index Hash Join (SIHJoin), which is able to process both remote and local linked data. [21] defines also a cost model for this join operator, which is the basis for optimization steps.

Distributed SPARQL [51] introduces the Semi-Join approach for querying SPARQL endpoints.

Contrary to the described approaches, the SPARQL client-server query processor SHEPHERD [1] is tailored to reduce SPARQL endpoint workload and generates shipping plans, where costly operators are placed at the client site by decomposing SPARQL queries into lightweight sub-queries that will be posted against the endpoint.

[6] formalizes federation (and also navigation) in SPARQL 1.1. Furthermore, [6] analyzes some classical theoretical problems such as expressiveness and complexity, and discusses algorithmic properties, like the impossibility of answering some unbounded federated queries.

[19] extends some of the distributed join approaches to process aggregate queries (like minimum, maximum, counting, summation and average computations) on SPARQL endpoints.

[1] http://www.w3.org/TR/void/

Framework	Platform	Join Approach	Cache
Jena/ARQ [41]	Jena	Bind Join, Nested Loop Join	no
Sesame [10]	Sesame	Nested Loop Join	no
DARQ [33]	Jena	Bind Join, Nested Loop Join	yes
ADERIS [27]	-	In-Memory Asymmetric Hash-Join (Note: [27] calls the join approach index nested loop join)	no
FedX [37]	Sesame	Bind Join parallelization (Vectored Evaluation)	yes
FedSearch [31]	Sesame	Bind Join parallelization (Vectored Evaluation), Parallel Competing Rank Join as Modification of Symmetric Hash Join	yes
GRANATUM [18]	Jena	Bind Join, Nested Loop Join	no
LHD [39]	Jena	Bind Join, Symmetric Hash Join	no
Splendid [15]	Sesame	Bind Join, Hash Join	no
GDS [40]	Jena	Bind Join, Semi-Join	yes
Avalanche [7]	Avalanche	Join-At-Endpoint	yes
Distributed SPARQL [51]	Sesame	Semi-Join	no
LDQPS [22]	stream-based query engine of LDQPS	Symmetric Hash Join	no
SIHJoin [21]	stream-based query engine of SIHJoin	Symmetric Hash Join	no
WoDQA [3]	Jena	Bind Join, Nested Loop Join	yes
SemWIQ [25, 24, 23]	Jena	Bind Join	yes
ANAPSID [2]	ANAPSID	Symmetric Hash Join	no

Table 1: Used distributed join approaches in federation frameworks

3 Distributed Join Approaches for SPARQL Endpoints

Different distributed join approaches require different pre- and post-processing steps. Figure 2 provides an overview of the variants of the operator graph for the different distributed join approaches based on the schema of a service clause in Listing 3.

Figure 2 a) presents the operator graph scheme for those distributed join approaches without preprocessing phase, and where the result of the remote query still needs to be joined with the result of applying `A` to the local data. In comparison, variant b) is used in case of distributed join approaches, which take the result of `A` to reformulate the remote query or a set of remote queries sent to the SPARQL endpoint. Furthermore, the result of a remote query must either already contain the results of the join between `A` and `B` or can still be associated to the preprocessed result of `A`, such that necessary join steps can be done within the SERVICE operator. Hence, a succeeding pure join operation is not needed for variant b). In variant c) this is not the case and a succeeding join operation between the results of `A` and the SERVICE operation is required to retrieve correct results.

Table 2 maps different distributed join approaches to their required operator graph scheme. We will discuss the different distributed join approaches in more detail in the following subsections.

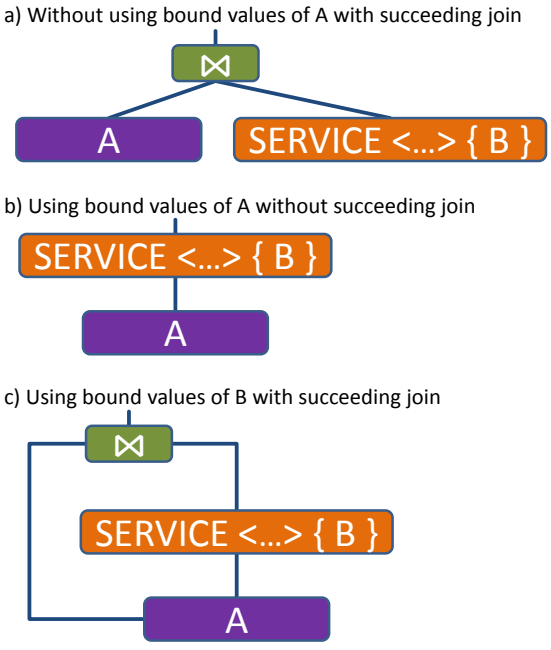

Figure 2: Operator graph schemes of different types of distributed joins

Operator Graph Scheme (according to Figure 2)	Distributed Join
a)	Trivial Approach
b)	Fetch-As-Needed/Bind Join (with/without Cache)
	Vectored Evaluation of Fetch-As-Needed/Bind Join
	Join-At-Endpoint
c)	Semi-Join Approach
	Bitvector-Join Approach
	Value Approach

Table 2: Distributed join approaches and their operator graph scheme

3.1 Trivial Approach

The trivial approach is also called Fetch-All and Ship-Whole [28]. The trivial approach just sends the query demands of B to the SPARQL endpoint (by encapsulating B in a single SELECT query), and the results returned by the SPARQL endpoint are joined with the results of A. For example, the trivial approach sends the remote query in Listing 5 to the SPARQL endpoint for the SERVICE-clause of the query in Listing 4.

```
PREFIX ex: <http://www.ifis.uni-
    luebeck.de/example/>
SELECT * WHERE {
 ?book ex:price ?price .
}
```

Listing 5: The trivial approach sends this remote query for the SERVICE-clause of the query in Listing 4

The trivial approach performs obviously very well if the result of the remote query is small, or nearly all of the results of the remote query have join partners in A (and are without duplicates). The trivial approach has nearly no overhead in these cases.

3.2 Fetch-As-Needed / Bind Join

The Fetch-As-Needed approach [28] is also called Bind join. It has several variants, but all variants fetch for a specific result of A its join partner from the SPARQL endpoint.

3.2.1 Basic Variant

The basic variant generates for each result of A a remote query, where it replaces common variables of A and B with their already bound values in the remote query. The result received from the SPARQL endpoint is then just joined with the currently considered result of A.

Listing 6 presents the remote query of the Fetch-As-Needed approach for the query in Listing 4 and the data in Listing 1. In this remote query (in comparison to the one of the trivial approach in Listing 5), the variable ?book has been replaced with its already bound value ex:book according to the result of A applied on the data in Listing 1. If there would be several results of A, also several remote queries would have been generated and sent to the SPARQL endpoint.

```
PREFIX ex: <http://www.ifis.uni-
    luebeck.de/example/>
SELECT * WHERE {
 ex:book ex:price ?price .
}
```

Listing 6: Remote query sent by the Fetch-As-Needed approach for the query in Listing 4 and the data in Listing 1

3.2.2 Fetch-As-Needed / Bind Join with Cache

This variant utilizes a cache to remember the answers of already sent remote queries, and just takes the cached results for previously considered bound values of common variables of A and B. In this way sending the same remote queries several times is avoided, but comes with the costs of managing a cache.

3.2.3 Vectored Evaluation of Fetch-As-Needed/ Bind Join

The vectored evaluation of a bind join sends only one SPARQL query with UNION clauses containing the original single requests with renamed variables for later post-processing and determination of corresponding intermediate results.

For example, let us assume that the results of A contain the bound values ex:book_i with $i \in \{1, ..., n\}$ of the variable ?book. Then the Vectored Evaluation of Fetch-As-Needed approach sends the remote query of Listing 7 to the SPARQL endpoint. After retrieving the result of the SPARQL endpoint, the SERVICE operator associates the results of A with those of the SPARQL endpoint by just considering which of the variables ?price_i is

bound. As a last step, the associated results must be combined and the variable ?price_i renamed to the original variable name ?price.

```
PREFIX ex: <http://www.ifis.uni-
    luebeck.de/example/>
SELECT * WHERE {
 {ex:book_1 ex:price ?price_1 .}
 UNION
 ...
 UNION
 {ex:book_n ex:price ?price_n .}
}
```

Listing 7: Remote query sent by the Vectored Evaluation of Fetch-As-Needed approach for the query in Listing 4

The Vectored Evaluation of Fetch-As-Needed approach reduces greatly the overhead of sending many queries to the SPARQL endpoint as is the case for the other Fetch-As-Needed variants. However, the sent query becomes significantly larger as well as the post-processing step slightly more complicated.

3.3 Join-At-Endpoint

The Join-At-Endpoint approach sends the results of A within the remote query, and the SPARQL endpoint computes the join result of A and B. For example, let us assume that the results of A contain the results {?book=ex:book_i, ?title="T i", ?author="Ghostwriter i"} with $i \in \{1, ..., n\}$. Then the Join-At-Endpoint approach sends the remote query of Listing 8 to the SPARQL endpoint. The answer returned by the SPARQL endpoint is already the join result of A and B, with which the query evaluation of the client continues.

The remote query of the Join-At-Endpoint approach is the largest of all approaches and the result of the join of A and B (to be sent back to the client) is typically also greater than the result of B (to be sent back to the client in the other approaches), which increases the communication costs. However, whenever the client has low computing resources, or whenever the results of the join must be further processed at the SPARQL endpoint or need to be sent to a third node (being different from the client and SPARQL endpoint), the Join-At-Endpoint approach will be of benefit because of its unique property (among the distributed join approaches) of the join computation at the SPARQL endpoint.

```
PREFIX ex: <http://www.ifis.uni-
    luebeck.de/example/>
SELECT * WHERE {
 ?book ex:price ?price .
 VALUES (?book ?title ?author) {
  (ex:book_1 "T 1" "Ghostwriter 1")
  ...
  (ex:book_n "T n" "Ghostwriter n")
 }
}
```

Listing 8: Remote query sent by the Join-At-Endpoint approach for the query in Listing 4

3.4 Semi-Join Approach

The Semi-Join approach [28] is based on equivalences between join and semi-join [13]:

$$A \bowtie B = A \bowtie (B \ltimes A) = A \bowtie (B \bowtie \pi_J(A))$$

where J is the set of common variables of A and B (and hence J contains the join variables). The Semi-Join approach transmits π_J(A) to the SPARQL endpoint, which filters the results of B regarding π_J(A) to avoid returning results of B, which do not have any join partner in A.

For example, let us assume that the results of A contain the bound values ex:book_i with $i \in \{1, ..., n\}$ of the variable ?book. Then the Semi-Join approach as described in [51] sends the remote query of Listing 9 to the SPARQL endpoint.

```
PREFIX ex: <http://www.ifis.uni-
    luebeck.de/example/>
SELECT * WHERE {
 ?book ex:price ?price .
 FILTER(?book = ex:book_1 ||
          ... || ?book = ex:book_n)
}
```

Listing 9: Remote query sent by the Semi-Join approach for the query in Listing 4

3.5 Value Approach

With SPARQL 1.1 [14] we can formulate the Semi-Join approach for one join variable in a more compact way by testing a variable to be in a set of values. This reduces the size of the query to be sent and many SPARQL evaluators are faster in processing this kind of expression. In

comparison to the Semi-Join approach, where duplicates in the tested values lead to additional comparisons, we avoid generating duplicates in the generated set of values. We call this approach the *Value approach*. In our example, the remote query of Listing 10 is sent to the SPARQL endpoint.

```
PREFIX ex: <http://www.ifis.uni-
    luebeck.de/example/>
SELECT * WHERE {
 ?book ex:price ?price .
 FILTER(?book in
        (ex:book_1, ..., ex:book_n))
}
```

Listing 10: Remote query sent by the Value approach for the query in Listing 4

If there are more than one join variables, the results of the Semi-Join approach and the Value approach may differ, as the Value approach tests each variable independent from each other if their bound value is in the given set. In comparison, the Semi-Join approach considers the values of all bound variables. Hence, the SPARQL endpoint applying the Semi-Join approach returns only results of `B` surely having a join partner in `A`. However, the SPARQL endpoint utilizing the Value approach may return some so called *false drops*: results of `B` not having any join partners in `A`.

Value (X Chars) Approach

The Value approach still sends relative big remote queries. The idea of the *Value (X chars) approach* is to send only X chars of the values instead the complete ones. The challenge is to send those characters, which do not increase the number of false drops. Considering the types of values (IRIs and literals), it seems to be a good choice to send the last characters of the values. Listing 11 contains the remote query of our example for the Value (3 chars) approach ($\mathtt{n} \leq 9$).

3.6 Bitvector-Join

Instead of sending $\pi_J(\mathtt{A})$ to the SPARQL endpoint as the Semi-Join approach does, the *Bitvector-Join* [28] sends a bloom filter in form of a bit vector to the SPARQL endpoint. At the client side, for each value v bound to a join variable of `A`, a fixed hash function h is applied and the bit at position $h(v)$ is set in the bloom filter (initialized in the beginning with bits all unset). After transmitting the

```
PREFIX ex: <http://www.ifis.uni-
    luebeck.de/example/>
SELECT * WHERE {
 ?book ex:price ?price .
 FILTER(substr(str(?book),strlen(str
     (?book))-2,3) in
         ("k_1", ..., "k_n"))
}
```

Listing 11: Remote query sent by the Value (3 chars) approach for the query in Listing 4

bloom filter the SPARQL endpoint checks for each result of `B` if the corresponding bit is set in the bloom filter by using the same hash function as the client on the join variables. Only those results of `B` are returned from the SPARQL endpoint, which pass the bloom filter check. This reduces greatly the number of sent bytes, but again some false drops may occur.

However, the Bitvector-Join cannot be directly translated to W3C conform remote SPARQL queries.

3.6.1 NonStandard Approach

We introduce the *NonStandard approach* to test how a slight extension of the SPARQL language enables us to support Bitvector-Joins [28]. For this purpose, we only need an additional built-in function `BitVectorFilter(?v,b,s)`, which returns true if the bit at position $h(\mathtt{?v})$ is set in the bloom filter `b` with the size `s`. The hash function h needs to be a fixed one (like in our implementation, where we use the Java standard hash function on objects), or additional parameters could be given in the built-in function to describe the hash function to be used.

In our example, the remote SPARQL query looks like the query in Listing 12. Obviously the size of the remote query is independent from the number of results of `A`, which is one of the advantages of the NonStandard approach.

3.6.2 W3C Conform Approach

Considering the SPARQL 1.1 [14] specification, a set of hash functions MD5, SHA1, SHA256, SHA384 and SHA512 is already specified, which return the checksum of these hash functions (as a hex digit string) calculated on the UTF-8 representation of the given parameter values. However, we do not have the possibility to form a bloom filter as a bit vector from the checksums of join variable values of `A`. The idea is now to use the check-

```
PREFIX ex: <http://www.ifis.uni-luebeck.de/example/>
SELECT * WHERE {
 ?book ex:price ?price .
 Filter(ex:BitVectorFilter(?book,28,5) ).
}
```

Listing 12: Remote query sent by the NonStandard approach for the query in Listing 4

sums directly to filter out irrelevant results of B at the SPARQL endpoint. However, the checksums are quite long, often longer than the original values. Hence, we propose to use only some characters of the checksums instead all in order to reduce the size of the remote query. Listing 13 presents the remote query for an example with three results of A and for the MD5 hash function.

In comparison to the Value (X chars) approach the advantage of the W3C conform approach is that checksums are quite irregular even for small changes of the input. Hence, we can reduce the number of false drops in those scenarios, where the values of the join variables of A do not differ much. However, it comes with the costs of an additional calculation of the checksums. Our experiments in the next section show the advantages of this approach especially in real-world scenarios.

```
PREFIX ex: <http://www.ifis.uni-luebeck.de/example/>
SELECT * WHERE {
 ?book ex:price ?price .
 Filter(substr(MD5(str(?book)),1,2)
     in ("7f","b1","19"))
}
```

Listing 13: Remote query sent by the W3C Conform approach (here MD5) for the query in Listing 4

4 Experimental Evaluation

We have run two different types of experiments. The first type of experiments uses synthetic datasets in order to have configurable properties of the input data and to test them. The second type of experiments runs real-world data in order to show common results with synthetic datasets and differences to real-world scenarios.

We describe the used underlying Semantic Web framework in Section 4.1, the experimental environment in Section 4.2, the used query, datasets and the results for the synthetic datasets in Section 4.3 and for the real-world scenario in Section 4.4, and finally a comprehensive analysis in Section 4.5.

4.1 LUPOSDATE

LUPOSDATE [16] is an open source Semantic Web database which uses different types of indices for large-scale datasets (disk based) and for medium-scale datasets (memory based) as well as processing of (possibly infinite) data streams. LUPOSDATE supports the RDF query language SPARQL 1.1, the rule language RIF BLD, the ontology languages RDFS and OWL2RL, parts of geosparql and stsparql, visual editing of SPARQL queries, RIF rules and RDF data, and visual representation of query execution plans (operator graphs), optimization and evaluation steps, and abstract syntax trees of parsed queries and rules. The advantages of LUPOSDATE are the easy way of extending LUPOSDATE, its flexibility in configuration (e.g. for index structures, using or not using a dictionary, ...) and it is open source [17]. These advantages make LUPOSDATE best suited for any extensions for scientific research.

We have integrated all the approaches described in this paper and use LUPOSDATE as experimental platform to evaluate their performances.

4.2 Experimental Setup

We have used two computers in the experiments: One runs the client and the other a SPARQL endpoint. For the experiments with synthetic data we used a symmetric configuration. The operating system of both computers for the experiments with synthetic datasets is Windows 7 Professional running on 4 Gbyte RAM and an Intel(R) Core(TM) 2 Duo CPU E6550 @ 2,33 GHz. The client and the SPARQL endpoint run a Java 1.8 virtual machine. Both computers are connected via 1 Gigabit/s LAN.

For the experiments running real-world data, we have used another computer with higher computing resources for the SPARQL endpoint, but the client is still running on the same computer (with a configuration as described above). We believe that this asymmetric configuration is more typical in real-world. The hardware and software configuration of the SPARQL endpoint for the experiments with real-world data consists of an Intel Xeon X5550 2 Quad CPU computer, each with 2.66 Gigahertz,

72 Gigabytes main memory, Windows 7 (64 bit) and Java 1.8.

We have run each query on their respective synthetic datasets 1000 times and on real-world data 20 times and present the average execution times in our figures.

4.3 Experiments with Synthetic Datasets

In the experiments with synthetic datasets, we use a query, which combines local data (residing at the client's computer) with remote data (residing at the SPARQL endpoint) over a simple join between two triple patterns. In more detail, we require the object of the local data to be the same as the subject of the remote data in the joined result (see Listing 14).

```
SELECT * WHERE {
  ?ls ?lp ?c .
  SERVICE <endpoint-url> {
    ?c ?rp ?ro .
  }
}
```

Listing 14: Query used in experiments with synthetic datasets

We developed data generators for the local and the remote data, which are especially designed for the used query. While the remote data (see Listing 15) is relatively simple and just has exactly one join partner for each object of the local data, the local data (see Listing 16) may consist of up to n triples with the same object (but different subjects and predicates). In this way we can analyze caching effects of the different approaches.

```
@prefix p:<http://www.ifis.uni-
    luebeck.de/semantic_web/sparql/
    endpoint/optimization/
    distributedjoin#>.
p:c0 p:rp0 p:ro0.
p:c1 p:rp1 p:ro1.
...
```

Listing 15: Scheme of remote data used in experiments with synthetic datasets

In the experiments, we used datasets with 1, 10 and 100 different objects without duplicates of the objects (i.e., $n = 1$). Additionally, we used datasets with 100 different objects and 2, 3 and 4 duplicates of each object (i.e., $n \in \{2, 3, 4\}$). For the remote data, we have used

```
@prefix p:<http://www.ifis.uni-
    luebeck.de/semantic_web/sparql/
    endpoint/optimization/
    distributedjoin#>.
p:s0_0 p:p0_0 p:c0.
...
p:s0_n p:p0_n p:c0.

p:s1_0 p:p1_0 p:c1.
...
p:s1_n p:p1_n p:c1.
...
```

Listing 16: Scheme of local data used in experiments with synthetic datasets

a dataset with 1000 triples reflecting the typical case that the remote data is larger than the local data.

Results of Experiments on Synthetic Datasets

We have run and tested all the different approaches described in this paper. The requests for the dataset of 100 triples to be joined with 4 duplicates go beyond the limit of SPARQL query lengths of the system for the Semi-Join approach as well as the Join-At-Endpoint approach. Hence, we cannot present the numbers for these approaches for the mentioned dataset, but for all the other approaches the query lengths are smaller and remain in the limit of the system.

Figure 3 presents the execution times for the different approaches applied to our different datasets. For the analysis it is also important to know how many bytes are sent to the SPARQL endpoint (see Figure 4), how many bytes are sent from the SPARQL endpoint back to the client (see Figure 5), and its total sum (see Figure 6).

Furthermore, we present in Figure 7 the execution times per transmitted byte (computed by the execution times divided by the sum of bytes sent and received from client over network). The larger this number is, the more time is needed for calculations besides transmitting the endpoint queries and their results. The reason for a large number is a high execution time in relation to a small size of transmitted bytes.

4.4 Experiments with Real-World Data

The goal of DBpedia[2] is to extract semantic data from Wikipedia and offer this data publicly freely available as Linked Open Data (LOD) [26]. In fact DBpedia is the

[2] http://wiki.dbpedia.org/

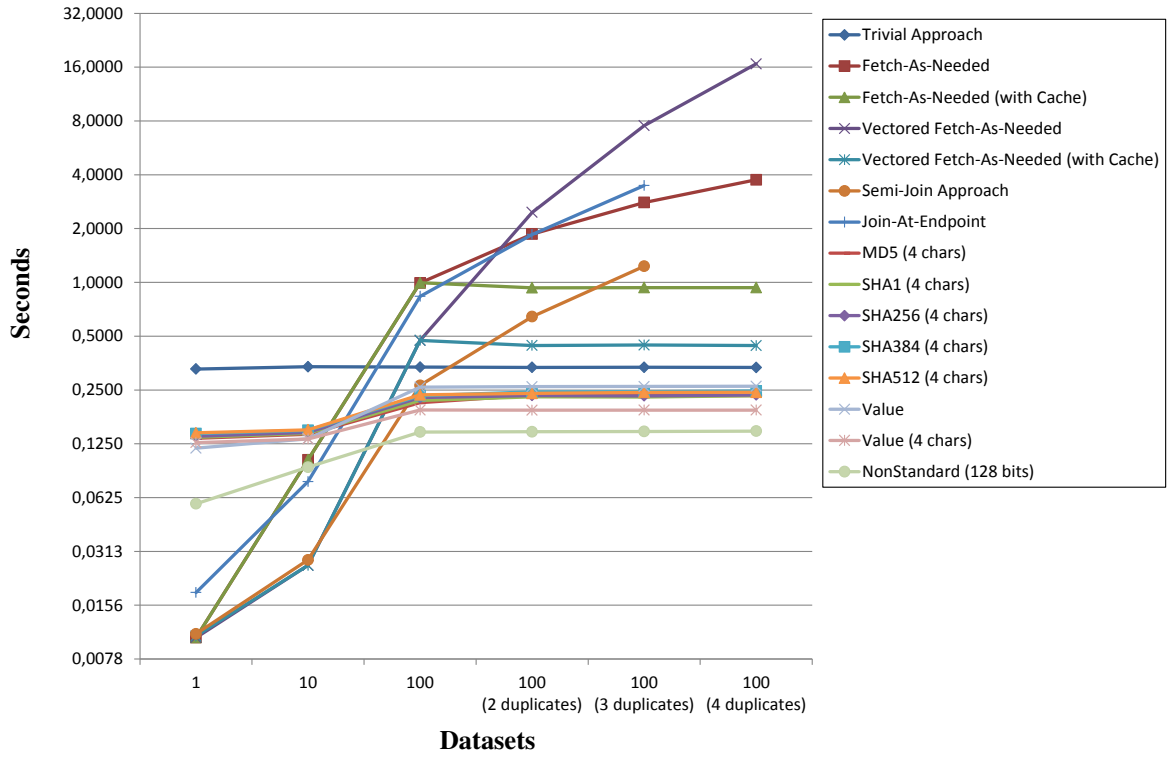

Figure 3: Execution times of different types of distributed joins in seconds for query on synthetic data

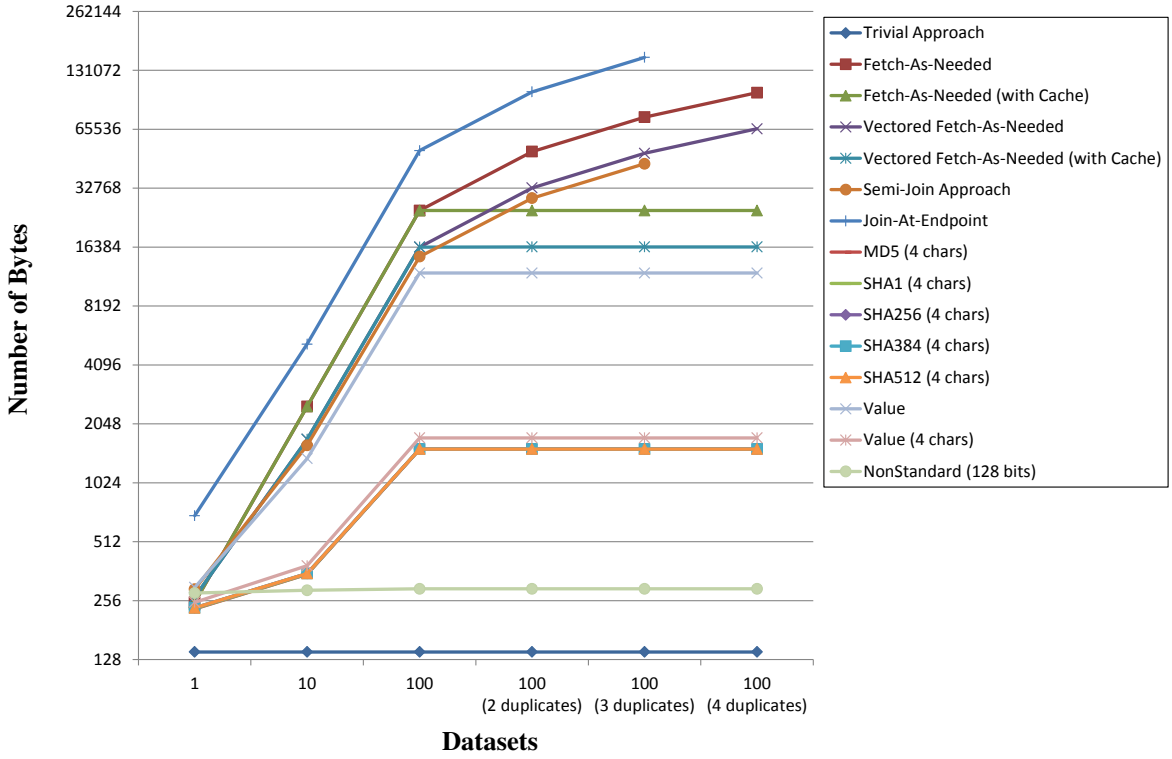

Figure 4: Bytes sent from client over network of different types of distributed joins for query on synthetic data

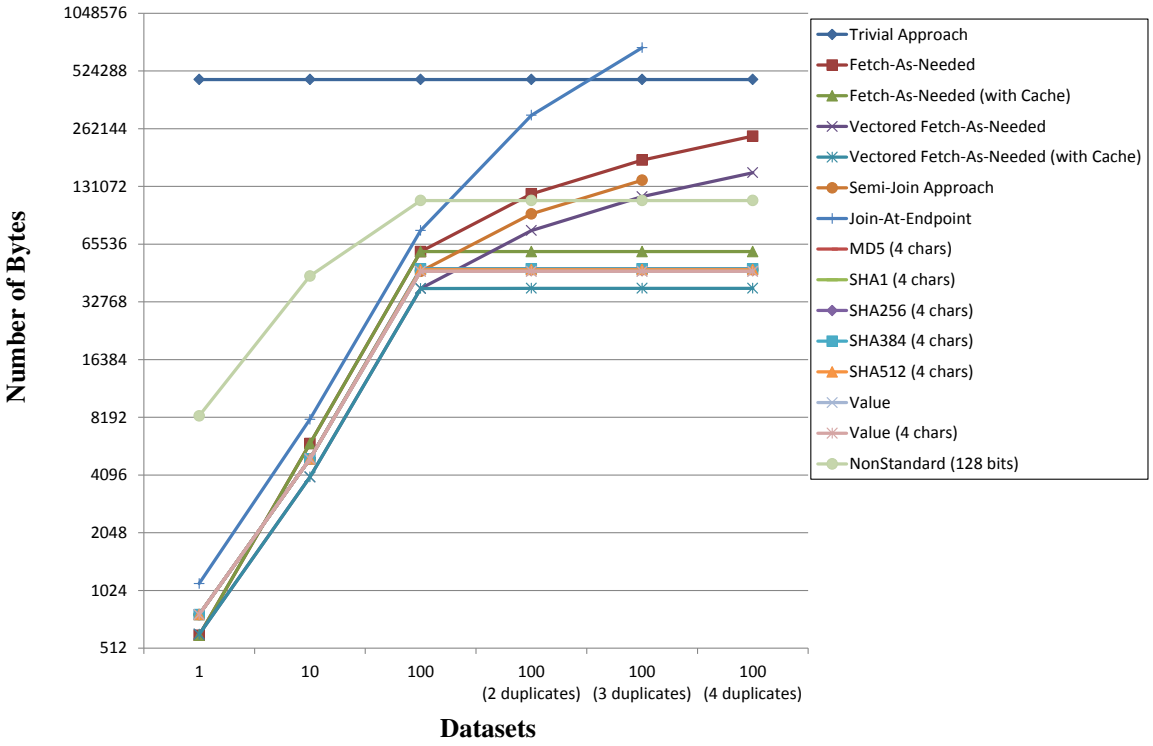

Figure 5: Bytes received from client over network of different types of distributed joins for query on synthetic data

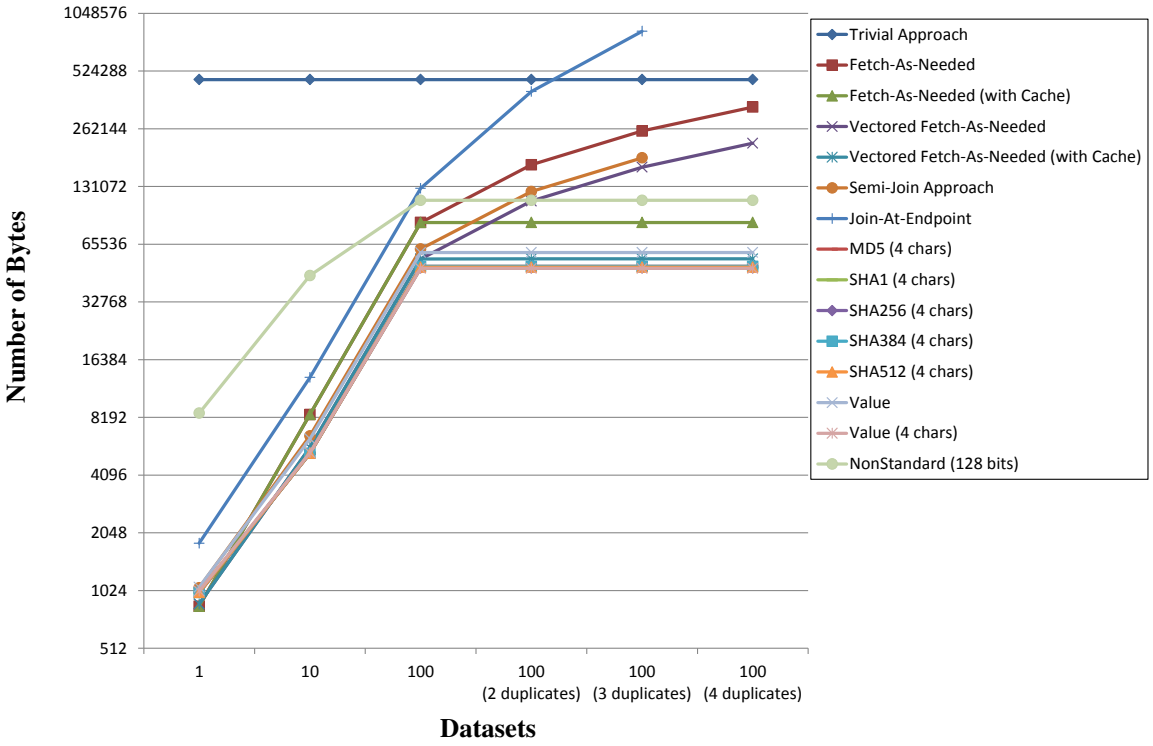

Figure 6: Sum of bytes sent and received from client over network of different types of distributed joins for query on synthetic data

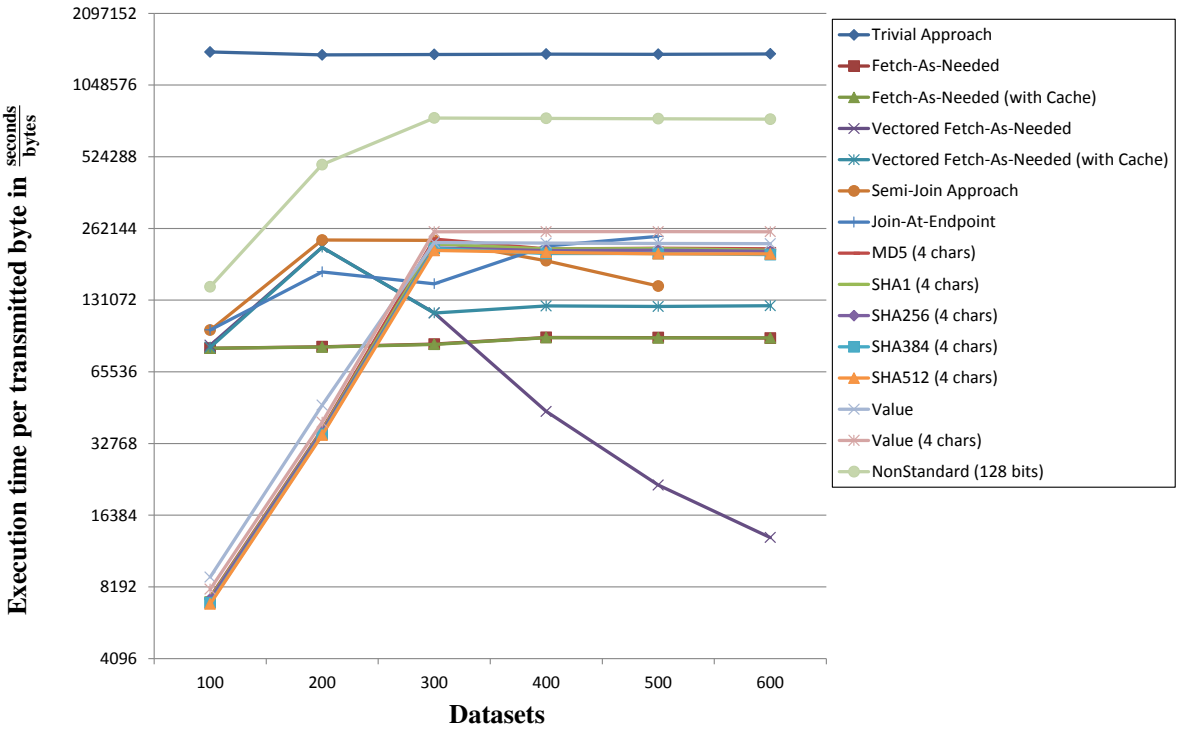

Figure 7: Execution times per (sent and received) byte over network of different types of distributed joins for query on synthetic data

central dataset in the LOD cloud to which most other datasets in LOD are linked. Hence users can ask sophisticated queries against DBpedia datasets to get the information available in Wikipedia (and users may combine the information in Wikipedia with that of other linked datasets).

```
PREFIX ont: <http://dbpedia.org/
    ontology/>
PREFIX rdf: <http://www.w3.org
    /1999/02/22-rdf-syntax-ns#>

CONSTRUCT {
 ?s rdf:type ont:Station.
} WHERE {
 ?s rdf:type ont:Station.
}
```

Listing 17: Query for constructing the local data in experiments with DBpedia datasets

For the real-world data in a larger setting, we have imported the DBPedia datasets *Mapping-based Types* and *Mapping-based Properties*, which were extracted from Wikipedia dumps generated in February / March 2015[3]. These datasets contain 37,666,266 triples.

For generating the local data we have run once the construct query of Listing 17, which generates data about railway stations described in wikipedia.

```
PREFIX ont: <http://dbpedia.org/
    ontology/>
PREFIX rdf: <http://www.w3.org
    /1999/02/22-rdf-syntax-ns#>

SELECT * WHERE {
 ?s rdf:type ont:Station.
 SERVICE <endpoint-url> {
     ?s rdf:type ont:Station.
     ?s ont:address ?address .
     ?s ont:location ?location .
     ?location ?p ?o .
 }
}
```

Listing 18: Query used in experiments with DBpedia datasets

In the experiments, we then run the query in Listing 18 on the local data (generated as described before) with datasets of sizes 100, 200, 300, 400, 500 and 600 tripels

[3] http://wiki.dbpedia.org/Downloads2015-04

(containing the same number of different stations). This query asks for the address and location of the railway stations (of the local data) as well as all information about the locations available.

Results of Experiments on Real-World Data

This time we have used bigger local datasets. All approaches, except of the trivial one and the Fetch-As-Needed approaches with and without cache, are adapted to send several queries and set up requests to the SPARQL endpoint block-wise (according to blocks of local data) and thus avoiding exceeding the maximum number of characters, which the query parser can process. In more detail, a query is sent for each 300 join partners of the local data. Hence the client sends 2 queries for the local datasets with sizes 400, 500 and 600 triples. In more sophisticated implementations, the client could integrate as many join partners in the query as the query size fits into the parser's limits. In this way approaches generating smaller query sizes would even benefit much more than in the currently implemented approach (generating a query for each 300 join partners), where the benefit is only based on smaller communication costs (and maybe smaller computation costs of the queries) instead of benefits based on avoiding query requests to the endpoint. Although we would need only one query for the NonStandard approach, we also send two queries for larger local datasets in our experiments. This is because the bit vector size needs to be increased otherwise (leading to higher computation costs) when trying to avoid many false drops. The experiments do not show an irregular increase of execution times in our real-world scenario for the local datasets larger than 300 triples in comparison to the smaller ones.

Figure 8 presents the execution times for the different approaches applied to our different local datasets and the DBpedia data on the endpoint. Furthermore, we present also how many bytes are sent to the SPARQL endpoint (see Figure 9), how many bytes are sent from the SPARQL endpoint back to the client (see Figure 10), its total sum (see Figure 11) and the execution times per transmitted byte (see Figure 12).

4.5 Analysis

We group our analysis results according to the following criteria:

4.5.1 Duplicate-Sensitive versus Duplicate-Insensitive Approaches

Some approaches are especially optimized for handling duplicates (like Fetch-As-Needed with Cache and Vectored Fetch-As-Needed with Cache), and avoiding extra costs is in the nature of other approaches like the Bitvector-Join approaches (MD5, SHA-x with $x \in \{1, 256, 384, 512\}$), Value, Value (4 chars), NonStandard and the trivial approach. For all these approaches, there are only few differences in the execution times, as only the client has to process the duplicates locally before joining (except of the trivial approach) for generating the query sent to the SPARQL endpoint, and during joining (having also duplicated entries in the join result). The main reason for nearly the same execution times is that the same amount of bytes are sent from and received by the client not depending on the number of duplicates.

Duplicate-insensitive approaches like Fetch-As-Needed, Vectored Fetch-As-Needed, Semi-Join approach and Join-At-Endpoint increase the number of bytes sent and received linearly to the number of duplicates. Considering this fact it is not surprising that the execution times are also linearly increasing with the number of duplicates.

4.5.2 Trivial Approach versus other Approaches

The trivial approach is an exception: The number of sent bytes is very low as the query sent to the SPARQL endpoint does not depend on the local data. Hence, the query length is always the same for all datasets. Thus, the result sent back from the SPARQL endpoint is also always the same. The execution times mainly depend on the size of this result: If it is small, the trivial approach can be one of the fastest. Also if the local data is in such a manner that the result set cannot be reduced (or not much reduced) when using the other approaches, the trivial approach is one of the best by avoiding the overhead of the other approaches.

The other approaches try to reduce the result of the SPARQL endpoint. For this purpose, they have to send larger queries (than the query of the trivial approach) containing information of the local data to the SPARQL endpoint. The main factors are here how large these queries are and how well they can reduce the result compared to the absolute minimum.

4.5.3 Join-At-Endpoint versus other Approaches

Join-At-Endpoint is an approach with extraordinary properties: The query sent to the SPARQL endpoint must contain all information of the client to process the join of the local with the remote data. Hence, the sent queries are the largest of all approaches for the synthetic dataset. The computation costs are also relatively expensive, as one can suppose by looking at the execution times per transmitted byte (see Figures 7 and 12). Furthermore, if

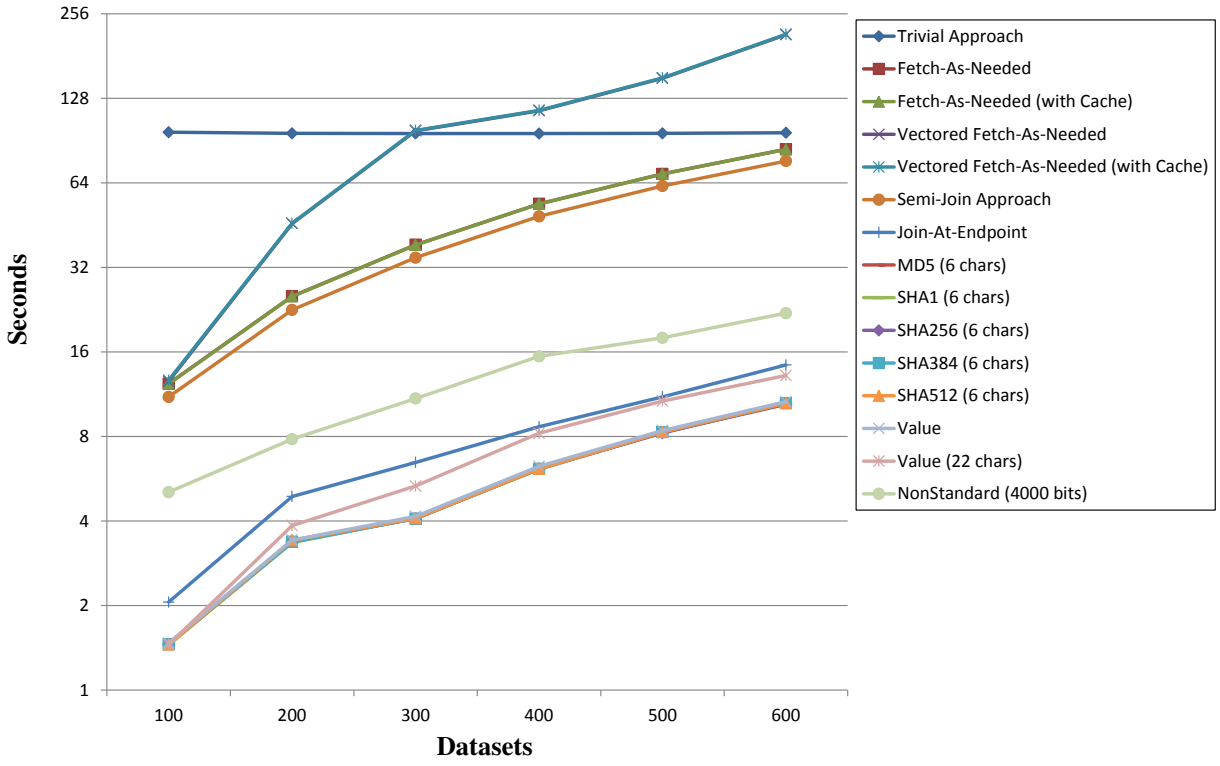

Figure 8: Execution times of different types of distributed joins in seconds for DBpedia query

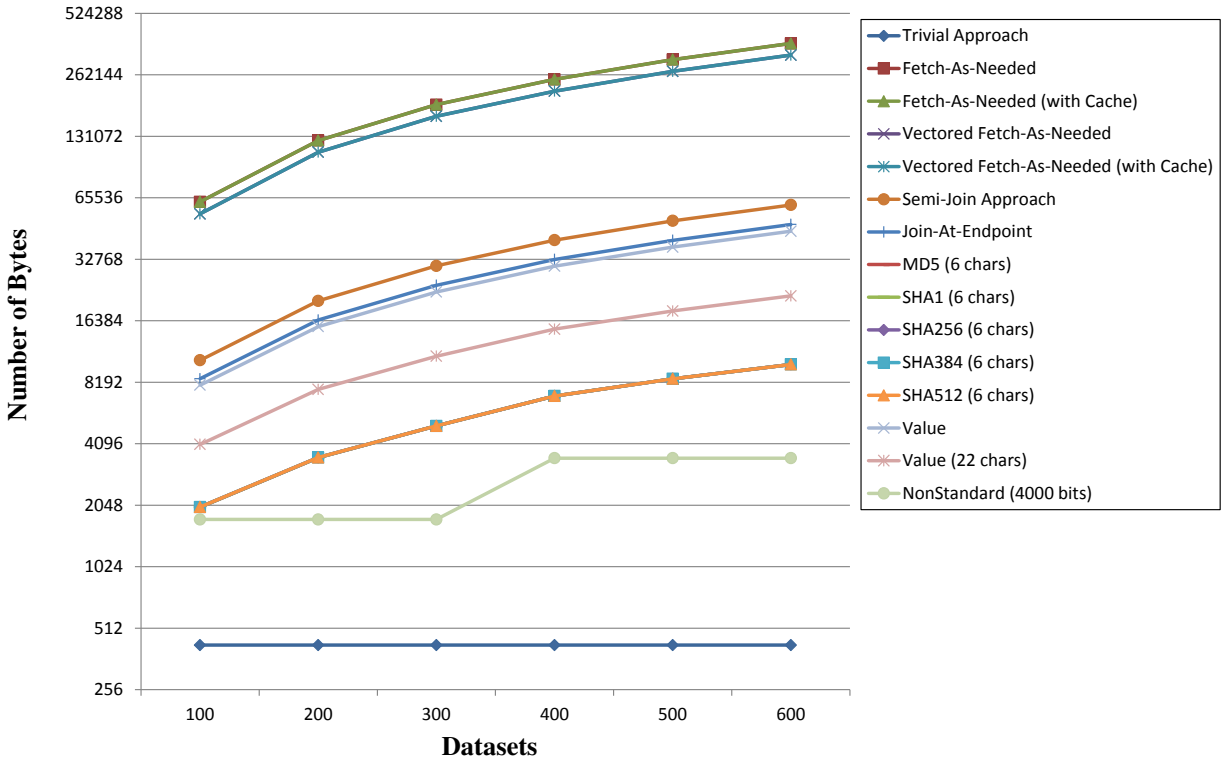

Figure 9: Bytes sent from client over network of different types of distributed joins for DBpedia query

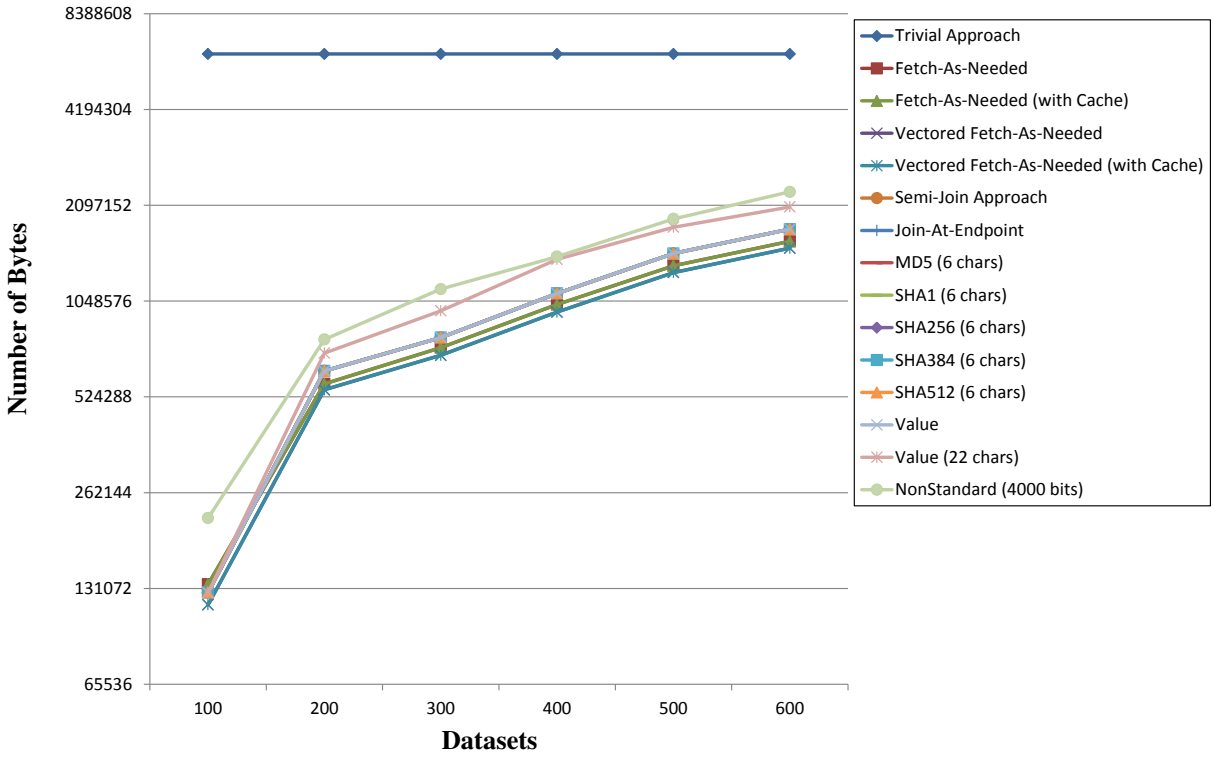

Figure 10: Bytes received from client over network of different types of distributed joins for DBpedia query

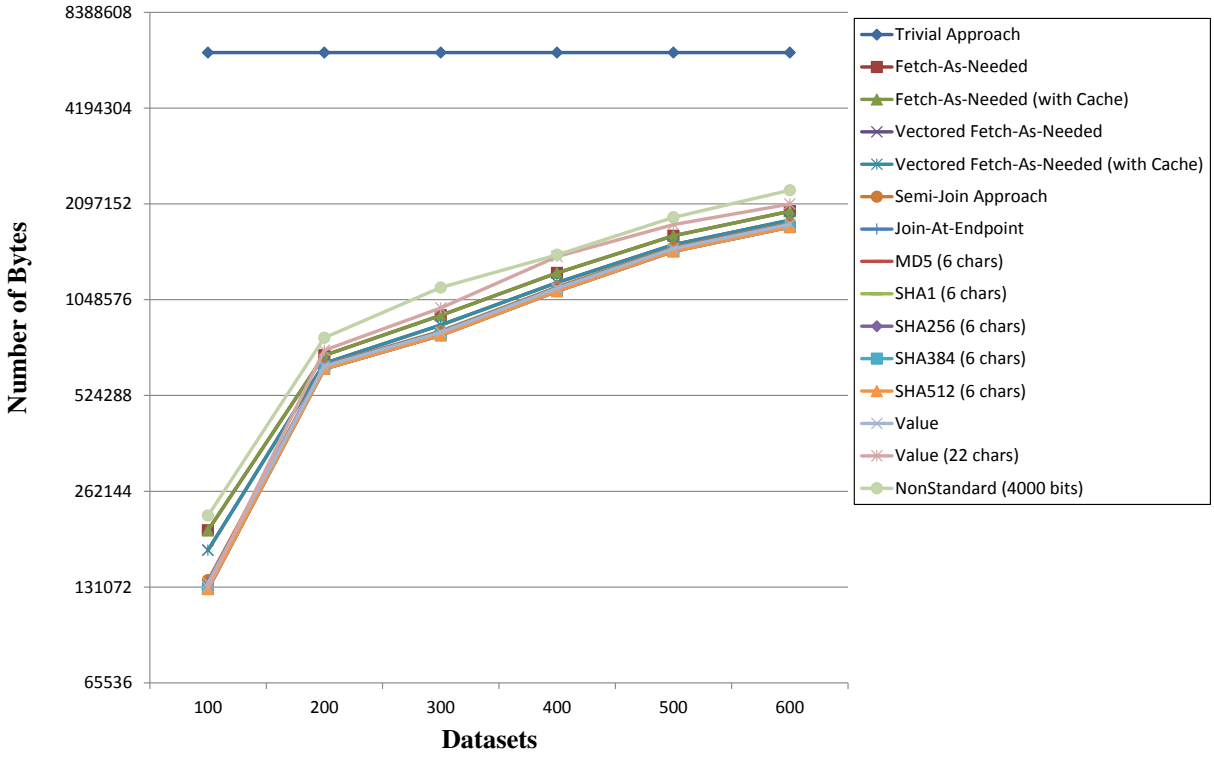

Figure 11: Sum of bytes sent and received from client over network of different types of distributed joins for DBpedia query

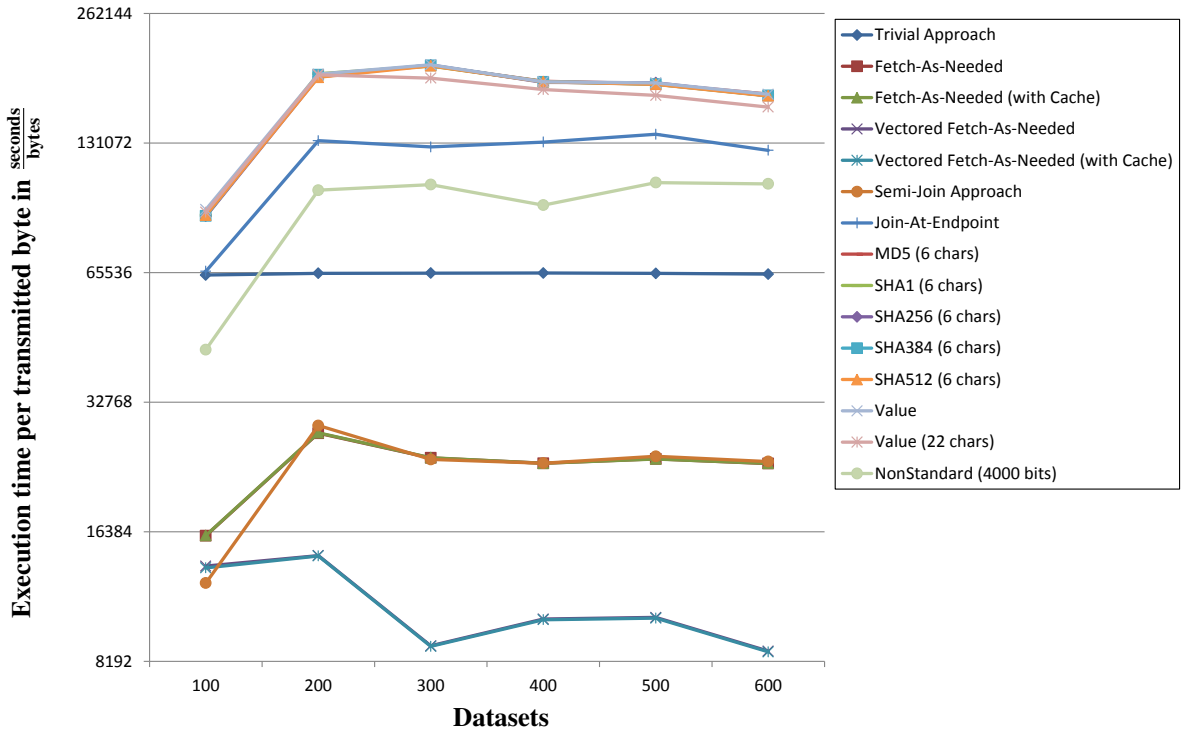

Figure 12: Execution times per (sent and received) byte over network of different types of distributed joins for DBpedia query

the join result increases in comparison to the result sent back by the other approaches, which is the case for duplicates, the communication costs dramatically increase leading to a bad performance.

Overall it seems that the Join-At-Endpoint approach will only have benefits in the scenarios where the SPARQL endpoint has much more computing resources than the client, or where the result must be anyway sent to another host.

4.5.4 Scenarios with few Join Partners

Except of the trivial approach for reasons already discussed in Section 4.5.2, all approaches have very small total communication costs (as sum of bytes sent from and received by the client) for few join partners. The communication costs do not seem to be the main factor for the Bitvector-Joins (except of NonStandard), Value and Value (4 chars) approaches. Their execution times are much higher than the Fetch-As-Needed approaches as well as Join-At-Endpoint, because they have high computation costs for scanning relevant intermediate results and filtering. In comparison, the SPARQL endpoint can fully utilize existing indices for the Fetch-As-Needed approaches as well as Join-At-Endpoint avoiding scans on data to be filtered out in succeeding steps. Specialized indices for e.g. the hashes used in Bitvector-Joins would greatly improve execution times for the Bitvector-Joins, which could be a task for future work.

Surprisingly, the Vectored Fetch-As-Needed approaches (with and without cache) are the slowest for many join-partners in the real-world scenario. The reason is obviously high communication costs, which are not much smaller than those of the Fetch-As-Needed approaches, for sending large queries (as many triple patterns must be repeated). There could be also high computation costs of the complex queries with many union operations of subqueries for each join partner. Especially the number of subqueries could significantly increase the computation costs, as many operations need to be done for each subquery. However, the execution times per transmitted byte (see Figure 12) disproves this hypothesis.

4.5.5 Bitvector-Joins using only W3C conform SPARQL Constructs versus using User-Defined Functions

Among the Bitvector-Joins applying a user-defined function as in the NonStandard approach has the best performance for the synthetic data (but not for the real-world data). The advantage of the user-defined function is its constant size of the sent query independent of the size

of the local data (which is not the case for the other, W3C conform, Bitvector-Join approaches). The computation costs are much lower for the NonStandard approach, as not so many string operations need to be done as well as set operations are avoided. In comparison to the W3C conform Bitvector-Join approaches the NonStandard approach leads to more false drops for those bitvector sizes with optimal performance, which is reflected in the bytes received (and hence also by the execution times per transmitted byte presented in Figures 7 and 12). This seems to be the main factor why the NonStandard approach is slower for the real-world scenario. For the synthetic datasets due to the low computation costs, the NonStandard approach is the fastest.

The Value (4 chars) has lower computation costs in comparison to the Bitvector-Join approaches for the synthetic datasets, because the determination of the hash value is avoided, while often having approximately the same communication costs. The Value approach has much higher communication costs for the synthetic datasets because of the high sizes of the sent queries.

However, for the real-world scenario we need to compare 22 chars (instead of only 4 as for the synthetic data) for best performance, but we still have a significant number of false drops leading to higher communication costs. These higher communication costs as well as the false drops are the main reasons why the Bitvector-Join approaches (except of the Nonstandard approach) as well as even the Value approach are faster in the real-world scenario. The Bitvector-Join and the Value approaches have a high number for the execution times per transmitted byte (see Figure 12) because of their low numbers of bytes sent and received from the client. Or in other words: These approaches spend their high computation costs per transmitted byte for drastically reducing the transmission costs.

4.5.6 Overall Ranking of the Distributed Join Approaches

The winner approaches are variants of the Bitvector-Join approaches except for *few join partners*, where the Fetch-As-Needed variants are the best followed by Semi-Join and Join-At-Endpoint approaches, and except for *large addressed data in the SPARQL endpoint*, where the trivial approach offers simple computations and low overhead.

For *W3C conform SPARQL endpoints* and more join partners, the Bitvector-Join, Value and Value (X chars) approaches are the fastest. Depending on the properties of the applied datasets the ranking among these approaches differ.

For *SPARQL endpoints with support of additional hash function* the NonStandard approach is not always the fastest, as the real-world scenario demonstrates. However, we still believe that the performance in federated environments could greatly benefit from slight extensions of the W3C recommendations of SPARQL [14]. We hope that our research can provide an impulse for future recommendations.

5 Summary and Conclusions

Whenever large datasets residing at different locations need to be combined, intelligent ways to reduce communication costs are the key to improve overall performance. For this purpose, distributed join approaches have been developed. Traditional distributed join approaches need to be checked for their application in publicly freely available SPARQL endpoints. As many SPARQL endpoints follow the SPARQL language specification, real-world realizations have to utilize some of the numerous existing features of this specification for advanced approaches on the one side, but missing features in this specification also limit real-world realizations on the other side. However, our contribution shows that many traditional distributed join approaches can be formulated as SPARQL queries, but some need to be altered (like the Bitvector-Join approaches), and we also develop new variants like the Value and Value (X chars) approaches.

Our experimental analysis demonstrates the advantages and disadvantages regarding the overall execution times as well as transferred bytes. In our experiments the Bitvector-Join approaches (adapted to the possibilities the SPARQL specification offers), the Value and Value (X chars) approaches perform best for W3C conform SPARQL endpoints depending on the properties of the used datasets. We also show that only slight extensions of the SPARQL specification would allow advanced types of distributed join approaches like the proposed NonStandard approach.

References

[1] M. Acosta, M. Vidal, F. Flöck, S. Castillo, C. B. Aranda, and A. Harth, "SHEPHERD: A Shipping-Based Query Processor to Enhance SPARQL Endpoint Performance," in *Proceedings of the ISWC 2014 Posters & Demonstrations Track a track within the 13th International Semantic Web Conference, ISWC 2014, Riva del Garda, Italy, October 21, 2014.*, 2014, pp. 453–456. [Online]. Available: http://ceur-ws.org/Vol-1272/paper_147.pdf

[2] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus, "ANAPSID: An Adaptive Query

Processing Engine for SPARQL Endpoints," in *The Semantic Web ISWC 2011*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, vol. 7031, pp. 18–34. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-25073-6_2

[3] Z. Akar, T. G. Hala, E. E. Ekinci, and O. Dikenelli, "Querying the Web of Interlinked Datasets using VOID Descriptions," in *Linked Data on the Web (LDOW2012)*, 2012.

[4] K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao, "Describing linked datasets - on the design and usage of void, the 'vocabulary of interlinked datasets'." in *WWW 2009 Workshop: Linked Data on the Web (LDOW2009)*, Madrid, Spain, 2009.

[5] M. Ali and A. Mileo, "How Good Is Your SPARQL Endpoint?" in *On the Move to Meaningful Internet Systems: OTM 2014 Conferences*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, vol. 8841, pp. 491–508. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-45563-0_29

[6] M. Arenas and J. Prez, "Federation and navigation in sparql 1.1," in *Reasoning Web. Semantic Technologies for Advanced Query Answering*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7487, pp. 78–111. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33158-9_3

[7] C. Basca and A. Bernstein, "Avalanche: putting the spirit of the web back into semantic web querying," in *Proceedings Of The 6th International Workshop On Scalable Semantic Web Knowledge Base Systems (SSWS2010)*, A. Fokoue, T. Liebig, and Y. Guo, Eds., November 2010, pp. 64–79. [Online]. Available: http://dx.doi.org/10.5167/uzh-44857

[8] T. Berners-Lee and D. Connolly, "Notation3 (N3): A readable RDF syntax," W3C, W3C Team Submission, 2008. [Online]. Available: http://www.w3.org/TeamSubmission/n3/

[9] D. Brickley and R. V. Guha, *RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recommendation*. W3C Recommendation, 2004, available at http://www.w3.org/TR/rdf-schema/.

[10] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: A generic architecture for storing and querying rdf and rdf schema," in *The Semantic Web ISWC 2002*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, vol. 2342, pp. 54–68. [Online]. Available: http://dx.doi.org/10.1007/3-540-48005-6_7

[11] C. Buil-Aranda, A. Hogan, J. Umbrich, and P.-Y. Vandenbussche, "SPARQL Web-Querying Infrastructure: Ready for Action?" in *The Semantic Web ISWC 2013*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 8219, pp. 277–293. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41338-4_18

[12] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[13] T. Connolly and C. Begg, *Database Systems A Practical Approach to Design, Implementation, and Management*. Addison-Wesley, 2005.

[14] S. H. Garlik, A. Seaborne, and E. Prud'hommeaux, *SPARQL 1.1 Query Language*. W3C Recommendation, 2013, available at http://www.w3.org/TR/sparql11-query/.

[15] O. Görlitz and S. Staab, "SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions," in *Proceedings of the Second International Workshop on Consuming Linked Data (COLD2011), Bonn, Germany, October 23, 2011*, 2011. [Online]. Available: http://ceur-ws.org/Vol-782/GoerlitzAndStaab_COLD2011.pdf

[16] S. Groppe, *Data Management and Query Processing in Semantic Web Databases*. Springer, May 2011.

[17] S. Groppe, "LUPOSDATE Semantic Web Database Management System," https://github.com/luposdate/luposdate, 2015.

[18] A. Hasnain, S. Sana e Zainab, M. Kamdar, Q. Mehmood, J. Warren, ClaudeN., Q. Fatimah, H. Deus, M. Mehdi, and S. Decker, "A Roadmap for Navigating the Life Sciences Linked Open Data Cloud," in *Semantic Technology*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2015, vol. 8943, pp. 97–112. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-15615-6_8

[19] D. Ibragimov, K. Hose, T. Pedersen, and E. Zimnyi, "Processing Aggregate Queries in a Federation of SPARQL Endpoints," in *The Semantic Web. Latest Advances and New Domains*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2015, vol. 9088, pp. 269–285. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-18818-8_17

[20] International Organization for Standardization (ISO), *Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)*. ISO/IEC 9075-2:2011, 2011, available

at http://www.iso.org/iso/catalogue_detail.htm?csnumber=53682.

[21] G. Ladwig and T. Tran, "SIHJoin: Querying Remote and Local Linked Data," in *The Semantic Web: Research and Applications*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, vol. 6643, pp. 139–153. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21034-1_10

[22] G. Ladwig and T. Tran, "Linked Data Query Processing Strategies," in *The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I*, 2010, pp. 453–469. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17746-0_29

[23] A. Langegger and T. L, "SemWIQ," http://sourceforge.net/projects/semwiq/, 2013.

[24] A. Langegger and W. Wöß, "SemWIQ - Semantic Web Integrator and Query Engine," in *GI Jahrestagung (2)*, ser. LNI, vol. 134. GI, 2008, pp. 718–722. [Online]. Available: http://subs.emis.de/LNI/Proceedings/Proceedings134/article2173.html

[25] A. Langegger, W. Wöß, and M. Blöchl, "A Semantic Web Middleware for Virtual Data Integration on the Web," in *The Semantic Web: Research and Applications*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, vol. 5021, pp. 493–507. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68234-9_37

[26] Linked Data, "Linked Data - Connect Distributed Data across the Web," 2015. [Online]. Available: http://www.linkeddata.org

[27] S. Lynden, I. Kojima, A. Matono, and Y. Tanimura, "Adaptive Integration of Distributed Semantic Web Data," in *Databases in Networked Information Systems*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, vol. 5999, pp. 174–193. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12038-1_12

[28] L. F. Mackert and G. M. Lohman, "R* optimizer validation and performance evaluation for distributed queries," in *Proceedings of the 12th International Conference on Very Large Data Bases*, ser. VLDB '86. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 149–159. [Online]. Available: http://dl.acm.org/citation.cfm?id=645913.671480

[29] Microsoft, "OPENROWSET (Transact-SQL)," 2015, accessed on 16.11.2015. [Online]. Available: https://msdn.microsoft.com/de-de/library/ms190312(v=sql.120).aspx

[30] G. Montoya, M.-E. Vidal, and M. Acosta, "A heuristic-based approach for planning federated sparql queries." *3rd International Workshop on Consuming Linked Data (COLD 2012) in CEUR Workshop Proceedings*, vol. 905, 2012.

[31] A. Nikolov, A. Schwarte, and C. Htter, "FedSearch: Efficiently Combining Structured Queries and Full-Text Search in a SPARQL Federation," in *The Semantic Web ISWC 2013*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 8218, pp. 427–443. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41335-3_27

[32] Oracle, "15.8 The FEDERATED Storage Engine," 2015, accessed on 16.11.2015. [Online]. Available: http://dev.mysql.com/doc/refman/5.7/en/federated-storage-engine.html

[33] B. Quilitz and U. Leser, "Querying Distributed RDF Data Sources with SPARQL," in *Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications*, ser. ESWC'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 524–538. [Online]. Available: http://dl.acm.org/citation.cfm?id=1789394.1789443

[34] N. Rakhmawati, J. Umbrich, M. Karnstedt, A. Hasnain, and M. Hausenblas, "A Comparison of Federation over SPARQL Endpoints Frameworks," in *Knowledge Engineering and the Semantic Web*, ser. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2013, vol. 394, pp. 132–146. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41360-5_11

[35] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, and A.-C. N. Ngomo, "A Fine-Grained Evaluation of SPARQL Endpoint Federation Systems," *Semantic Web Interoperability, Usability, Applicability*, 2015, under review. [Online]. Available: http://www.semantic-web-journal.net/system/files/swj954.pdf

[36] M. Schmachtenberg, C. Bizer, and H. Paulheim, "Adoption of the Linked Data Best Practices in Different Topical Domains," in *The Semantic Web ISWC 2014*, 2014, pp. 245–260.

[37] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt, "FedX: A Federation Layer for Distributed Query Processing on Linked Open Data," in *The Semanic Web: Research and Applications*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, vol. 6644, pp. 481–486. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21064-8_39

[38] W3C OWL Working Group, *OWL 2 Web Ontology Language: Document Overview (Second Edition)*. W3C Recommendation, 11 December 2012, available at http://www.w3.org/TR/owl2-overview/.

[39] X. Wang, T. Tiropanis, and H. C. Davis, "LHD: Optimising Linked Data Query Processing Using Parallelisation," in *Proceedings of the WWW2013 Workshop on Linked Data on the Web, Rio de Janeiro, Brazil, 14 May, 2013*, 2013. [Online]. Available: http://ceur-ws.org/Vol-996/papers/ldow2013-paper-06.pdf

[40] X. Wang, T. Tiropanis, and H. Davis, "Evaluating Graph Traversal Algorithms for Distributed SPARQL Query Optimization," in *The Semantic Web*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7185, pp. 210–225. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-29923-0_14

[41] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds, "Efficient RDF storage and retrieval in Jena2," in *Proc. First International Workshop on Semantic Web and Databases*, 2003. [Online]. Available: http://www.cs.uic.edu/~ifc/SWDB/papers/Wilkinson_etal.pdf

[42] World Wide Web Consortium, "SPARQL 1.1 Query Language: Translation to the SPARQL Algebra," http://www.w3.org/TR/sparql11-query/#sparqlQuery, 2013.

[43] World Wide Web Consortium, "World Wide Web Consortium (W3C)," http://www.w3.org, 2015.

[44] World Wide Web Consortium (W3C), *RDF/XML Syntax Specification (Revised)*. W3C Recommendation, 2004, available at http://www.w3.org/2004/REC-rdf-syntax-grammar-20040210/.

[45] World Wide Web Consortium (W3C), *SPARQL Query Language for RDF*. W3C Recommendation, 2008, available at http://www.w3.org/TR/rdf-sparql-query/.

[46] World Wide Web Consortium (W3C), *SPARQL 1.1 Protocol*. W3C Recommendation, 2013, available at http://www.w3.org/TR/sparql11-protocol/.

[47] World Wide Web Consortium (W3C), *SPARQL 1.1 Query Results CSV and TSV Formats*. W3C Recommendation, 2013, available at http://www.w3.org/TR/sparql11-results-csv-tsv/.

[48] World Wide Web Consortium (W3C), *SPARQL 1.1 Query Results JSON Format*. W3C Recommendation, 2013, available at http://www.w3.org/TR/sparql11-results-json/.

[49] World Wide Web Consortium (W3C), *SPARQL Query Results XML Format (Second Edition)*. W3C Recommendation, 2013, available at http://www.w3.org/TR/rdf-sparql-XMLres/.

[50] World Wide Web Consortium (W3C), "Semantic Web," 2015. [Online]. Available: http://www.w3.org/standards/semanticweb/

[51] J. Zemánek, S. Schenk, and V. Svátek, "Optimizing SPARQL Queries over Disparate RDF Data Sources through Distributed Semi-Joins," in *Proceedings of the Poster and Demonstration Session at the 7th International Semantic Web Conference (ISWC2008), Karlsruhe, Germany, October 28, 2008*, 2008. [Online]. Available: http://ceur-ws.org/Vol-401/iswc2008pd_submission_69.pdf

AUTHOR BIOGRAPHIES

Sven Groppe earned his diploma degree in Informatik (Computer Science) in 2002 and his Doctor degree in 2005 from the University of Paderborn. He earned his habilitation degree in 2011 from the University of Lübeck. He worked in the European projects B2B-ECOM, MEMPHIS, ASG and TripCom. He was a member of the DAWG W3C Working Group, which developed SPARQL. He was the project leader of the DFG project LUPOSDATE, an open-source Semantic Web database, and one of the project leaders of two research projects, which research on FPGA acceleration of relational and Semantic Web databases. His research interests include databases, Semantic Web, query and rule processing and optimization, Cloud Computing, peer-to-peer (P2P) networks, Internet of Things, data visualization and visual query languages.

Dennis Heinrich received his M.Sc. in Computer Science in 2013 from the University of Lübeck, Germany. At the moment he is employed as a research assistant at the Institute of Information Systems at the University of Lübeck. His research interests include FPGAs and corresponding hardware acceleration possibilities for Semantic Web databases.

Stefan Werner received his Diploma in Computer Science (comparable to Master of Computer Science) in March 2011 at the University of Lübeck, Germany. Now he is a research assistant/PhD student at the Institute of Information Systems at the University of Lübeck. His research focuses on multi-query optimization and the integration of a hardware accelerator for relational databases by using run-time reconfigurable FPGAs.