C Basca

Abraham Bernstein

Querying a messy Web of Data with Avalanche

TECHNICAL REPORT – No. IFI-2013.03

2013

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

C Basca, Abraham Bernstein
Querying a messy Web of Data with Avalanche
Technical Report No. IFI-2013.03
Dynamic and Distributed Information Systems
Department of Informatics (IFI)
University of Zurich
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland
http://www.ifi.uzh.ch/ddis/

University of Zurich^UZH

Cosmin Basca
Abraham Bernstein

Querying a Messy Web of Data with Avalanche

TECHNICAL REPORT – No. IFI-2013.03

June 2013

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

Querying a Messy Web of Data with AVALANCHE

Cosmin Basca[a,1,*], Abraham Bernstein[a]

[a]*Dynamic and Distributed Information Systems, University of Zurich, Switzerland*

Abstract

The Web thrived on messiness or to use positive attributes: diversity, flexibility, and openness. By limiting any convention to the communications protocol (HTTP) and the structure of data formatting (HTML) it enabled usages that were beyond the imagination of its inventors. With the advent of the Semantic Web, a Web of Data is emerging interlinking ever more machine readable data fragments represented as RDF documents or queryable semantic endpoints. Recent efforts have enabled applications to query the entire Semantic Web for up-to-date results. Such approaches are either based on a centralized store, centralized indexing of semantically annotated meta-data, or link traversal and URI dereferencing as often used in the case of Linked Open Data. These approaches violate the openness principle by making additional assumptions about the structure and/or location of data on the Web and are likely to limit the diversity of resulting usages. As a consequence, the guiding question of this paper is: *How can we support querying the messy web of data whilst adhering to a minimal, least-constraining set of principles that mimic the ones of the original web and will—hopefully—support the same type of creative flurry?*

In this article we propose a technique called AVALANCHE, designed to allow a data surfer to query the Semantic Web transparently without making any prior assumptions about the data distribution, schema-alignment, pertinent statistics, data evolution, and accessibility of servers. Specifically, AVALANCHE can perform up-to-date queries over SPARQL endpoints. Given a query it first gets on-line statistical information about potential data sources and their data distribution. Then, it plans and executes the query in a concurrent and distributed manner trying to quickly provide first answers.

The main contribution of this paper is the presentation of this open and distributed SPARQL querying approach. We empirically evaluate AVALANCHE using the realistic FedBench data-set over 26 servers, as well as investigate its behavior for varying degrees of instance-level distribution "messiness" using the LUBM synthetic data-set spread over 100 servers. Results show that AVALANCHE is robust and stable in spite of varying network latency finding first results for 80% of the queries in under 1 second. It also exhibits stability for some classes of queries when instance-level distribution messiness increases. We also illustrate, how AVALANCHE addresses the other sources of messiness (pertinent data statistics, data evolution and data presence) by design and show its robustness by removing endpoints during query execution. Finally, we point out the challenges that still exist, discussing potential solutions.

Keywords: federated SPARQL, RDF distribution messines, query planing, adaptive querying, changing network conditions

1. Introduction

With the introduction of the World Wide Web the way we share knowledge and conduct day to day activities has changed fundamentally. Indeed, the massive growth of the web was partially based on the fact that the only limitations it set on participation where: the communications protocol (HTTP) and the structure of data formatting (HTML), which included the notion of linking documents. The rest was consciously left unspecified resulting in a plethora of usages and allowing for highly scalable implementations, since it avoids any central control. Indeed, this messy plethora of usage resulted in an almost genetic search for "good" web usages giving rise to the web as we know it today. Or to put in in Sir Tim Berners-Lee words: "... if we end up building all the things I can imagine we'll have failed".[2]

With the advent of the Semantic Web, a Web of Data is emerging interlinking ever more machine readable data fragments represented as RDF documents or queryable semantic endpoints. It is in this ecosystem that unexplored avenues for application development are emerging. While some application designs include a Semantic Web data crawler, others rely on services that facilitate access to the Web of Data either through the SPARQL protocol or various APIs like the ones exposed by *Sindice* or *Swoogle*. As the mass of data continues to grow—Linked Open Data [5] accounts for 27 billion triples as of January 2011[3]—the scalability factor combined with the Web's uncontrollable nature and its heterogeneity will give raise to a new set of challenges. A question marginally addressed today is how to support the same messiness in querying the Web of Data that gave rise to the virtually endless possibilities of using the traditional Web. In other words: *How can we support querying the messy web of data whilst adhering to a minimal, least-constraining set of principles that mimic the ones of the original web and will—hopefully—support the same type of creative flurry?*.

Translating the guiding principles of the Web to the Web of Data proposes that we should use a single communications protocol (i.e. HTTP with encoded SPARQL queries) and use a common data representation format (some encoding of RDF), which allows embedding links. In addition, it implicitly proposes that:

(a) we cannot assume any (or control the) distribution of data to servers,
(b) there is no guarantee of a working network,
(c) there is no centralized resource discovery system (even though crawled indices akin to Google in the traditional web may be provided),
(d) the size of RDF data no longer allows us to consider single-machine systems feasible,
(e) data will change without any prior announcement,
(f) there is absolutely no guarantee of RDF-resources adhering to any kind of predefined schema, be correct, or refer/link to other existing data items—in other words: the Web of Data will be a mess and "this is a feature not a bug."

As an example consider the life sciences domain: here information about drugs, chemical compounds, proteins and other related aspects is published continuously. Some research institutions expose part or all of their data freely as RDF dumps relying on others

*Corresponding author. Tel: +41 44 635 4318
Email addresses: basca@ifi.uzh.ch (Cosmin Basca), bernstein@ifi.uzh.ch (Abraham Bernstein)
[1]Partial support provided by Swiss National Science Foundation award number 200021-118000
[2]http://vimeo.com/11529540, *Web 3.0* by Kate Ray
[3]http://www4.wiwiss.fu-berlin.de/lodcloud/state/#domains

to index it as in the cases of the CheBi[4] and KEGG[5] datasets, while others host their own endpoints like in the case of the Uniprot dataset.[6] Hence, anybody querying the data will have no control over its distribution,[7] no guarantees about the availability and network connectivity of the information sources,[8] data content changes continuously due to scientific breakthroughs/discoveries, and a plethora of schema's are used.[9] Often-times problem domains and researchers' questions span across several datasets or disciplines that may or may not overlap. Even in the light of this messiness, the data about drugs, chemical compounds, proteins, and their interrelations is queried constantly resulting in is a strong need to provide integrated and up-to-date information.

Several approaches that tackle the problem of querying the entire Web of Data have emerged lately and most adhere to the explicit principles. They do, however, not address the implicit principles. One solution, *uberblic.org*[10], provides a centralized queryable endpoint for the Semantic Web that caches all data. This approach allows searching for and joining potentially distributed data sources. It does, however, incur the significant problem of ensuring an up-to-date cache and might face crucial scalability hurdles in the future, as the Semantic Web continues to grow. Additionally, it violates a number of the implicit principles locking-in data. Furthermore, as Van Alstyne *et al.* [41] argue, incentive misalignments would lead to data quality problems and, hence, inefficiencies when considering the Web of Data as "one big database."

Other approaches base themselves on the guiding principles of Linked Open Data publishing and traverse the LOD cloud in search of the answer. Obviously, such a method produces up-to-date results and can detect data locations only from the URIs of bound entities in the query. Relying on URI structure, however, may cause significant scalability issues when retrieving distributed data sets, since (i) the servers dereferenced in the URI may become overloaded and (ii) it limits the possibilities of rearranging (or moving) the data around by binding the id (*i.e.*, URI) to its storage location. Just consider for example the *slashdot effect*[11] on the traditional web. Finally, traditional database federation techniques have been applied to query the Web of Data. One of the main drawbacks with traditional federated approaches stemming from their ex-ante (i.e., before the query execution) reliance on *fine-grained* statistical and schema information meant to enable the mediator to build efficient query execution plans. Whilst these approaches do not assume central control over data they do assume ex-ante knowledge about it facing robustness hurdles against network failure and changes in the underlying schema and statistics (invalidating implicit principles b and f).

In this paper, we propose Avalanche , a novel approach for querying the messy Web of Data which: (1) *makes no assumptions about data distribution, schema, availability,*

[4]http://www.ebi.ac.uk/chebi/downloadsForward.do

[5]http://bioit.fleming.gr/mrb/Controller?workflow=ViewModel&eid=242

[6]http://beta.sparql.uniprot.org/

[7]Different copyright and intellectual property policies may prevent access to downloading part or the entire dataset but permit access to it on a per-query basis with potential restrictions like time and/or quota limits.

[8]Some institutions move repositories or change access policies, resulting in server unavailability.

[9]Some sub-disciplines may favor dissimilar but overlapping attributes describing their results, have differing habits about using same-named attributes, and use a diversity of taxonomies with varying semantics.

[10]http://platform.uberblic.org/

[11]http://en.wikipedia.org/wiki/Slashdot_effect

or partitioning and is skew resistant for some classes of queries, (2) provides *up-to-date results* from distributed indexed endpoints, (3) is *adaptive* during execution adjusting dynamically to external network changes, (4) *does not require detailed fine-grained ex-ante statistics* with the query engine, and (5) is *flexible* as it makes limited assumptions about the structure of participating triple stores. It does, however, assume that the query will be distributed over triple-stores and not "mere" web-pages publishing RDF.[12] The system, as presented in the following sections, is based on a first prototype described in [3] and brings a number of new extensions and improvements to our previous model.

Consequently, Avalanche proposes a novel technique for executing queries over Web of Data SPARQL endpoints. The traditional *optimize then execute* paradigm—highly problematic in the Web of Data context in its original conceptualization—is extended into an exhaustive, concurrent, and dynamically-adaptive meta-optimization process where fine-grained statistics are requested in a first phase of the query execution. Hence, the main contributions of our approach are:

- an on-demand transparent querying approach over the Web of Data, without fine-grained prior knowledge about its distribution
- a novel, inter-plan adaptive SPARQL execution paradigm
- a novel planning strategy and cost model for dealing with the involvement of large numbers of endpoints
- a formal description of our approach with possible optimizations for each step
- a reference implementation of the Avalanche system

Hence, Avalanche supports messiness at various levels: data-distribution, schema-alignment, prior registration with respect to statistics, constantly evolving data, and unreliable accessibility of servers (either through network or host failure, HTTP 404's, or changes in policy of the publishers).

In the remainder we first review the relevant related work of the current state-of-the-art. Section 3 provides a detailed description of Avalanche. In Section 4 we evaluate several planning strategies and estimate the performance of our system. In Section 5 we present several future directions and optimizations, and conclude in Section 6.

2. Related work

Several solutions for querying the Web of Data over distributed SPARQL endpoints have been proposed before. They can be grouped into two streams: **I.** distributed query processing, **II.** RDF indexing, and **III.** statistical information gathering over RDF sources.

[12]An assumption that is not overly limiting, since one could imagine service providers providing SPARQL endpoints that dynamically load RDF-documents at query time. Since those on-the-fly endpoints only cover one RDF-document at a time they are not hampered by the same scalability issues as a centralized SPARQL endpoints would be.

Distributed query processing: A broad range of RDF storage and retrieval solutions exist. They can be grouped along the dimensions of *partition restrictiveness* (i.e., the degree to which the system controls the data distribution) and the intended *source addressing space* (i.e., the design goal in terms of physical distribution of hosts from single machine through clusters and the cloud to a global uncontrolled network of servers) as shown in Figure 1. Although not intended as a measure of scalability and performance the Figure positions the various approaches relative to the desired goal – a globally addressable and highly flexible system: both paramount features when handling messy semi-structured data at large-scale.

Research on distributed query processing has a long history in the database field [37, 22]. Its traditional concepts are adapted in current approaches to provide integrated access to RDF sources distributed on the Web of Data. For instance, *Yars2* [17] is an end-to-end semantic search engine that uses a graph model to interactively answer queries over semi-structured interlinked data, collected from disparate Web sources. Another example is the *DARQ* engine [32], which divides a SPARQL query into several subqueries, forwards them to multiple, distributed query services, finally, integrating the results of the subqueries. Inspired by peer-to-peer systems, *Rdfpeers* [8] is a distributed RDF repository that stores three copies of each triple in a peer-to-peer network, by applying global hash functions to its subject, predicate and object. Stuckenschmidt et. al [38] consider a scenario in which multiple distributed sources contain data in the form of publications. They describe how the *Sesame* RDF repository [7] needs to be extended, by using a special index structure that determines which are the relevant sources to be considered for a query. *Virtuoso* [9]—a data integration software developed by OpenLink Software—is also focused on distributed query processing. The drawback of these solutions is, however, that they assume total control over the data distributions – an unrealistic assumption in the open Web.

Similarly, *SemWIQ* [24] uses a mediator distributing the execution of SPARQL queries transparently. Its main focus is to provide an integration and sharing system for scientific data. Whilst it does not assume fine-grained control over the instance distribution they assume perfect knowledge about their `rdf:type` distribution. Addressing this drawback some [44, 35] propose to extend SPARQL with explicit instructions controlling where to execute certain sub-queries. Unfortunately, this assumes an ex-ante knowledge of the data distribution on part of the query writer. Finally, Hartig et al. [18] describe an approach for executing SPARQL queries over Linked Open Data [5] based on graph search. Whilst they make no assumptions about the openness of the data space, the Linked Open Data rules requires them to place the data on the URI-referenced servers – a limiting assumption for example when caching/copying data. A notable approach to browse the WoD and run structured queries on it is depicted by Sig.ma [39], a system designed to automatically integrate heterogenous web data sources. Suited to handle schema messiness Sig.ma differs from Avalanche mainly in its scope, which is that of aggregating various data sources in the attempt to offer a solution, while Avalanche (tackling data distribution messiness) does not integrate RDF indexes, but "guides" the query execution process to find exact matches.

Other flexible techniques have been proposed, such as the evolutionary query answering system *eRDF* by Guéret et. al [12, 30, 13], where genetic algorithms are used to "learn" how to best execute the SPARQL query. The system learns each time a triple pattern gets executed. As the authors demonstrate, *eRDF* behaves better the

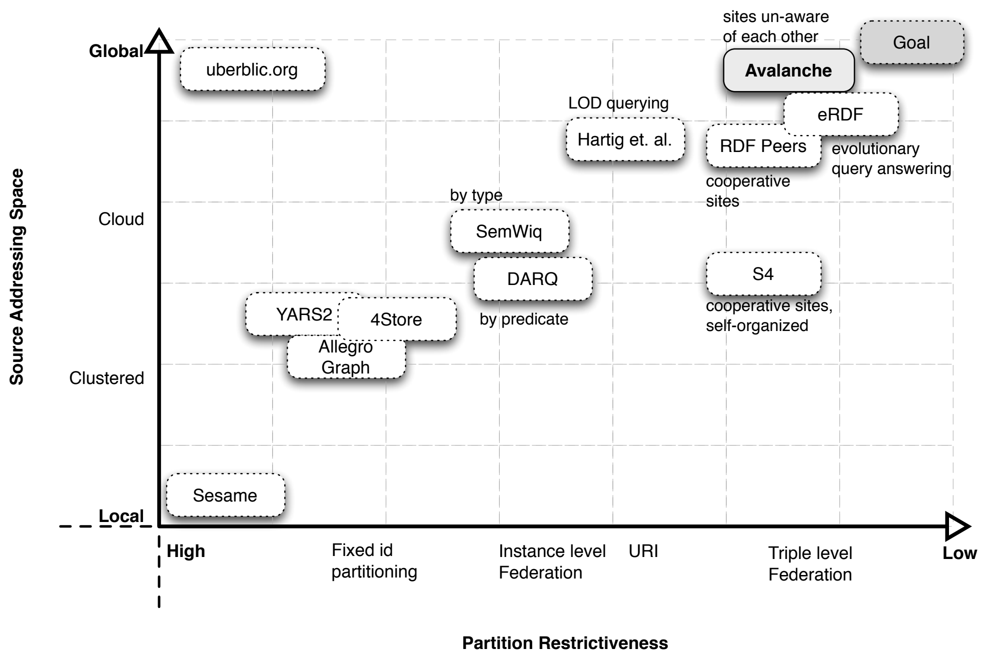


Figure 1: Distributed SPARQL processing systems and algorithms

more complex the query, while simple queries (one or two triple pattern queries) render low performance. Finally Muehleisen et. al [28] advance the idea of a self organized RDF storage and processing system called *S4*. The approach relies on the principles of swarm-logic and exposes certain similarities with peer-to-peer systems.

RDF indexing: A number of methods and techniques to store and index RDF have been proposed to date, some like Hexastore [42] and RDF3X [29] construct on-disk indexes based on B+Trees while exploiting all possible permutations of *Subjects*, *Predicates* and *Objects* in an RDF triple. Other notable approaches include [2], where RDF is index using a matrix for each triple term pair – an approach suitable for low selectivity queries, suffering in performance however when highly selective queries are asked. Furthermore GRIN [40] proposes a special graph index which stores "center" vertexes and their neighborhoods leading to lower memory consumptions and faster times to answer graph based queries than traditional approaches such as Jena[13] and Sesame[14].

Query optimization: Research on query optimization for SPARQL includes query rewriting [19], join re-ordering based on selectivity estimations [26, 4, 29], and other statistical information gathering over RDF sources [23, 16]. *RDFStats* [23] is an extensible RDF statistics generator that records how often RDF properties are used and feeds

[13]http://jena.sourceforge.net/
[14]http://www.openrdf.org/

automatically generated histograms to *SemWIQ*. Histograms on the combined values of SPO (*S*ubject *P*redicate *O*bject) triples have proved to be especially useful to provide selectivity estimations for filters [4]. For joins, however, histograms can grow very large and are rarely used in practice. Another approach is to precompute frequent paths (i.e., frequently occurring sequences of S, P or O) in the RDF data graph and keep statistics about the most beneficial ones [26]. It is unclear how this would work in a highly distributed scenario. Finally, Neumann et. al [29] note that for very large datasets (towards billions of triples) as even simple index scans become too expensive, single triple pattern selectivity is not enough to ensure accurate join selectivity estimation. As pattern combinations are more selective, they successfully integrate *holistic sideways information passing* with the recording of detailed join cardinalities of constants joined with the entire graph as means of improving join selectivity. An alternative approach is represented by summarizing indexes as described by Harth et. al. [16] in *data summaries*.

3. Avalanche – The Design and Implementation of an indexed Web of Data Query Processing System

Avalanche is part of the larger family of Federated Database Management Systems or FDBMS's [20]. Focusing primarily on answering SPARQL queries over WoD endpoints, Avalanche relies on a commonly used data representation format: RDF and SPARQL as the main access operation. In contrast to relational FDBMS, where the schema changes are costly and, therefore, happen seldom, the WoD is subjected to constant change both schema and content-wise. Hence, the major design difference between Avalanche and previous systems is that *it assumes that the distribution of triples to machines participating in the query evaluation is unknown prior to query execution.* To achieve its *loose coupling* Avalanche adheres to strict principles of transparency as well as heterogeneity, extensibility and openness.

Transparency. A critical aspect in federated query processing is exhibited by transparency. Avalanche addresses transparency at four levels:
- *location transparency*: when submitting a query to Avalanche the user is not aware nor does it need to concern itself with where data is actually located,
- *physical data independence transparency*: similarly, users are not aware of how data is physically stored,
- *replication and fragmentation transparency*: Avalanche is data-distribution agnostic, incidentally also one of its main contributions and,
- *network transparency*: built on top of HTTP, Avalanche inherently derives the protocol's flexibility and transparency.

Heterogeneity, extensibility, and openness. Participating endpoints are not constrained in any way with regard to the schemas, vocabularies, or ontologies used. Furthermore, over time the federation can evolve unrestrained as new data sources can be added without impacting existing ones. Opaque to physical data representation or structure, Avalanche can virtually accommodate any kind of data sources that can be exported as RDF or as SPARQL endpoints.[15]

[15]A well-known technology in this regard is the D2R server and its D2RQ declarative language [6], intended to facilitate the mapping of relational data to RDF.

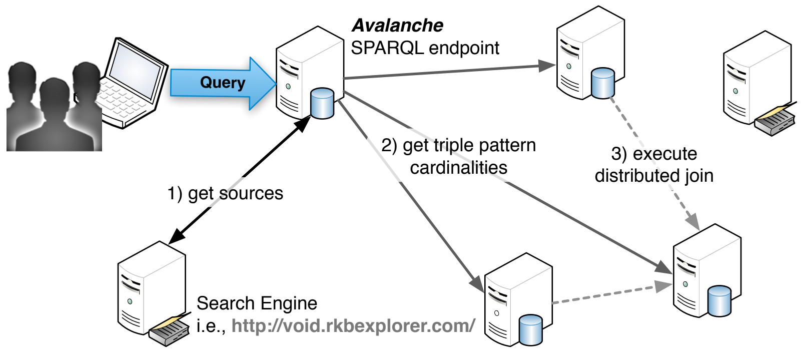

Query
Avalanche
SPARQL endpoint
1) get sources
2) get triple pattern cardinalities
3) execute distributed join
Search Engine
i.e., http://void.rkbexplorer.com/

Figure 2: An idealized view of the Avalanche execution model illustrating the three major phases: source discovery, statistics gathering, and query planing/distributed execution

In addition, akin to peer to peer systems (p2p), Avalanche does not assume any centralized control. Any computer on the internet can assume the role of a Avalanche-broker. However, Avalanche is not a p2p system, since participating sites do not make a portion of their resources—*CPU, RAM*, or *disk*—directly available to other members, nor are they bookkeeping information concerning neighboring hosts.
Another important distinction to prior art, lies within the early stages of the query execution. Traditionally, statistical information is indexed *ex-ante*, i.e., ahead of query time in the federation's meta-database from where it is later retrieved to aid the source selection and query optimization processes. Avalanche relies on each participating site to manage their respective statistics individually – a trait shared to a varying degree by virtually any optimized RDF-store. Consequently, *query-relevant statistical information is retrieved at the beginning of each query execution phase* as illustrated in Figure 2.

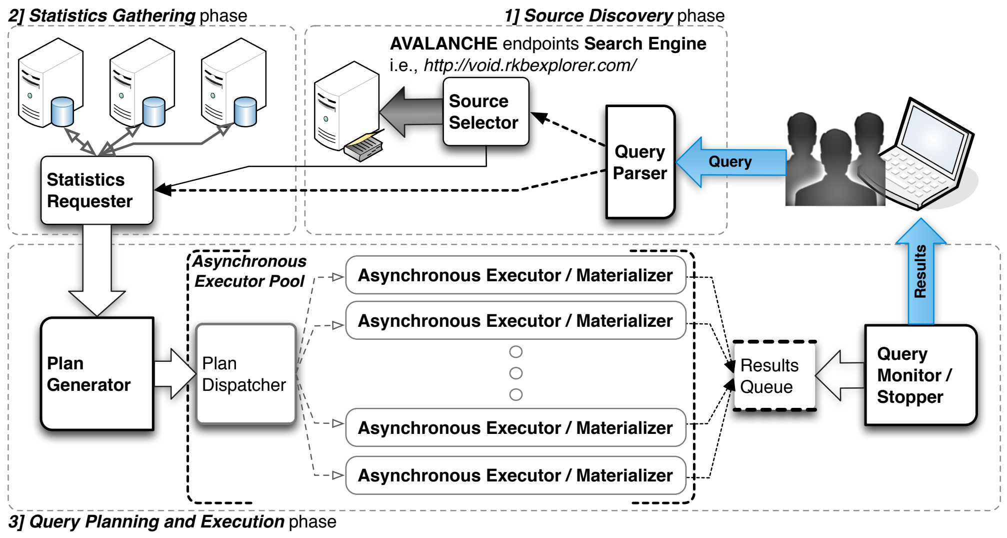
In the following, we will first outline our approach detailing its basic operators and the actual system using a motivating example. This will lead the way towards thoroughly describing the Avalanche components and its novelty.

3.1. Avalanche *Overview*

The Avalanche system consists of the following major components working together in a concurrent asynchronous pipeline: *i*) the Avalanche *Source Selector* relying on the *endpoints Web Directory* or *Search Engine*, *ii*) the *Statistics Requester*, *iii*) the *Plan Generator*, *iv*) the *Plan Executor Pool*, *v*) the *Results Queue* and *vi*) the *Query Execution Monitor/Stopper* as illustrated in Figure 3.

These components are coordinated into three query execution phases: *1*) the *Source Discovery* phase, *2*) the *Statistics gathering* phase, and *3*) the concurrent *Query Planing and Execution* phase. We will now discuss how all the components are coordinated into these three execution phases. The detailed technical description of the elements will be covered in the following subsections.

During *Source Discovery*, participating hosts are identified by the *Source Selector*,



2] Statistics Gathering phase
1] Source Discovery phase
AVALANCHE endpoints Search Engine
i.e., http://void.rkbexplorer.com/
Source Selector
Query Parser
Query
Statistics Requester
Results
Asynchronous Executor Pool
Asynchronous Executor / Materializer
Asynchronous Executor / Materializer
Plan Generator
Plan Dispatcher
Results Queue
Query Monitor / Stopper
Asynchronous Executor / Materializer
Asynchronous Executor / Materializer
3] Query Planning and Execution phase

Figure 3: The AVALANCHE execution pipeline

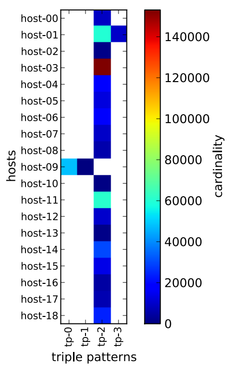
which interfaces with a *Search Engine* such as *voID store*,[16] *Sindice's*[17] SPARQL endpoint, or a *Web Directory*. A lightweight endpoint-schema inverted index can also be used. Ontological prefix (the shorthand notation of the schema, i.e. foaf) and schema invariants (i.e. predicates, concepts, labels, etc) are appropriate candidate entries to index. More complex source selection algorithms and indexes have been proposed [25] that could successfully be used by AVALANCHE given adequate protocol adaptations.

The next step—*Statistics gathering*—queries all selected AVALANCHE endpoints (from the set of known hosts H) for the individual cardinalities $card_{i,j}$ (number of instances) for each triple pattern tp_i from the set of all triple patterns in the query T_Q as detailed in Definition 3.1. The voID[18] vocabulary can be used to describe triple pattern cardinalities when predicates are bound or when schema concepts are used, along with more general purpose dataset statistical information making use of terms like: void:triples, void:properties, void:Linkset, etc. Additionally, the same can be accomplished by using aggregating SPARQL COUNT-queries for each triple pattern or by simple specialized index lookups in some triple-optimized index structures [42].

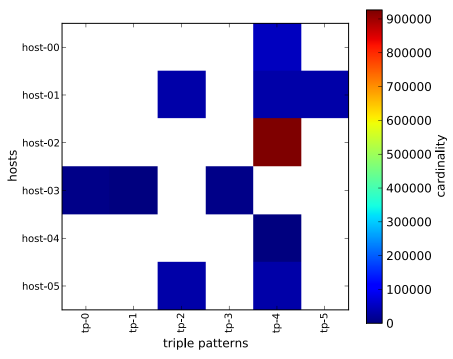
Definition 3.1 *Given a query Q, T_Q is the set of all triple patterns $\in Q$ and H the set of all reachable hosts. $\forall tp_i \in T_Q$ and $\forall h_j \in H$, we define $\boldsymbol{card_{i,j} = card(tp_i, h_j)}$ as the* ***triple pattern cardinality*** *of triple pattern tp_i on host h_j.*

During the *Query Planing and Execution* phase, the *Plan Generator* proceeds with constructing the plan matrix (see Definition 3.2): a two dimensional matrix listing the

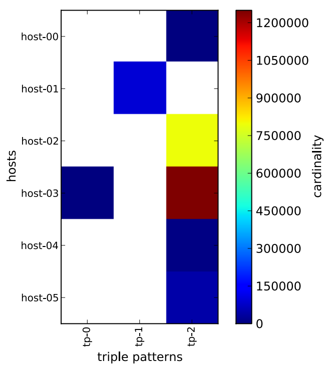
[16]http://void.rkbexplorer.com/
[17]http://sindice.com/
[18]http://www.w3.org/2001/sw/interest/void/



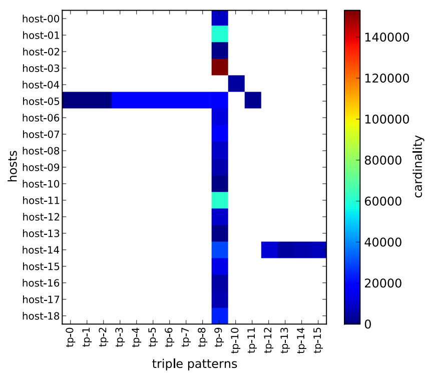
(a) $FQ5$



(b) $FQ14$



(c) $FQ20$



(d) $FQ33$

Figure 4: Plan matrixes represented as heat-maps for selected Fedbench benchmark queries – for further details about the specific queries and benchmark please refer to Section 4.

cardinalities of all triple patterns of a query by possible hosts. Consider, for example, the the plan matrixes for a selection of FedBench queries visualized in Figure 4 as a heat map, where white indicates the absence of triples matching a triple pattern tp_i on some host h_j (i.e., $card_{i,j} = 0$). Focusing on Figure 4 a) we, for example, see that only host-09 has triples matching tp_1.

Definition 3.2 *The two dimensional set $PM_Q = \{card_{i,j} \mid \forall tp_i \in T_Q$ and $\forall h_j \in H\}$ is called the **plan matrix**, where $card_{i,j}$ are triple pattern cardinalities as ascertained in Definition 3.1.*

To ensure the numerical stability of the following computations the cardinalities are stored as logarithmic probabilities by normalizing the cardinalities with the total number of triples $T_{MAX} = \sum_{i=0}^{|H|} triples_{h_i}$, where $triples_{h_i}$ represents the total number of triples on host h_i.

The plan matrix is instrumental for the generation of query plans. Every query plan p contains one triple-pattern/host pair (tp_i, h_j) for each triple pattern tp_i in the query T_Q, where all tp_i contain at least one triple (i.e., $card(tp_i, h_j) \neq 0$; see Definition 3.3). Thus, planing is equal to exploring the set of possible triple-pattern/host pairs resulting in valid plans. Visually, this equals to finding sets of non-zero cardinality squares, where each column is represented exactly once.

Definition 3.3 *A **query plan** is the set $p = \bigcup(tp_i, h_j)$ that contains exactly one triple-pattern/host-pair (tp_i, h_j) per $tp_i \in T_Q$, where $card(tp_i, h_j) \neq 0$ and $h_j \in H$.*

While some queries can produce no plans, the universe of all plans (see Definition 3.4) has a theoretical upper-bound equal to $|H|^{|T_Q|}$ (see Definition 3.4). Albeit an exponential number of possible plans can theoretically exist, our empirical evaluation suggests that real-world datasets often produce sparse plan matrixes—possibly a consequence of the LoD's heterogeneity—resulting in a significantly lower number of valid plans (i.e., akin to the plan matrixes in Figure 4). Hence, the task of the *Plan Generator* is to explore the space of all possible valid SPARQL 1.1 rewritings of the original query Q by pairing triple patterns from T_Q with available endpoints from H.

Definition 3.4 *The set of all plans for query Q, $P_Q = \{p_i \mid p_i$ is a query plan as in Definition 3.3 $\}$ is called the **query plan space** or **universe of all plans**, with $0 \leq |P_Q| \leq |H|^{|T_Q|}$.*

It is important to note that factors such as the sheer size of the Web of Data, its unknown distribution, and multi-tenancy aspect may prevent Avalanche from guaranteeing result completeness. Whilst the proposed planning system and algorithm are complete the execution of all plans to ensure completeness could be prohibitively expensive. Hence, Avalanche will normally not be allowed to exhaust the entire search space—unless the query is simple or the search space is narrow enough. Therefore, Avalanche will try to optimize the query execution to quickly find the **first K** results by picking plans first that are more "promising" in terms of getting results quickly. In addition, the *Query Execution Monitor* will monitor for termination conditions to finish query execution when further execution seems less promising. To further reduce the size of the search space, a windowed version of the search algorithm can be employed. Here only the first PS partial plans are considered with each exploratory step, thus sacrificing completeness.

As depicted in Figure 3 the *Plan Generator* relies on statistics about data contained on the different hosts; statistics provided by the *Statistics Requester.* Any generated plan gets assigned to a socket-asynchronous *Plan Executor and Materializer.* All executors are executed concurrently and managed by the asynchronous *Executor Pool.* While the asynchronous pool can accommodate orders of magnitude more individual workers than traditional parallel thread or process-based pools a parameter is used to limit the concurrency in order to avoid flooding the system (remote database operations still remain expensive). Finally, once a plan is finished the executor will signal its completion by placing the results in the *Results Queue.*

3.2. Contextualizing Example: Finding Drug Information from Various LoD Sources

To contextualize Avalanche further, consider the example query $Q_{ex.}$ in Listing 1, executing over the Fedbench[19] benchmark datasets. Specifically the query touches data that are distributed across three life-sciences domain datasets: DrugBank,[20] KEGG,[21] and ChEBI[22]. It is Avalanche's goal to find all drugs from DrugBank, together with their URL from KEGG and links to their repective graphical depiction from ChEBI.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX chebi: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?drug ?keggUrl ?chebiImage
WHERE
{
    ?drug        rdf:type                    drugbank:drugs .
    ?drug        drugbank:keggCompoundId     ?keggDrug .
    ?drug        drugbank:genericName        ?drugBankName .
    ?keggDrug    chebi:url                   ?keggUrl .
    ?chebiDrug   dc:title                    ?drugBankName .
    ?chebiDrug   chebi:image                 ?chebiImage .
}
```

Listing 1: Motivating example - Life Sciences query from the Fedbench benchmark.

During query execution Avalanche performs a search for all possible rewritings of the original query (Listing 1) as SPARQL 1.1. One such decomposition is exemplified in Listing 2, where the SERVICE clause is used to bind subqueries to the specified endpoints.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX chebi: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?drug ?keggUrl ?chebiImage WHERE {
    SERVICE <http://drugbank-endpoint/sparql> {
        ?drug        drugbank:genericName        ?drugBankName .
        ?drug        drugbank:keggCompoundId     ?keggDrug .
        ?drug        rdf:type                    drugbank:drugs
    } .
    SERVICE <http://chebi-endpoint/sparql> {
        ?chebiDrug   chebi:image                 ?chebiImage .
        ?chebiDrug   dc:title                    ?drugBankName
    } .
    SERVICE <http://kegg-endpoint/sparql> {
        ?keggDrug    chebi:url                   ?keggUrl
    }
}
```

Listing 2: Motivating example query rewritten as a SPARQL 1.1 federated query.

[19]https://code.google.com/p/fbench/
[20]http://www.drugbank.ca/
[21]http://www.genome.jp/kegg/
[22]http://www.ebi.ac.uk/chebi/

3.3. Optimizing requirements for Avalanche *endpoints*

From the overview and the example we can infer that Avalanche employs both a common ID space and a set of capabilities for Avalanche endpoints to optimize execution. We succinctly discuss these in turn.

Common IDs. A requirement for executing joins between any two hosts is that they share a common *id* space. The natural identity on the web is given by the URI itself. However some statistical analyses of URIs on the web[23] show that the average length of a URI is 76 characters, while analyses of the Billion Triple Challenge 2010 dataset[24] demonstrate that the maximum length of RDF literals is 65244 unicode characters long with most of the string literals being 10 characters in length. Therefore, using the actual RDF literal constants (URIs or literals) can lead to a high cost when performing distributed joins.

To reduce the overhead of using long strings we used a number encoding of the URIs. To avoid central points of failure based on dictionary encoding or similar techniques, we propose the use of a hash function responsible for mapping any RDF string to a common number-based id format. For our experiments, we applied the widely used SHA family of hash functions on the indexed URIs and literals. An added benefit of a common hash function is that the hosts involved in answering a query, can agree on a common mapping function prior to executing the query. Note that this proposition is not a necessary condition for the functioning of Avalanche but represents an optimization that will lead to performance improvements.

Endpoint operations. To optimize SPARQL execution performance Avalanche takes advantage of a number of operations that extend the traditional SPARQL endpoint functionality, which we discuss in the following.

Whilst we acknowledge that the implementation of these procedures puts a burden on these endpoints their implementation should be trivial for most triple-stores. Some of the operations are either SPARQL 1.1 compliant or can be expressed as plain SPARQL queries as fully detailed in Appendix A, while others will be internally available in any indexed triple store and "only" need to be exposed (i.e. *set filtering* or *set merge*). From a functional point of view the procedures are classified into: ***i)*** *execution operators* and ***ii)*** *state management operators.*

The next subsections will describe the basic Avalanche operators and the functionality of its most important elements: the *Plan Generator* and *Plan Executor / Materializer* as well as explain how the overall execution pipeline stops.

3.4. Generating Query Plans

Enumerating the space of possible query execution plans lays at the core of query processing and planning. Traditionally, optimal plan space traversal was accomplished by using techniques such as *dynamic programming.* A popular heuristic when doing so is to discard all plans with the exception of left-deep ones. Such exhaustive strategies of traversing the entire plan universe in order to find the best (or lowest cost) plan can

[23] http://www.supermind.org/blog/740/average-length-of-a-url-part-2
[24] http://gromgull.net/blog/category/semantic-web/billion-triple-challenge/

become prohibitively expensive for queries where the number of joins is high[25] – even when only left-deep plans are considered and the execution is centralized [33]. Moreover, when dealing with *uncertain* constraints (i.e. *fast first* results) RDBMS's like Oracle RDB [1] heuristically execute several plans competitively in parallel for a short interval of time to increase the likelihood of hitting the most relevant cases under the assumption of a ZIPF distribution.

The distributed context within which AVALANCHE is designed to operate in conjunction with the flexibility of the RDF model which can lead to arbitrarily complex queries, renders the planning space exponential in the worst of cases: $|H|^{|T_Q|}$ (see Definition 3.4). The problem becomes even harder when factors like the WWW's no-guarantees / open world assumption are taken into account. As a consequence, AVALANCHE extends the competitive plan execution paradigm to its logical conclusion attempting to execute all plans (or a window of a predetermined size) concurrently. Therefore, one of the main differences between traditional query optimization and AVALANCHE lays in the scope of employing the optimization methods. While commonly, query optimizers focus on finding the absolute best plan exhaustively, AVALANCHE does not attempt to ascertain a plan's absolute cost and instead uses the cost model as a ranking mechanism for all plans whether partial or complete. The simple rationale behind this strategy being that while a generalizable absolute cost model is difficult to obtain for the problem of on-demand querying on the WoD, *the concurrent execution of plans will have a higher probability of yielding results if "productive" plans are pushed early in the execution pipeline.*

As with any large search space there are two possible ways of traversal: either in a deterministic fashion or employing a stochastic algorithm. The AVALANCHE planner falls within the deterministic class. Although, a breadth-first traversal strategy will rank the plans starting with the best (according to the cost heuristic) the exponential space complexity can easily result in an indefinite waiting time to start the actual execution phase. This renders the approach impractical for many setups and workloads. In contrast, greedy strategies like depth-first traversal are better suited to the asynchronous nature of the Web due to their characteristic support for iterative execution models.

Hence, AVALANCHE's plan generator is designed as an *informed, best-first multi-path, depth* traversal algorithm. Borrowing from both breadth and depth search, the *Plan Generator* traverses the universe of all plans as outlined in Algorithm 1. Its main focus is that of generating query plans which are likely to produce results fast with a minimum of cost, in the order dictated by the *cost model.*

3.4.1. The cost model

Commonly, cost model functions can be classified into: (i) cost models that aim to reduce the total time to execute the query and (ii) cost models that strive to reduce the response time (or first result latency). The first class of cost models are in general pertinent to single query execution scenarios. Since a complete result set is not in AVALANCHE's scope the second class of cost functions is desirable. According to [31] the general comprehensive cost model takes into account all factors that influence the cost of the execution of a distributed query.

An exact approximation of the cost of a plan is not as crucial of an aspect as for traditional setups when the planner would search for a single best plan. This stringent

[25] As reported in [33], at the time (c. 2003) a number of 15 joins was considered prohibitive.

constraint is relaxed in AVALANCHE mainly due to the following factors: (a) AVALANCHE's *Plan Generator* exhaustively traverses the space of all plans and (b) AVALANCHE executes many plans concurrently at any given time. In consequence, since AVALANCHE needs to rank all generated plans according to their cost—not necessarily in an absolute fashion, but following the cost trend—two simplifying assumptions can be considered:

∘ *Network*: We assume that network latency and bandwidth are relatively uniformly distributed between participating sites. Although a gross approximation, the assumption holds true in most cases for geographically "near" sites. Furthermore, many participants on the WWW follow this assumption.

∘ *Distributed Joins*: A widely encountered phenomenon on the WoD, multi-tenancy gives rise to a number of difficulties and problems ranging from management of RDF data to query and index optimization both locally and at a global scale. Since AVALANCHE's scope is the indexed WoD, it is unrealistic to assume that full index statistical information is always available or can always be shared between participating sites. Therefore, in the absence of more exact and elaborate metrics and estimations join selectivity is estimated. The main advantages of this model are: (1) there is no need for joint distribution statistics to be available and (2) it bears virtually no computation and network cost. However, there are many fallacies introduced as it neither overestimates nor does it underestimate the actual join between any two BGPs.

In the following we discuss the impact these assumptions have on the cost model.

Selectivity estimation. In the absence of exact statistics regarding *triple patterns* and *basic graph patterns* selectivity is usually estimated. However, as AVALANCHE starts with the premise that triple pattern cardinalities are know as reported by `getTPCardinality` (Appendix A), triple pattern selectivities are computed and not estimated. For a given triple pattern $\boldsymbol{tp}$ bound to a given host $\boldsymbol{h}$ its selectivity represents the probability of selecting a triple that matches from the total number of triples involved and is thus directly computed as follows:

$$sel_{tp}^{h} = P_{match}(tp, h) = \frac{card(tp, h)}{T_{MAX}} \tag{1}$$

where T_{MAX} represents the total number of triples across all participating hosts as detailed in Definition 3.1.

Exact basic graph pattern selectivities require knowledge about join triple pattern cardinalities – a prohibitively expensive requirement considering its characteristic exponential space complexity. Most RDF database management systems with very few exceptions—and in limited cases only—proceed to estimating the selectivity of BGPs and AVALANCHE is no exception. However, in doing so AVALANCHE discriminates between star shaped graph patterns[26] and the rest. For simplicity we will later refer to them as *star graph patterns* or *stars*. Any given basic graph pattern *bgp* can be decomposed into the set of all contained stars referred to as $\mathbb{S}_{bgp}$ and a remainder graph pattern which contains all triple patterns that do not form stars called $\mathbb{NS}_{bgp}$. In consideration of the above, the selectivity of *bgp* is estimated according to the the following formula:

[26]Graph theoretic constructs, they materialize in the realm of SPARQL queries as groups of triple patterns that join on the same subject or object

$$SEL_{bgp}^{h} = \prod_{tp' \in \mathbb{NS}_{bgp}} sel_{tp'}^{h} \times \prod_{star \in \mathbb{S}_{bgp}} (\min_{tp'' \in star} sel_{tp''}^{h}) \tag{2}$$

The equation captures the intuition that non-star pattern triple-patterns are estimated via independent combination of their selectivities. Obviously, independence is not correct but oftentimes found as an acceptable approximation. The selectivity of a star pattern, in contrast is estimated by the selectivity of its minimal participating triple-pattern.

Cost model. When ranking plans, Avalanche employs a common no-preference multiobjective optimization method: the method of *Global Criterion* [43]. Avalanche uses this method as an envelope to combine the folowing heuristic objectives:

- **plan selectivity estimation**: this objective relies primarily on selectivity estimation as it appears in equations 1 and 2 and is defined according to the following equation:

$$SEL_{plan} = \prod_{sq \in SQ_{plan}} SEL_{bgp_{sq}}^{h_{sq}} \tag{3}$$

  where *plan* represents a partial or complete plan and SQ_{plan} is the set of subqueries in *plan*.
- **number of subqueries**: stemming from a *data-locality* assumption (related assertions are usually on the same host) this second heuristic is intended to bias the plan generator towards plans (or partial plans) that will result in query decompositions with fewer subqueries and is defined as follows:

$$SIZE_{plan} = |T_{plan}| - |SQ_{plan}| \tag{4}$$

  where *plan* represents a partial or complete plan, $T_{plan} = \{tp_i \mid tp_i \in plan\}$ the set of triple patterns in *plan*, and SQ_{plan} the set of subqueries in *plan*.

  Since Avalanche needs to compare partial plans with various degrees of completion whilst exploring the universe of all plans P_Q the number of subqueries is "normalized" by the number of triple patterns considered so far.

  Finally, Avalanche minimizes the cost of a plan by combining the previous heuristic functions according to the following equation:

$$COST_{plan} = || < SEL_{plan}, SIZE_{plan} > -z^{ideal} || \tag{5}$$

where $z^{ideal} =< 1, 0 >$ and the $||.||$ norm is the L_2 norm or the euclidean norm. A flexible design, the cost model $COST_{plan}$ supports pluggable heuristics by simply extending the heuristics vector and z^{ideal} accordingly.

Numerical stability. The issue of numerical stability is risen when dealing with very high selective triple or basic graph patterns in the context of large RDF spaces. To avoid numeric overflow effects, $COST_{plan}$ is computed in logarithmic space. Hence, the notion of *logarithmic probability* or in our case *logarithmic selectivity* is used:

$$logsel_{tp}^{h} = -log(sel_{tp}^{h}) \tag{6}$$

An added benefit, the cost computation is simplified since in logarithmic space the products from equations 2 and 3 are transformed into sums. Furthermore, since the method of global criterion is sensitive to the scaling of the considered objective functions, as recommended in [27] the objectives are normalized into the uniform [0,1] range.

3.4.2. Searching for the "best" plans.

As exhibited in Algorithm 1 the planner will try to optimize the construction of plans using an informed (best-first) multi-path depth traversal strategy. Therefore, plans are generated in ascending order by minimizing the heuristic cost-approximation function of each plan $COST_{plan}$, described in Equation 5. Stemming from the attempt to preserve the data-set distribution flexibility that publishers enjoy on the Web and therefore on the WoD, the plan composition space exposes an exponential worst case space complexity: $O(m^n)$, where $m = |H|$ is the number of involved sites and $n = |T_Q|$ is the number of triple patterns (number of joins - 1) in the query Q.

With each exploratory step the size of the exploration fringe $\mathbb{F}$ increases by the number of sites $|H|$ (line 19). This happens for each expanded state or partial plan represented by a $< tp, h_i >$ pair, where $tp \in T_Q$ is the current triple pattern and $h_i \in H$ a participating endpoint or host. The algorithm is complete and exhaustive as it iterates over all possible plans. While search algorithms normally stop and return when the *solution* is found, the PLANGENERATOR procedure is not halted and instead each solution or plan is **emitted** to the caller (line 11).

The traversal generator procedure, although classified as *informed* or *best-first* (BFS), borrows conceptually from *depth-first* search (DFS) algorithms. The DFS aspect is necessary in order to produce viable plans quickly and is encoded by the partial sort of the local fringe $\mathcal{N}$ in function NODES line 29, forcing the exploration of direct descendant partial plans of the current state. In contrast, the BFS aspect is triggered right after a plan has been emitted by sorting the entire fringe $\mathbb{F}$ (line 13). This is critical since the planner must select for expansion the next best plan available.

Pruning. As the exploration space grows quickly pruning invalid or $\emptyset$ plans is desired. Early pruning is achieved immediately after the *statistics gathering* phase when the plan matrix PM_Q is available, by removing all hosts (matrix rows) for which the cardinality of all triple patterns is 0. In the absence of triple-patten cardinalities, early pruning would not be possible and the maximum number of plans would have to be considered: $|H|^{|T_Q|}$.[27] Hence, queries that produce a $\emptyset$ plan matrix (where all elements are 0) are stopped during this early optimization step.

Furthermore, during execution the same join can be often shared by multiple competing plans. Consequently, joins that are $\emptyset$ (empty) are recorded and used as dynamic feedback for the planner, which then prunes any plan that contains an $\emptyset$ join. This aspect transforms the AVALANCHE planner into an *adaptive* planner as seen in line 26 of the NODES function.

[27]For a realistic example of plan space upper bound vs. possible plans after pruning we refer to Table 3 on page 28.

Algorithm 1 The plan generator algorithm

Precondition: Q a well-formed SPARQL query, $\mathbb{T}$ the set of all triple patterns $\in Q$
Postcondition: $\mathcal{N}$ a set of search nodes, P a query plan

```
procedure planGenerator($Q$)
    $\mathbb{V} \leftarrow \emptyset$                                    ▷ $\mathbb{V}$: set of visited nodes
    $\mathcal{C} \leftarrow \emptyset$                                    ▷ $\mathcal{C}$: set of closed nodes
    $\mathbb{F} \leftarrow$ nodes($\mathbb{V}$, $\mathbb{T}$, $\varnothing$)                 ▷ $\mathbb{F}$: active exploration fringe
    $\rho \leftarrow 0$                                    ▷ $\rho$: current plan counter
    while $\mathbb{F} \neq \emptyset$ do
        if $\rho = MAX_{plans}$ then             ▷ $MAX_{plans}$: maximum number of plans
            break
        $best \leftarrow \mathbb{F}$.pop()
        if solution($best$) then                         ▷ $best$ is a leaf search node
            emit Plan($Q$, $best$, $\rho$)                        ▷ emit generated plan
            $\rho \leftarrow \rho + 1$
            $\mathbb{F}$.sort()                         ▷ sort fringe $\mathbb{F}$ based on $COST$
        if $best \notin \mathcal{C}$ then
            $\mathcal{C} \leftarrow \mathcal{C} \cup \{best\}$
            $T_{nxt} \leftarrow \{tp\}$, $tp \in \mathbb{T} \wedge tp \notin$ triplePatterns($best$)
            if $T_{nxt} = \emptyset$ then    ▷ $T_{nxt}$: next unexplored triple pattern in partial plan
                continue
            $\mathbb{F} \leftarrow \mathbb{F} \cup$ nodes($\mathbb{V}$, $T_{nxt}$, $best$)                 ▷ expand search space

function nodes($V$, $T$, $parent$)                      ▷ partial fringe expansion function
    $\mathcal{N} \leftarrow \emptyset$              ▷ $\mathcal{N}$: the nodes, $V$: visited queue, $T$: a set of triple patterns
    for $tp \in T$ do
        for $h \in H$ do                                 ▷ $H$: the set of all endpoints
            $n \leftarrow$ node($tp$, $h$, $parent$)                           ▷ create new node
            if $n \notin V \wedge n \neq \varnothing$ then
                $V \leftarrow V \cup \{n\}$
                $\mathcal{N} \leftarrow \mathcal{N} \cup \{n\}$
    $\mathcal{N}$.sort()                        ▷ sort (partial fringe) $\mathcal{N}$ based on $COST$
    return $\mathcal{N}$
```

3.5. Executing Plans

Since in Avalanche state is managed remotely, the broker is only tasked with the orchestration of the overall query execution process. As such, the broker is primarily an I/O bound process with little need for multicore parallelism. In this sense the use of socket-asynchronous system design is indeed a natural match for its implementation.

Once a plan is emitted by the *Plan Generator* it is directly assigned to a new executor worker in the asynchronous *Executor Pool*. Asynchronous concurrent pools can support orders or magnitude more workers in comparison to traditional thread or process based pools limited by the relatively low number of system threads / processes. If the number of open sockets is too high for asynchronous communication concurrency can be throttled down (just like in traditional thread-pools).

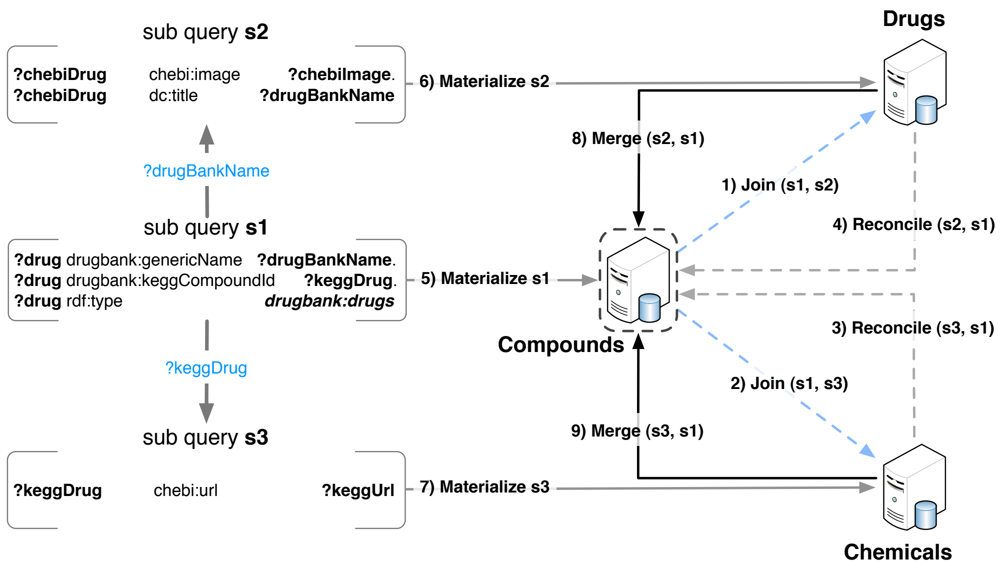


Figure 5: Graphical illustration of the execution process for example query $Q_{ex.}$.

As soon as a plan is assigned to a worker the process described in Algorithm 2 unfolds. Figure 5 illustrates this process for the query Q_{ex}.

A first step consists of sorting the subqueries (if more than 1) in order of their selectivity estimation SEL^h_{sq} on the designated host h. The distributed join is than executed in left-deep fashion, starting with the most selective subquery as see in line 5 and steps 1 and 2 in figure. Necessary for the next phase, the order in which joins occurred is recorded in the J_Q queue. The next phase is optional, since it's an optimization. When enabled, the partial results that have been produced in the earlier join can be reconciled (filter out the pairs that do not match) in reverse order of their counter-part joins (line 6, steps 3,4 in figure). Reconciliation can be naive (send the entire set compressed or not) or optimized. The former is used when the cost of creating the optimized structures is higher than just sending the set. In the latter hashes can be send when the item size is larger than its hash or following [34] *bloom-filters* can be employed. Bloom-filters are space-efficient lossy bit-vector representations of sets by virtue of using multiple hash functions for recording each element.

Finally results are materialized in parallel (line 7, steps 5,6,7 in figure) and then merged on the host corresponding to the first subquery – the one with the lowest estimated selectivity, (line 8, steps 8,9 in figure). To increase execution performance, since many plans contain the same or overlapping subqueries, a *memoization* strategy is employed. Hence, partial results are kept for the duration of the entire query execution and not just for the current plan. This acts as a site-level cache memory bypassing the database altogether for "popular" result sets when resources permit.

When the merge is completed, the *Plan Executor* worker process will signal the Avalanche *Query monitor* via the *Results Queue*. Note that the finished plans do

Algorithm 2 The plan executor

Precondition: P a valid query plan, R_Q the AVALANCHE *Results Queue*

```
1: procedure EXECUTEPLAN($P$, $R_Q$)
2:     $r \leftarrow \emptyset$                                  ▷ $r$: the results
3:     if ISFEDERATED($P$) then                    ▷ $P$ has more than 1 subqueries
4:         SORTSUBQUERIES($P$)                     ▷ Sort subqueries in $P$ by $SEL^h_{sq}$
5:         $J_Q \leftarrow$ DISTJOIN($P$)              ▷ distributed join of subqueries
6:         DISTRECONCILIATION($J_Q$)                ▷ reconcile partial results
7:         $\mathbb{S}_Q \leftarrow$ DISTMATERIALIZE($P$)   ▷ distributed materialization (||)
8:         $r \leftarrow$ DISTMERGE($\mathbb{S}_Q$)          ▷ merge partial results
9:     else
10:        $r \leftarrow$ SPARQL($P$)                    ▷ execute SPARQL query
11:    $R_Q \leftarrow R_Q \cup r$                       ▷ append results
```

not contain the final results, as the matches are kept remotely. It is the *Query monitor's* responsibility to retrieve the results and update the overall state of the broker accordingly. In the remainder of this subsection we will describe in detail the inner-workings of the operations described above.

Distributed Join & Reconciliation. The join and reconciliation procedures are detailed in Algorithms 3 and 4 respectively. Joining is implemented in a left-deep fashion while the reconciliation procedure is straight-forward. One important aspect to note is that the execution of a plan can be stopped (line 6 in Algorithm 4) if the cardinality of a join is 0. This information is recorded and fed back into the planner for dynamic pruning.

Algorithm 3 The distributed join operation

Precondition: P a valid query plan
Postcondition: J_Q a queue, containing the joins in order

```
1: function DISTJOIN($P$)
2:     $J_Q \leftarrow \emptyset$                                ▷ $J_Q$: joins queue
3:     $\mathbb{S} \leftarrow$ SUBQUERIES($P$)                ▷ $\mathbb{S}$: set of all subqueries in $P$
4:     $\mathbb{S}$.sort()                          ▷ sort by selectivity estimation $SEL$
5:     if ISFEDERATED($P$) then
6:         while $\mathbb{S} \neq \emptyset$ do
7:             $best \leftarrow \mathbb{S}$.pop()
8:             for $sq \in \mathbb{S}$ do
9:                 $best \bowtie sq$                              ▷ remote join
10:                $J_Q \leftarrow J_Q \cup \{[best, sq]\}$               ▷ record join
11:    else
12:        SPARQLREMOTE($P$)             ▷ SPARQL query but keep results remotely
13:    return $J_Q$
```

Algorithm 4 The distributed reconciliation operation

Precondition: J_Q joins queue

```
procedure distReconciliation($J_Q$)
    $J_Q$.reverse()
    for [$left, right$] $\in J_Q$ do
        $\kappa \leftarrow$ reconcile($left$, $right$)          ▷ remote reconciliation step
        if $\kappa = 0$ then
            halt                       ▷ stop plan execution when cardinality = 0
```

Distributed Materialization & Merge. The final execution phases are detailed in Algorithms 5 and 6 respectively. The simple materialization procedure is executed in parallel on all subquery hosts with the important note that locally kept selectivity estimations for each subquery in $\mathbb{S}_Q$are updated to actual join cardinalities, available at this stage remotely (line 5 in Algorithm 5). This information is later used to find out the host with the highest partial result cardinality. This host (*best* in line 2 in Algorithm 6) is than used as the "hub" on which all other partial results are merged (lines 3-5 in Algorithm 6).[28]

Algorithm 5 The distributed materialization operation

Precondition: P a plan

Postcondition: $\mathbb{S}_Q$a queue containing the plan subqueries sorted by cardinality κ

```
function distMaterialize($P$)
    $\mathbb{S}_Q \leftarrow \emptyset$                                  ▷ $\mathbb{S}_Q$: subqueries queue
    for $sq \in P$ do
        $\kappa \leftarrow$ materialize($sq$)        ▷ $\kappa$: the cardinality of partial results on $sq$
        $\mathbb{S}_Q \leftarrow \mathbb{S}_Q \cup \{[\kappa, sq]\}$
        if $\kappa = 0$ then
            halt                       ▷ stop plan execution when cardinality = 0
    $\mathbb{S}_Q$.sort()                                          ▷ sort by $\kappa$
    return $\mathbb{S}_Q$
```

3.6. Stopping the query execution

Since we have no control over distribution and availability of the RDF data and SPARQL endpoints, providing a complete answer to the query is an unreasonable assumption except for the cases involving few endpoints and rather simple queries. Instead, the *Query Monitor / Stopper* monitors for the following *stopping conditions*:

- a global timeout set for the whole query execution,
- returning the *first K* unique results to the caller,

[28]For brevity and graphical simplicity of Figure 5, the "Compounds" endpoint (in the middle) was also assigned to be the merge host.

Algorithm 6 The distributed merge operation

Precondition: $\mathbb{S}_Q$ subqueries queue
Postcondition: r a valid SPARQL results set

```
1: function DISTMERGE($\mathbb{S}_Q$)
2:     $\kappa$, $best \leftarrow \mathbb{S}_Q$.popLeft()        ▷ $\kappa$: the cardinality of partial results on $best$
3:     while $\mathbb{S}_Q \neq \emptyset$ do
4:         $sq \leftarrow \mathbb{S}_Q$.popLeft()
5:         MERGE($best$, $sq$)                               ▷ merge results from $sq$ on $best$
6:     $r \leftarrow$ GETRESULTS($best$)                     ▷ retrieve the final results from $best$
7:     return $r$
```

∘ to avoid waiting for the timeout when the number of results is $\ll$ K we measure relative result-saturation. Specifically, we employ a sliding window to keep track of the last n received result sets. If the standard deviation (σ) of these sets falls below a given threshold then we stop execution. Specifically, using Chebyshev's inequality [21] we stop when $1 - \frac{1}{\sigma^2} > 0.9$.

4. Evaluating Avalanche's Robustness Against Messiness

In the introduction we claimed that the AVALANCHE system provides the capability to query the messy Web of Data. Specifically, we claimed that the proposed system: (1) *makes no assumptions about data distribution, schema, availability, or partitioning* and is skew resistant for some classes of queries, (2) provides *up-to-date results* from distributed indexed endpoints, (3) is *adaptive* during execution adjusting dynamically to external network changes, (4) *does not require detailed fine-grained ex-ante statistics* with the query engine, and (5) is *flexible* as it makes limited assumptions about the structure of participating triple stores.

AVALANCHE is able to provide up-to date results without any ex-ante statistics (2 and 4) by accessing participating triple-stores at run-time and is open due to the limited assumptions it makes on triple-stores (5). Whilst skew resistance (1) and adaptiveness (3) seem possible due to its multi-plan competitive planing/execution strategies (see Sections 3.4 and 3.5) it has not been shown that these strategies are actually successful.

In the following we will describe the experimental evaluation of the AVALANCHE system. Specifically, we will provide empirical evidence ascertaining AVALANCHE's planner quality and the system's overall robustness to varying data distributions and network conditions like different latencies and endpoint unreliability. Specifically, we evaluate AVALANCHE's planer quality as well as robustness against network latency and endpoint stability (in Section 4.1) using a real world dataset. In addition, we will show AVALANCHE's robustness against various data distributions (Section 4.2) using a synthetic dataset.

Experimental setup. For all experiments a cluster of 6 physical machines with 64GB of RAM, 24 AMD Opteron 6174 Cores @2.2 GHz, and running Debian GNU/Linux 6.0.6 64bit was used. In addition the AVALANCHE broker was executed on a separate machine with 72GB of RAM, 8 Intel(R) Xeon(R) CPU X5570 Cores @2.93GHz, and running Fedora release 12 (Constantine) 64bit. For all evaluations we will first succinctly introduce the setup and then discuss the evaluated characteristics.

4.1. Evaluation Setting I: Analyzing AVALANCHE *with real-world data*

To evaluate the generalizability of our results to a real-world setting we chose a real-world dataset specifically tailored for the evaluation of federated RDF stores. This subsection first outlines the dataset, its distribution to hosts, the queries used and then discusses AVALANCHE's execution results on this dataset.

The Data and its Distribution. We chose the recently published Fedbench[29] [36] dataset as it comes pre-partitioned using a real-world partitioning schema and, additionally, offers 36 SPARQL queries. For summarized statistics about each participating dataset refer to Table 1.[30]

Table 1: Fedbench datasets statistics

Collection	**Dataset**	**version**	**# triples**	**Dataset**	**version**	**# triples**
Cross Domain	DBpedia subset	3.5.1	43.6M	Jamendo	2010-11-25	1.05M
	NY Times	2010-01-13	335k	GeoNames	2010-10-06	108M
	LinkedMDB	2010-01-19	6.15M	SW Dog Food	2010-11-25	104k
Life Sciences	DBpedia subset	3.5.1	43.6M	Drugbank	2010-11-25	767k
	KEGG	2010-11-25	1.09M	ChEBI	2010-11-25	7.33M
SP^2Bench	*SP^2Bench* 10M	v1.01	10M			

[a] Data available from http://code.google.com/p/fbench/wiki/Datasets

Following the natural partitioning of the benchmark we adopted the assumption that each dataset is published on its own distinct server. For bigger datasets such as Geonames and DBPedia we assumed in addition that the publishers decided to further split the data into multiple RDF stores. We captured this by splitting some of the larger datasets as detailed in Table 2. Hence, additional distribution messiness was introduced by splitting the Geonames triples randomly over 11 hosts while for DBpedia larger dumps were distributed to single hosts and the smaller ones were integrated into the *Other* AVALANCHE endpoint.

The Queries. The triple store[31] we used for implementing AVALANCHE endpoints does not currently not support SPARQL features beyond traditional BGP pattern matching. Hence, we ignored all Fedbench queries that contain the *OPTIONAL* and *FILTER* graph pattern modifiers. This is a limitation of the current system and evaluation, which we discuss in detail in Section 5. Additionally, as *UNION* is not supported either, queries containing the operator were split and executed as separate queries, which is aligned

[29]http://code.google.com/p/fbench

[30]Another viable option would be the Billion Triple Challenge 2011 dataset. Whilst one could define splits based on the provenance of each triple (group triples originating from the same source/publisher in the same store) it does, however, lack a standard agreed upon set of queries suited for the evaluation of distributed triple stores.

[31]An in-house and updateable extension of Hexastore was used as the RDF store technology behind all AVALANCHE endpoints in our evaluations.

Table 2: The distribution of the Fedbench dataset to AVALANCHE hosts

Dataset	**Avalanche Host**	**#triples**	**Dataset**	**Avalanche Host**	**#triples**
NY Times	News	314k	LinkedMDB	Movies	6.14M
Jamendo	Music	1.04M	SW Dog Food	SW	84k
KEGG	Chemicals	10.9M	ChEBI	Compounds	4.77M
Drugbank	Drugs	517k	SP2B-10M	Bibliographic	10M
Geonames	Geography_1	9.98M		Geography_7	9.95M
	Geography_2	9.99M		Geography_8	9.99M
	Geography_3	9.93M		Geography_9	9.99M
	Geography_4	9.94M		Geography_10	9.99M
	Geography_5	9.98M		Geography_11	7.98M
	Geography_6	9.98M			
DBPedia subset	Infobox_Types	5.49M		Infobox_Properties	10.80M
	Titles	7.33M		Articles_Categories	10.91M
	Images	3.88M		SKOS_Categories	2.24M
	Other	2.45M			

with the common practice of executing unions as individual subqueries in parallel. In addition to the resulting 33 queries we supplemented the Fedbench queries with another 5 more complex queries from the life sciences domain, as listed in Appendix D. The translation table to the original names (where applicable) is available in Appendix C.

4.1.1. Experiment #1: AVALANCHE *vs. Baseline System*

In order to observe the performance gains characteristic to the execution model proposed in AVALANCHE we implemented a "baseline" query execution pipeline as described in the following. The core idea behind the baseline approach is to essentially gather relevant data from the distributed sources and efficiently recompose the partial results obtained in this fashion locally.

A naive approach would be to multicast the query Q to all participating sites but allow them to answer only those parts (triple patterns) of the query that the endpoint actually "understands", in effect mapping $Q \mapsto Q_{known}$. The decision to discard triple patterns is made locally by the RDF store itself and is implemented as defined in Equation 7:

$$T_{Q_{known}} = \{tp_i \mid \forall tp_i \in T_Q \iff card(tp_i, h) > 0, h \text{ is the current endpoint}\} \tag{7}$$

where $T_{Q_{known}}$ represents the "known" set of triple patterns composing query Q_{known}.[32]

Since traditional SPARQL query execution assumptions are broken when executing queries in the manner mentioned earlier, the baseline engine is left with the non-trivial task of recombining the partial results. A further complication arises from the fact that a SPARQL SELECT query is essentially a $G \mapsto R$ mapping, where G is the original graph data and R represents the results table with the query projection variables as column names. To make sense of the incoming partial result tables R_i the reverse mapping $G_i \leftarrow R_i$ needs to be performed. The resulting graph $G_{known} \leftarrow \bigcup R_i$ would then

[32]Depending on the indexing mechanisms used by the RDF store, other triple pattern exclusion rules can be imagined (i.e: discard triple patterns if the predicate belongs to an unknown namespace – provided namespace information is available).

contain all the original triples stemming from the partial result tables. The engine is now left with the task of re-executing the original query Q on the local graph G_{known}.

Limitations. While conceptually simple, a number of hurdles render the implementation non-trivial. First, it is possible that some of the reduced queries Q_{known} do not contain any of the selective triple patterns from Q (if any). These subqueries usually appear when executing distributed query plans with one major difference, they are associated with other subqueries with which they usually join and their execution is deferred in favor of the more selective ones. Furthermore when executed partial results bindings are supplied resulting in increased selectivity. In the worst case $Q_{known} \equiv < s, p, o >$ which in turn attempts to retrieve the entire remote knowledge-base, a prohibitively expensive operation for both requester and provider. Second, since the final results for Q can only be computed after obtaining G_{known} two execution strategies emerge:

i) Wait until all R_i partial results are retrieved and then execute Q on G_{known}. This is suitable for situations where partial results are inexpensively obtained and/or the query is complex.

ii) Every time a partial result R_i arrives, merge it with G_{known} and execute query Q retaining the results. This pipelined strategy obviously pays off when (some) partial results are expensive to obtain additionally offering the possibility of an early stop when Q is satisfied without having to wait for all partial tables. However, it incurs the cost of executing Q with each retrieved partial table.

Finally, the method is not complete due to the following reason: the resulting Q_{known} can remove some triples from the result set, triples which may be match points with other remote endpoints. In contrast AVALANCHE is complete since it considers all possible decompositions of Q and not just one decomposition like Q_{known}.

Results. Based on the assumption that the selectivity distribution of the generated Q_{known} subqueries on participating endpoints is ZIPF-ian, we chose to implement the pipelined execution model due to its obvious performance benefits. Furthermore, the same asynchronous behavior as in AVALANCHE was encoded in the baseline, while G_{known} was implemented by a fast loading and high performance in memory indexed RDF store[33]. A consequence of this choice is that the same stopping conditions that AVALANCHE employs can be used to determine wether the engine should stop the query execution or not, hence, eliminating other unknown hidden factors when comparing the two systems.

The time taken to complete all the considered Fedbench queries by both systems is graphed in Figure 6. With very few exceptions AVALANCHE proved to be faster than the *baseline* system. When retrieving first results the baseline system is slower than AVALANCHE in 65% of the queries, becoming slower for 92% of the queries by the time total[34] results are retrieved. This is better captured in Figure 7, where the geometric mean over all queries is computed. Clearly, for the 38 selected Fedbench queries AVALANCHE exhibits superior average performance for both cases: retrieving first results and achieving query completion.

[33]We used the IOMemory RDF store provided by the `rdflib` package: https://github.com/RDFLib

[34]By total we refer to the number of results retrieved until the system is stopped by the *Results Monitor*.

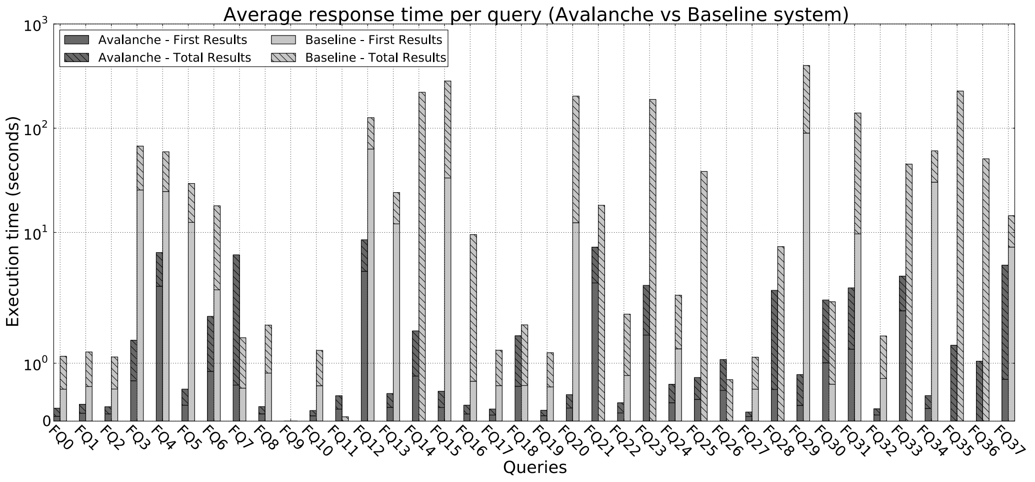


Figure 6: Average query execution times for each of the Fedbench queries. Avalanche vs. the Baseline System.

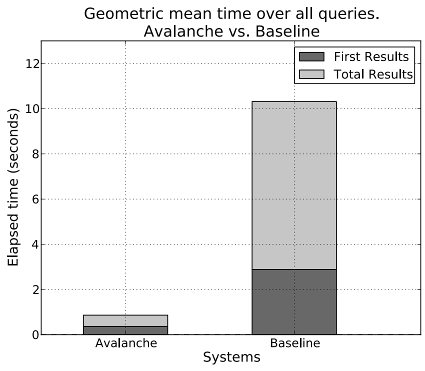


Figure 7: Geometric mean of the execution time over all queries: Avalanche vs. the Baseline System.

Furthermore, as mentioned previously the baseline system is not guaranteed to be complete, a fact exhibited by queries: $FQ11$, $FQ14$, $FQ21$, $FQ23$, $FQ25$, $FQ26$, $FQ28$ and $FQ33$ as seen in Figure 8, which depicts the recall for all queries. The ground-truth used to compute the recall was obtained by disabling the *Results Monitor* component, thus, enforcing none of the stopping conditions.

The baseline system although slower in most cases and incomplete in some, exhibits some positive properties. First, it is of a much more simple design and finally for some classes of queries it can be faster than Avalanche. For example for query $FQ30$ first results are retrieved with a negligible 0.37 seconds faster (a difference that can be attributed to the statistics gathering phase in Avalanche), while the same happens for total results in query $FQ7$ which completes 4.6 seconds faster than Avalanche.

In light of these results, we can safely say that Avalanche exhibits significant performance and conceptual benefits over the naive baseline system.

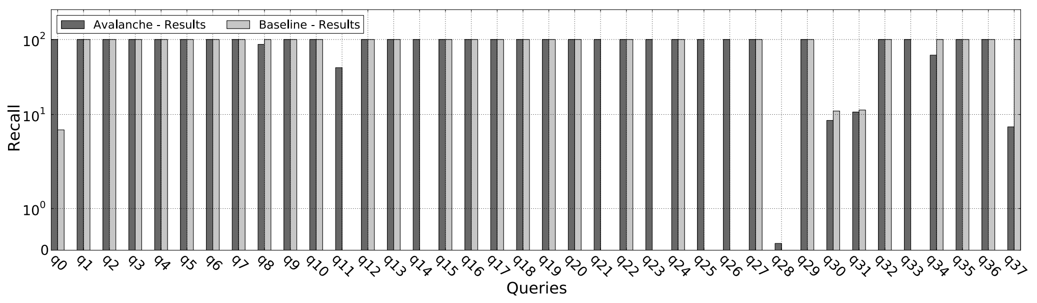


Figure 8: Recall for each of the Fedbench queries. AVALANCHE vs. the Baseline System.

4.1.2. Experiment #2: Planner Quality Assessment

As described in Section 3.4 the AVALANCHE planning strategy uses heuristics to explore the universe of all plans P_Q. In the following, we will analyze the quality of the planning algorithm used in AVALANCHE by *1*) comparing it to a perfect omniscient (or oracle) planner and *2*) by observing the relative ranking of productive plans within the query plan universe P_Q.

Comparison to a perfect planner. To construct an omniscient or perfect planner, we used the following strategy. In a first run, AVALANCHE recorded (serialized) "productive" plans to a local storage, while in a second run a custom planner component—referred to as the *Perfect Planner*—was employed. Its role was to generate only productive plans for the given query, allowing AVALANCHE to execute them in the exact same manner as before. This setup allows us to simulate an *oracle planner*.[35]

The results of running all 38 Fedbench queries comparing the standard AVALANCHE planner (labeled *dfs*) with the *oracle planner* are depicted in Figure 9. Over all queries AVALANCHE was 0.37 seconds slower on average when retrieving first results and 0.76 seconds slower on average when retrieving all results (i.e., completing the query execution). If we split the depicted answer times into two groups—one for queries that are at most 1 second slower when answered with the AVALANCHE heuristic planner (a total of 27 queries) and another for queries that are more than one second slower (the remaining 11 queries)—then we observe that for the first group AVALANCHE produces first results with an average of 0.14 seconds extra latency (respectively 0.11 seconds extra latency for total results) than in the perfect planner case. The difference is more dramatic for the second group of queries, where AVALANCHE produces first results with an average of 1.92 seconds extra latency and 2.35 seconds extra latency for total results. The largest differences are recorded for the Fedbench queries $FQ4$, $FQ7$, $FQ21$ and $FQ37$ (Listings 20, 23, 37 and 53) and is explained by a higher plan-rank assigned by the AVALANCHE planner to the productive plans and/or interfering non-productive plans that consume system resources needlessly. Moreover, since queries $FQ35$ and $FQ36$ produce no results the perfect planner returns immediately while the AVALANCHE planner continues to search for a productive plan coming to a halt after about one second.

[35]A plan generator connected to an oracle, akin to an *oracle machine*, i.e., a Turing machine connected to an oracle

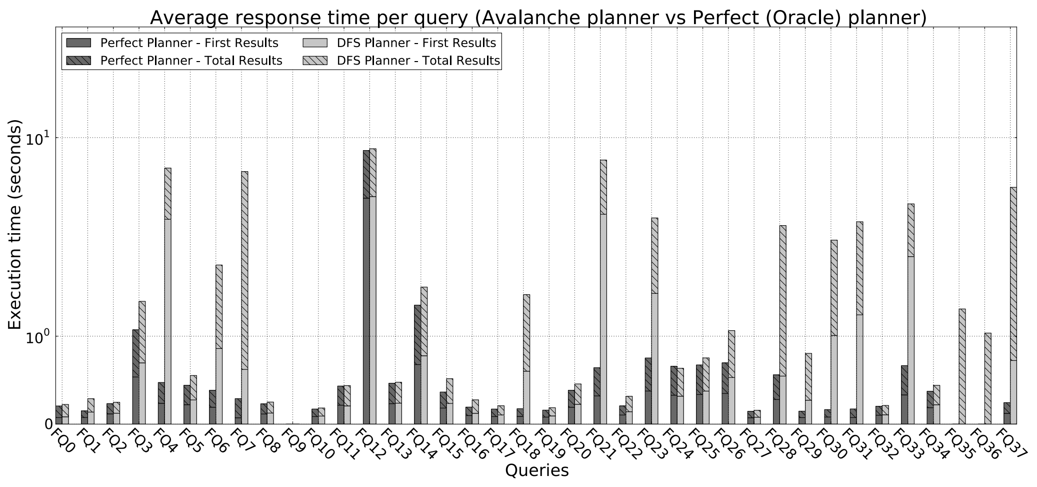


Figure 9: Average query execution times for each of the Fedbench queries. Avalanche standard dfs planner vs. oracle planner.

In general this difference is to be expected. The effort of discarding (and executing) unproductive plans in conjunction with the plan space exploration takes time. Hence, the Avalanche planner is naturally slower than a no-effort planner (like the oracle planner) is used. However, as exhibits by Figure 9 the delays are clearly limited and acceptable to many applications. Hence, Avalanche exhibits a good performance in the conditions of this evaluation when acting based on join-estimate heuristics.

Table 3: Total possible plans and first productive plan rank as generated by Avalanche

query	*FQ0*	*FQ1*	*FQ2*	*FQ3*	*FQ4*	*FQ5*	*FQ6*	*FQ7*	*FQ8*	*FQ9*	*FQ10*
max plans[b]	26^1	26^2	26^3	26^5	26^5	26^4	26^4	26^4	26^1	26^1	26^1
# plans[c]	6	26	1	18	324	18	180	2592	1	0	1
1st plan	**1**	**25**	**1**	**1**	**2**	**3**	**23**	**48**	**1**	-[a]	**1**
query	*FQ11*	*FQ12*	*FQ13*	*FQ14*	*FQ15*	*FQ16*	*FQ17*	*FQ18*	*FQ19*	*FQ20*	*FQ21*
max plans[b]	26^2	26^5	26^7	26^6	5^{26}	26^3	26^3	26^4	26^5	26^3	26^5
# plans[c]	26	18	1	10	10	7	1	126	1	5	594
1st plan	**2**	**6**	**1**	**6**	**2**	**7**	**1**	**20**	**1**	**1**	**71**
query	*FQ22*	*FQ23*	*FQ24*	*FQ25*	*FQ26*	*FQ27*	*FQ28*	*FQ29*	*FQ30*	*FQ31*	*FQ32*
max plans[b]	26^2	26^5	26^3	26^3	26^5	26^3	26^9	26^5	26^2	26^2	26^1
# plans[c]	24	180	1	18	49	1	270	45	104	104	1
1st plan	**9**	**1**	**1**	**2**	**1**	**1**	**1**	**1**	**27**	**20**	**1**
query	*FQ33*	*FQ34*	*FQ35*	*FQ36*	*FQ37*						
max plans[b]	26^{16}	26^{12}	26^{16}	26^{11}	26^6						
# plans[c]	18	1	10	18	324						
1st plan	**6**	**1**	-[a]	-[a]	**17**						

[a] query has no results
[b] maximum number of plans if no triple-pattern cardinalities are available ≡ upper bound
[c] maximum number of possible plans deduced when triple pattern cardinalities are considered

Plan ranking. As can be seen in absolute values in Table 3 and normalized relative to total number of plans in Figure 10 Avalanche succeeds in assigning a low rank (1 ≡ best rank) to the first productive plan. When the number of possible plans is large, the simple selectivity-estimation-based cost model will assign higher ranks, as is the case of query $FQ21$ where the first productive plan is the 71^{st} plan generated out of 594 possibilities. However, due to the asynchronous-concurrent manner in which plans are executed, the negative effect of assigning higher ranks to plans[36] is mitigated to a relatively high degree as shown in the previous analysis agains the perfect planner.[37]

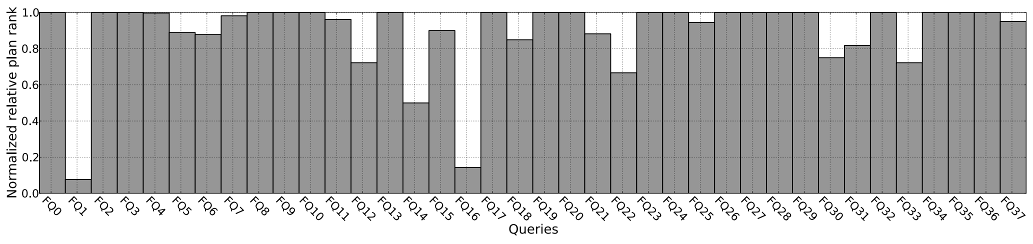


Figure 10: Normalized relative plan ranking: first plan compared to the possible number of plans / query for each Fedbench queries.The higher the bar the closer the first proposed plan is to the optimal one.

4.1.3. Experiment #3: Varying Network Latency

Changing network conditions can impede the execution of any distributed SPARQL processing. Two critical network factors stand out: bandwidth and latency. Since the slow-down effect of a low-bandwidth connection can in general be overcome with a certain degree of success by either compressing the message or making use of binary communication protocols and since AVALANCHE employs *bloom filter* optimized joins to reduce communication I/O, we decided to focus our attention in this experiment on connection latency. The majority of requests in the AVALANCHE system are between the AVALANCHE broker and the participating endpoints. Hence, for this experiment the connection between the broker and each endpoint was routed through a TCP delayer proxy, which would introduce delays according to a predefined configuration. We chose to simulate three types of latency distributions:

- *No Delay* $\rightarrow$ a *local cluster* network with negligible connection latency,
- *Gamma 1* $\rightarrow$ a *fast network* with an average connection latency of 0.3 seconds. Simulated by a gamma distribution with $\alpha = 1$ & $\beta = 0.3$ (Figure 11),
- *Gamma 2* $\rightarrow$ a *slow network* with an average connection latency of 3 seconds. Simulated by a gamma distribution with $\alpha = 3$ & $\beta = 1.0$ (Figure 11).

Additionally, the TCP socket buffer size was set to the standard value of 16KB.

AVALANCHE successfully finds results for all the considered benchmark queries under all simulated latency variations. Looking at Figure 12 we can clearly observe that the speed with which AVALANCHE answers queries across the different connection types increases dramatically as we move towards slower connections like *Gamma 2*. First, results are produced after an average of 0.36 seconds when connection latency is negligible, while for the *Gamma 1* and *Gamma 2* cases first results are found after an average of

[36]The rank is equivalent to the order in which a plan is generated, higher ranks imply a larger delay until plan execution.

[37]Non-productive plans are quickly discarded after the first empty join.

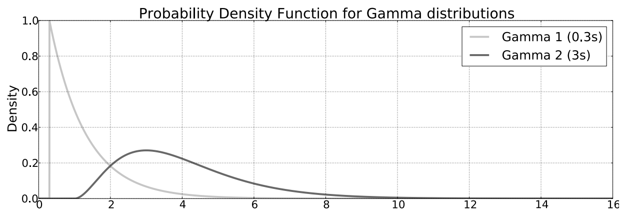
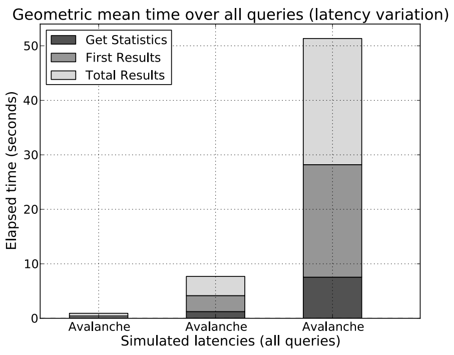


Figure 11: Propability desnity function (pdf) for the simulated *Gamma* 1 and *Gamma* 2 latency distributions.

2.93 seconds and 20.64 seconds respectively. The situation is similar for achieving the stop condition (called total query results[38]): 0.49 seconds on average for the *No Delay* setup, 3.52 and 23.15 seconds on average for the *Gamma 1* respectively *Gamma 2* setups. Although this performance decrease is dramatic AVALANCHE exhibits a sub-linear slowdown as graphed in Figure 13 compared to the broker-endpoints average latency slowdown.



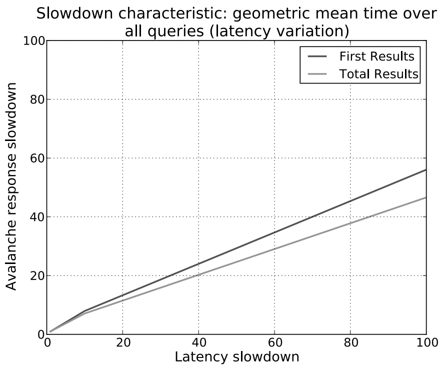


Figure 12: Geometric mean of the execution time over all queries for the three connection setups.

Figure 13: Slowdown introduced by the three connection setups.

This behavior is attributed to AVALANCHE mainly because of its adaptive asynchronous design. In essence plans that return quickly are favored by the asynchronous scheduling *Results Queue*. As a consequence, AVALANCHE is largely dependent on the *critical plan* for first results. The *critical plan* should ideally be the first productive plan. However, given that network conditions are uncontrollable, a slower plan might produce results faster because it shares a faster network connection. This is also observed in Figure 14, where the individual average times for answering all Fedbench queries $FQ_i, i \in [0, 37]$ queries under all three network conditions are graphed. As the broker-endpoints connections experience more lag, AVALANCHE exhibits a stable behavior overall depending mainly on the *critical plan(s)*, albeit slower with the slowdown depicted in Figure 13.

[38]Not to be mistaken with the ground truth result set for the respective query.

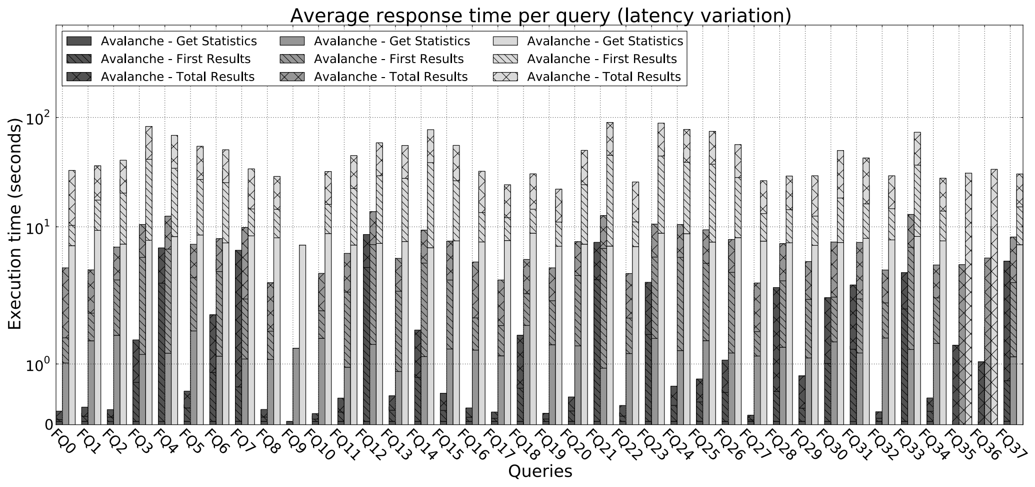


Figure 14: Average response time for each Fedbench query under different latency distributions. The graph differentiates between the time necessary to get the statistics, execute the first plan, and execute all plans.

4.1.4. Experiment #4: Varying Endpoint Availability

Another source of messiness stems from the uncontrollable nature of the underlying communication protocol stacks on the Web as well hardware and physical crashes of servers and routers. There is no guarantee that a host replying to requests at any given moment T will be available at time $T + \Delta t$. To observe the behavior of Avalanche in such a case we have designed an experiment. where some hosts disappear during query execution.

First, in order to have multiple plans per query we replicated some of the Fedbench endpoints used throughout this experimental setup. Specifically, we replicated the *News, Movies* and *Music* in the Cross Domain collection and *Drugs* in the Life Sciences collection Avalanche endpoints (see Table 2). This resulted in the increase of total number of triples over all hosts by about 8 million additional assertions. Furthermore, the already burdened physical machines had to support the 4 additional replicated endpoints.

Then, to emulate a crash the replicated endpoints were started in a "fail" mode, meaning that they would abruptly terminate themselves immediately after reporting the triple pattern cardinalities. This case is most interesting as the hosts will be considered by the *Query Planner* component as it received cardinalities from them, even-though all query plans containing subqueries allocated to them will fail to execute. The two other cases—the host being unavailable during either the source selection or statistics gathering phase—are less interesting as they handled by design (i.e., the hosts are not even considered in the planing). We compared Avalanche when replicated hosts would fail seamlessly during query execution with the case when the replicas would not fail. Note that the obtained results should not be directly compared to results obtained elsewhere in this section, as the Avalanche endpoints were simulated on some of the physical nodes, which experience additional load in this replicated setting.

Figure 15 graphs the arrival time of the first and total results for the cross domain and life sciences queries ($FQ_i, i \in [0, 15] \cup [33, 37]$) and Figure 16 graphs the average

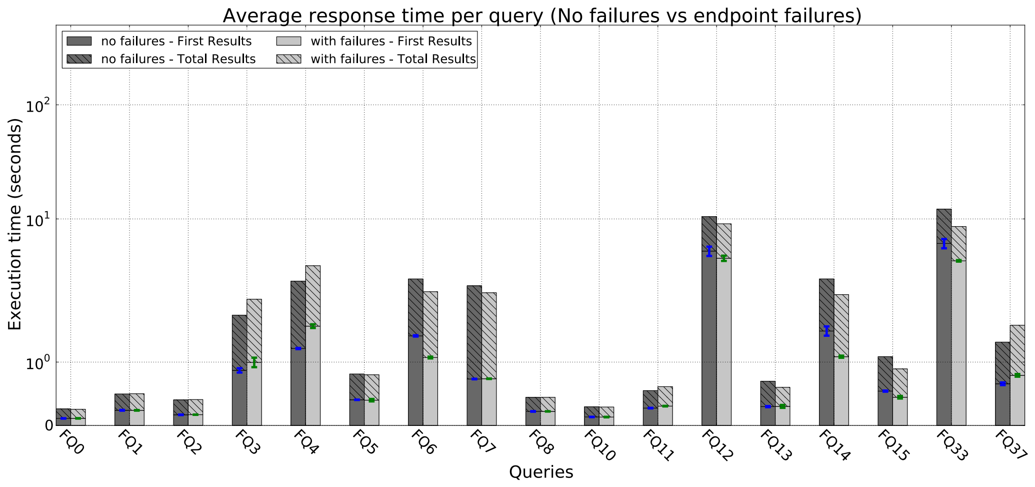


Figure 15: Average response time for Cross Domain and Life Sciences Fedbench queries when endpoints fail.

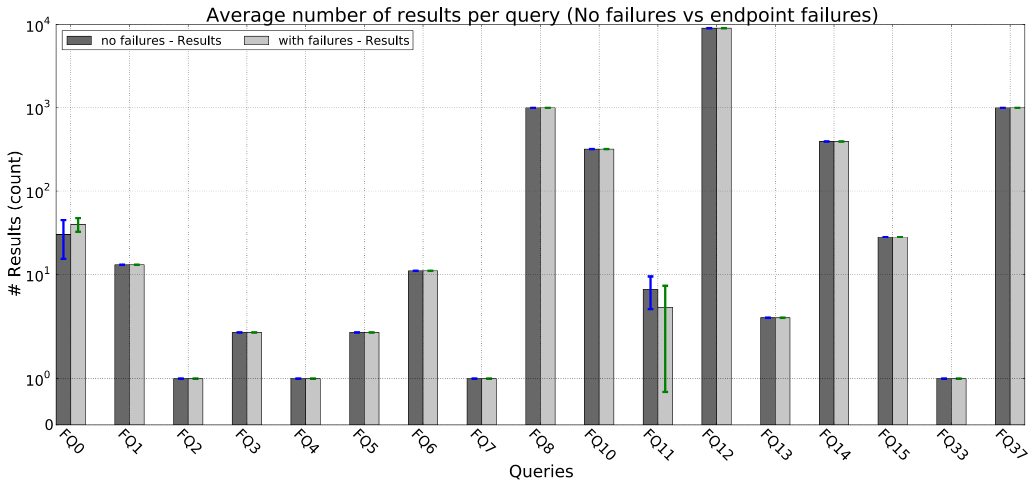


Figure 16: Average # of results time for Cross Domain and Life Sciences Fedbench queries when endpoints fail.

number of results obtained over the same queries.[39] AVALANCHE's *Plan Generator* adapts dynamically to external changing conditions, such as endpoints going offline, due to various reasons. Such events are usually detected when a plan that contains at least one subquery assigned to an offline host is executed. Upon detection, the planner's internal state is dynamically readjusted first by removing the corresponding row for the host from the *Plan Matrix* P_M and secondly by pruning all partial plans containing the offline host

[39]Queries $FQ9$, $FQ35$, and $FQ36$ were not considered since they produce no results be default, while query $FQ34$ could not be run in the fully replicated scenario since the physical machine did not have enough resources to accommodate the extra replicated servers in this case.

generated up to the detection moment. In most cases AVALANCHE is not impacted by the fact that a host has failed when at least another alternate plan to produce results exists. Of course, if all query relevant hosts fail, then the query will timeout without any results found. As the results indicate AVALANCHE is able to return the a result set of similar size than the one without disappearing hosts within a similar time-frame as the stable host setting.

4.2. Evaluation Setting II: Analyzing AVALANCHE *with synthetic data*

One of the key characteristics of the WoD is represented by its semantic heterogeneity stemming from a plethora of intertwining applications domains. Currently this aspect alone represents an important part of a federated query's selectivity. However, it is not inconceivable that in the future *schema-homogeneous* partitions of the WoD will increase in size reducing the usefulness of schema/vocabulary information during planing. These kind of *instance-level messy* distributed RDF datasets, hence, significantly complicates distributed query processing as it is unclear if triples matching one triple pattern from one host are likely to join with matches to a second triple pattern from the same host or another. This kind of messiness attenuates the effect of locality.[40] While AVALANCHE was not designed with the intend of addressing *instance-level messiness* we investigate the behavior of our proposed execution paradigm when individual instances (triples) are spread across a large number of *semantically-homogenous* hosts with increasing degrees of messiness.

To this end we employed the synthetic LUBM benchmark dataset [14]. Specifically, we generated the LUBM2000 benchmark configuration, resulting in 2000 universities, and accounting to a total of 276 million triples. In contrast to the previous setup, where 26 **schema-heterogeneous** endpoints were used, a total of 100 **schema-homogeneous** endpoints are created. Such a setup allows us to flexibly mimic instance-level "distribution messiness" by reassigning triples to hosts. Note, that this setup situates AVALANCHE in a ***worst case*** scenario, where the *Source Discovery Phase* reports a large number of semantically-identical sources—all sharing the same schema—but with an unknown distribution of triples.

The Data and its distribution. As illustrated in Figure 17, the LUBM triples were allocated to hosts according to the three *LUBM2000 D1, LUBM2000 D3*, and *LUBM2000 D5* distributions (in short $D1$, $D3$ respectively $D5$). The degree of distribution messiness increases with each case as detailed in the remainder of this section.

A *coarse-grained level of messiness* is achieved in the *LUBM2000 D1* data-distribution. Here all data belonging to a university is placed on the same host. To simulate various levels of server load we assign universities to hosts using the following procedure. Half the universities are randomly assigned to a host ensuring a basic load for each host. The second half of the universities are assigned to a host by drawing the host id from a normal normal distribution with mean $\mu = 50$ and standard deviation $\sigma = 14$. This leads to a higher load for some hosts (towards the middle of Figure 17).

To achieve a higher degree of instance-level messiness *LUBM2000 D3 & LUBM2000 D5* additionally distribute triples of one university across 3 or even 5 hosts. The initial

[40]Note that supporting this messiness is one underlying principles of the Semantic Web, as everyone can annotate any resource with some triple.

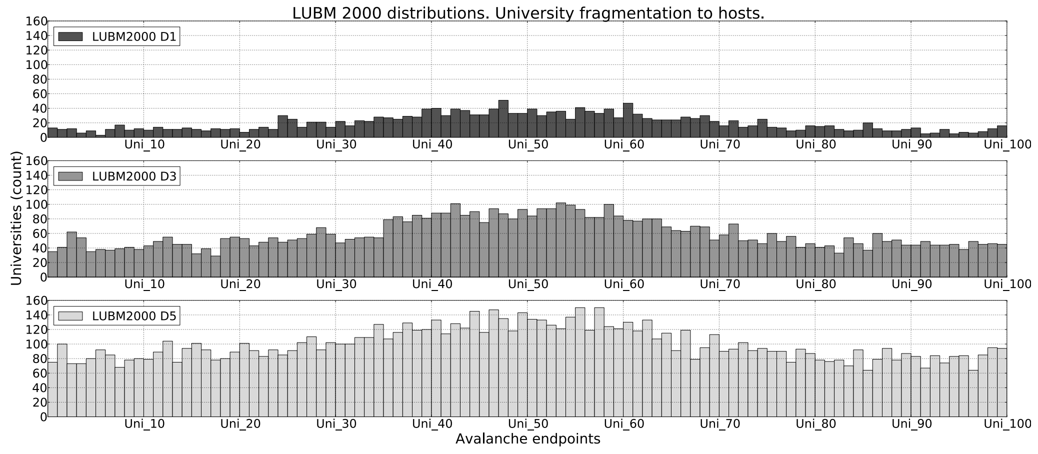


Figure 17: The data distributions chosen over 100 Hosts. The y-axis denotes the number of universities about which a host contains information.

host for a university is still determined using the same procedure as with $D1$. Once that host is determined, however, 2 (or 4) additional hosts are randomly selected. For $D3$ each university's triples are distributed over 3 hosts using a normal distribution with $\mu = 1.5$ and $\sigma = 0.3$. imilarly, for data distribution $D5$, each university's triples are distributed over 5 hosts using a normal distribution with $\mu = 2.5$ and $\sigma = 0.5$. Hence, the bulk of the university's data is still on one host with part data distributed elsewhere. This mimics a Brownian motion of the data away from its originating source – one host contains most of the data while the rest is diffused to other hosts with the chosen probability density function. Consequently, as Figure 17 shows, the hosts will have data about more universities.

The Queries. Although we employed the LUBM benchmark data generator for each of the distributions, we chose not to use the original LUBM benchmark queries since they are *a*) geared towards reasoning systems and *b*) present a coarse grain of complexity in terms of composing triple patterns and number of unbound variables rendering them unsuitable for an in-depth evaluation of AVALANCHE. Instead we devised the 11 SPARQL queries of varying complexity listed in Appendix B (listings 4 through 14) based on the observation that the number of joins involved, their size (number of participating triple patterns), and type are important descriptors of a queries' potential complexity and therefore induced effort. For example star joins can be executed in parallel as n-way joins reducing the complexity of such an operation. However, when joins are chained in a *read-after-write* manner one is forced to process them serially.

Consequently, queries $LQ_i, i \in [0, 10]$ are constructed in order of increased complexity by combining increasingly longer *read-after-write* join chains with increasingly larger sized star patterns.

4.2.1. Experiment #5: Varying Data Distribution

The results of running all eleven queries on the three data distributions ($D1$, $D3$, and $D5$) are graphed in Figures 18 and 19. All runs are warm runs and each query was run 5 times. The following AVALANCHE stopping configuration was used: *1*) the *timeout* was set to 300 seconds (5 minutes), *2*) a *stop sliding window* of size 3 plans, *3*) a *relative saturation* of 0.9, *4*) a number of 512 maximum concurrent asynchronous connections at any given moment, and *5*) a 0.01 bloom-filter false positive error rate.

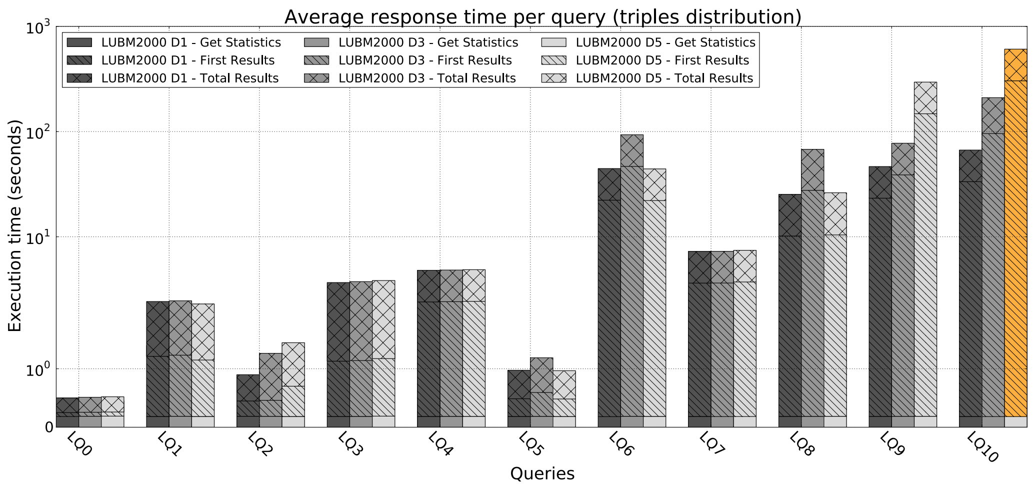


Figure 18: Query execution times for all data distributions. Timeout cases are represented with orange.

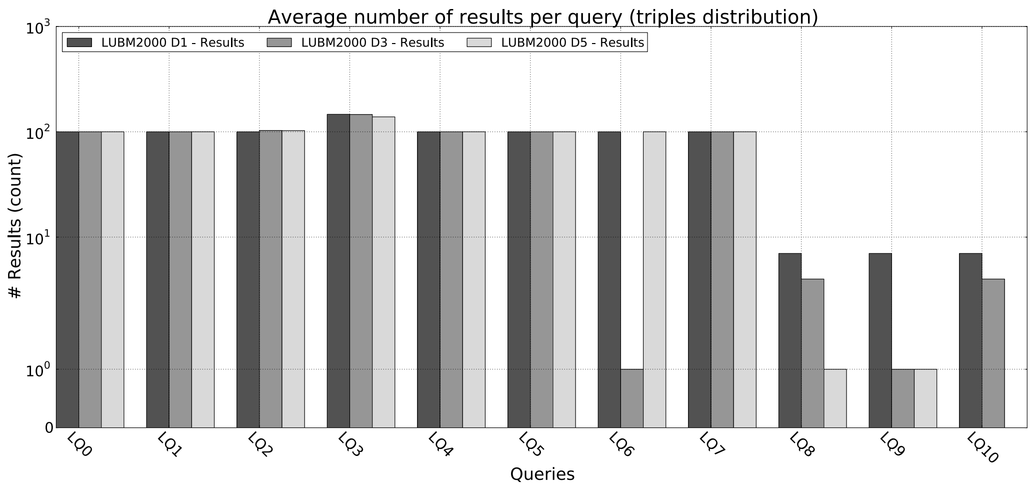


Figure 19: Number of retrieved results (average) for all data distributions.

As can be observed in Figure 18 AVALANCHE exposes a relatively stable performance characteristic without timing-out for queries $LQ0$ through $LQ7$. Instance level spread is actually a benefiting factor for these queries that target replicated knowledge by providing more “chunks” of partial results, which in turn increases AVALANCHE’s chances of generating a “productive” plan. Looking at Figure 19, we can clearly observe that regardless of the degree of messiness (a university’s triples spread to 1, 3 or 5 endpoints), AVALANCHE succeeds in retrieving about the same number of results exhibiting a highly stable behavior. An exception is exhibited by $LQ6$ (Listing 10) where performance degrades only for distribution $D3$. This kind of system behavior is expected in some cases, due mainly to the estimative nature of the cost model. In this particular case the first "productive" plan is discovered relatively late compared to the other 2 distribution cases.

$LQ8$, $LQ9$ and $LQ10$ form a second group of queries. These queries target very specific knowledge pertinent to a single university leaving Avalanche with the task of identifying those endpoints (1, 3 or 5), which produce the desired result when combined. As can be observed, performance degrades dramatically[41] with the number of hosts on which data is spread and with the number of joins generated by the query. This result suggests that naïve selectivity estimation based cost models are not enough when dealing with *fine-grained triple-level messiness* at this scale, warranting novel and (more) accurate estimation statistics. Another effect of increased triples-spread is observed in the decline in recall for this second group of queries (Figure 19). A possible explanation for this observation is that as triples are distributed over more hosts, finding candidate joins becomes harder while the ones that are favored first are usually the more selective and, hence, the ones with fewer results.

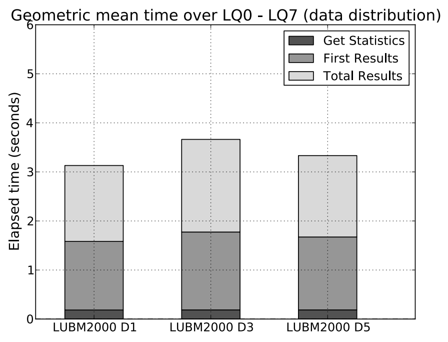


Figure 20: Geometric mean of the execution time over all queries for $D1$,$D3$ and $D5$, queries $LQ0$ through $LQ7$.

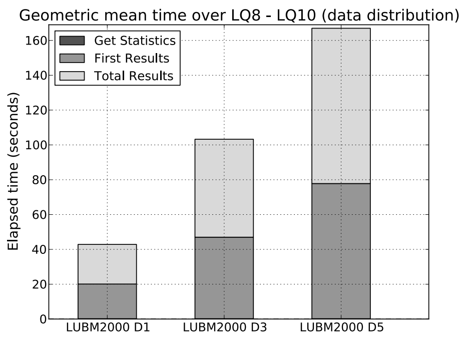


Figure 21: Geometric mean of the execution time over all queries for $D1$,$D3$ and $D5$, queries $LQ8$ through $LQ10$.

The systems' overall behavior for the two query groups is observed more clearly in Figures 20 and 21, where the geometric mean over answering all queries against each distribution is shown. The Figures highlight the elapsed times for the three important execution phases in Avalanche. The *statistics gathering* phase accounts for a negligible part of the entire execution process and accounts to a mere 0.2 seconds on average for both query groups.[42] We observe that Avalanche exposes a stable behavior for the first group of queries finding first answers after an average of 1.5 seconds and completing the query after an average of 1.7 seconds. For the second group of queries, Avalanche exposes a slowdown effect in terms of finding first answers, retrieving them after an average of 48 seconds and completing the query after an average of 56 seconds. Finally, while

[41]Query $FQ10$ times out—depicted in orange—for distribution $D5$ (triples spread over 5 endpoints).

[42]A fact we attribute to the read-optimized Hexastore-inspired indexing model employed by the RDF store used.

Avalanche becomes slower it however, maintains its *robustness* as it will eventually find results.

4.3. Summary

Both evaluation settings in Sections 4.1 and 4.2 are witness to Avalanche's stability against messiness. For the real world data-distribution setup based on Fedbench Avalanche was able to find first results in under one second for about 80% of the considered queries, while total results were retrieved under one second for about 70% of the queries, with the slowest running query taking about 5.5 seconds to complete. A notable exception is represented by query $FQ12$, which generates a large intermediate result set, potentially blocking or slowing down access to underlying shared resources like network connections and database indexes. This is alleviated to some extent by (a) relying on asynchronous socket API's and (b) isolating the execution of expensive queries/joins inside threads or processes. Other possibilities of reducing the overhead of expensive semi-joins is by compressing intermediate result sets. Even more, a good replacement strategy for semi-joins are bloom-joins, where the actual data sent is the bit-vector forming the bloom filter of the intermediate results set. The bloom-join is advantageous for large result sets as $sizeof(ResultSet_{subquery}) >> sizeof(BitVector_{bloomfilter})$.

Furthermore, as shown in the the third experiment when the broker-endpoints network latency changes then Avalanche's slowdown compared to the connection's slowdown exhibits a sub-linear characteristic as graphed in Figure 13. Avalanche is also able to dynamically adapt when some participating endpoints go offline when they are not the sole query results providers. Considering the synthetic LUBM dataset where a "brownian" spread of triples from their source host is simulated, Avalanche exhibits a high level of stability when answering queries that are selective with respect to knowledge that is likely to be replicated (i.e. classes) as seen in Figure 20. Avalanche does become progressively slower for queries that target specific resources (Figure 21). This happens since the objective functions considered do not leverage in any way the data distribution aspect.

5. Limitations, Optimizations, and Future Work

The work presented here exhibits two kinds of limitations. On one side the external validity of the evaluation is limited; on the other side the system could be extended and/or optimized. We will discuss both of these topics in turn.

Evaluation limitations. Our experiments rely on a limited number of physical resources available for accommodating the endpoints, the number of physical machines used is 4 to 16 times smaller than required in reality, where an endpoint would most often reside on an individual server. When one machine accommodates multiple endpoints, then these endpoints compete for shared (such as RAM, disk I/O, network I/O, and CPU-time). We think that the impact on our finding is mitigated by the choice of machines with more cores then endpoints. Furthermore, real-world endpoints would have to answers multiple query requests, each of which also competes for machine resources. Hence, we believe that our setup is as realistic as possible in an experimental laboratory-setup and allows generalizing the results.

Furthermore, we do not support SPARQL filters. However, with the exception of computationally intensive filters (i.e. regular expressions), they are usually pushed towards remote endpoints and can result in the reduction of the partial result-sets that need to be sent between endpoints, at the expense of a somewhat higher CPU workload. Consequently, the implementation of filters is likely to improve the performance of Avalanche by reducing the amount of data sent of over the network indicating that our results can be seen as a worst-case. Given these considerations, we believe that BGP pattern matching is more difficult problem when querying the indexed WoD while SPARQL filters are a secondary, optimizing concern that we intend to address as part of the next iteration of the Avalanche system.

System limitations and optimizations. The Avalanche system has shown how a completely heterogeneous distributed query engine that makes no assumptions about data distribution could be implemented. The current approach does have a number of limitations. In particular, we need to better understand the employed objective functions for the planner, investigate if the requirements put on participating triple-stores are reasonable, explore if Avalanche can be changed to a stateless model, and empirically evaluate if the approach scales to an even larger number of active hosts. Note that for real data distributions the results show it scales well to hundreds of participating hosts. Here we discuss each of these issues in turn.

The *Query Preprocessing* phase in it's current implementation gets executed once each time a query is dispatched. There are cases where depending on the data distribution and query formulation the simple source selection algorithm proposed—a schema (or ontology) index of hosts—can omit relevant sources, potentially leading to a limited result-set or no results at all. Consider for example the case of a query where the first triple pattern has a bound predicate in the FOAF namespace, while the second chained triple pattern contains no bound variables. Whilst the planer will know that it has to look for hosts with FOAF predicates for the first triple pattern it has no plan-time (or prior) information about what hosts are likely to match the second one resulting in an explosion of the plan space. Dynamic source selection based on partial results or the results of prior runs might help to to prune this search space.

One of the core optimizations employed by Avalanche lies in the cost model used. The heuristics considered estimate the joins based on their selectivity. Although the estimations are good enough in most cases more accurate statistics pertinent to join estimations could significantly improve Avalanche's performance as highlighted by the LUBM queries $LQ8$, $LQ9$ and $LQ10$ (see Section 4.2). One approach to gather such estimations would be to record "popular" join cardinalities helping the planer find productive plans faster. Other approaches might consider investigating sampling techniques for estimating distributed joins.

In order to support Avalanche existing triple-stores should be able to:

◦ report statistics: **cardinalities** can be exported as VOID statistical descriptors, bloom filters or other future extensions,

◦ support the execution of **distributed joins**, common in distributed databases as detailed in Section 3.3. This entails the ability to execute sub-queries. The operation could be delegated to an intermediary but would be inefficient. And

◦ share the **same key space**. Whilst the keys could be URI strings this would results in bandwidth-intensive joins and merges as well as CPU intensive string comparisons.

As argued above the first requirement is already supported by most triple-stores via the SPARQL-standard. We still need to investigate how complex the second and third extension are in practice. One possibility would be the an extension of the SPARQL standard with the above-mentioned operations (namely distributed joins), which we will attempt to propose.

The current AVALANCHE process assumes that hosts keep partial results throughout plan execution to reduce the cost of local database operations and that result-views are kept for the duration of a query. This limits the number of queries a host can handle. We intend to investigate if a stateless approach is feasible. Note that the simple approach—the use of REST-ful services [11]—may not be applicable as the size of the state (i.e., the partial results) may be too large and overburden the available bandwidth.

Probably one of the most hard to control, potential limiting aspect of AVALANCHE stems from its aggressive concurrent execution of multiple same-query decompositions. A positive effect is that at least some of the unproductive plans have the effect of "warming up" the underlying endpoints. This is beneficial when multiple different but overlapping queries (where some basic graph pattern is shared) are executed by the system. However, this is also detrimental when multiple disjoint queries are executed resulting in a thinner spread of overall system resources especially when they reside on overlapping hosts – an effect we call "plan flooding". A possible solution would be controlling the amount of load a query is allowed to put on the overall WoD. Finding the correct specification of that "allowable load" is an open problem.

We designed AVALANCHE with the need for handling messy semi-structured data at large scales. The core idea follows the principle of *decentralization.* It also supports *asynchrony* using asynchronous HTTP requests to avoid blocking, *autonomy* by delegating the coordination and execution of the distributed join/update/merge operations to the hosts, *concurrency* through the pipeline shown in Figure 3, *symmetry* by allowing each endpoint to act as the initiating AVALANCHE node for a query caller, as well as *fault tolerance* via proper exception and time-out handling and stopping conditions.

We would like to point out that AVALANCHE completely ignores schema. Whilst this allows us to provide a schema-agnostic solution it does delegate the problem to the querying user. As a large number of publications on schema-integration [10] and the owl:sameAs problem (i.e., [15]) show a lot of work might still be needed to address this kind of messiness transparently. Hence, this is beyond the scope of AVALANCHE.

6. Conclusion

In this paper we presented AVALANCHE, a novel approach for querying the Web of Data that (1) makes no assumptions about data distribution, availability, or partitioning exhibiting skew resistance for classes of queries that are selective with regards to replicated knowledge (i.e. Class information), (2) is dynamically adaptive to changing external network conditions, (3) provides up-to-date results, and (4) is flexible since it makes few limiting assumptions about the structure of participating triple stores. Specifically, we showed that AVALANCHE is able to execute non-trivial queries over distributed data-sources with an ex-ante unknown data-distribution. We showed that an extensible cost model based on a common *Multi Objective Optimization* method—the method of *Global Criterion*, where different heuristics can be plugged in without imposing changes to existing ones—can yield good performance in spite of different data distributions or changing latency while allowing for a messy Web of Data.

By design AVALANCHE handles messiness generated by (i) schema alignment and data evolution, as AVALANCHE is schema agnostic its current view of the world is as a set of triples, (ii) data distribution through its extensible cost model, and (iii) source un-availability, as AVALANCHE dynamically dismisses plans issued to hosts that are not present anymore during the execution phase, still allowing other hosts (sources) to produce new and more results.

AVALANCHE's main limitation with respect to messiness is its assumption that participating data-sources are indexed (i.e., stored in some kind of triple store rather than "just" provided as files). In the light of its robustness against other kinds of messiness, however, we believe that AVALANCHE's capabilities outweigh this disadvantage—in particular since it would be simple to "wrap" any (known)file-based source with a combination of a triple-store and crawler.

To our knowledge, AVALANCHE is the first Semantic Web query system that makes no assumptions about the data distribution whatsoever. Whilst it is only a first implementation with a number of drawbacks it represents an important step towards querying a messy Web of Data by embracing its messiness as necessity (rather than an impediment) in order to foster its unpredictable growth.

Acknowledgements

This work was partially supported by the Swiss National Science Foundation under contract number 200021-118000.

References

[1] Gennady Antoshenkov and Mohamed Ziauddin. Query processing and optimization in Oracle Rdb. *The VLDB Journal The International Journal on Very Large Data Bases*, 5(4):229–237, December 1996.

[2] M Atre, W Chaoji, M J Zaki, and J A Hendler. Matrix "bit" loaded:a scalable lightweight join query processor for rdf data. *International World Wide Web Conference*, 2010.

[3] C Basca and A Bernstein. Avalanche: Putting the spirit of the web back into semantic web querying. *The 6th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2010)*, pages 64–79, Nov 2010.

[4] A Bernstein, C Kiefer, and M Stocker. Optarq a sparql optimization approach based on triple pattern selectivity estimation. Technical report, University of Zürich, 2007.

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

[6] Christian Bizer. D2RQ - treating non-RDF databases as virtual RDF graphs.

[7] J Broekstra, A Kampman, and Frank van Harmelen. Sesame: an architecure for storing and querying rdf data and schema information. *Spinning the Semantic Web*, pages 197–222, 2003.

[8] M Cai and M R Frank. Rdfpeers: a scalable distributed rdf repository based on a structured peer-to-peer network. *13th International World Wide Web Conference (WWW)*, pages 650–657, 2004.

[9] O Erling and I Mikhailov. Rdf support in the virtuoso dbms. Technical report, OpenLink Software, Apr 2009.

[10] J Euzenat and P Shvaiko. *Ontology matching*. Springer-Verlag, Heidelberg (DE), 2007.

[11] R T Fielding and R N Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2:115–150, May 2002.

[12] C Guéret, P Groth, and S Schlobach. erdf: Live discovery for the web of data. *Billion Triple Challenge at ISWC*, 2009.

[13] C Guéret, E Oren, S Schlobach, and M Schut. An evolutionary perspective on approximate rdf query answering. *Proceedings of the 2nd international conference on Scalable Uncertainty Management*, pages 215–228, 2008.

[14] Y Guo, Z Pan, and J Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, 2005.

[15] H Halpin, P Hayes, J P McCusker, D McGuinness, and H S Thompson. When owl:sameas isn't the same: An analysis of identity in linked data. *9th International Semantic Web Conference (ISWC2010)*, Nov 2010.

[16] A Harth, K Hose, M Karnstedt, A Polleres, K U Sattler, and J Umbrich. Data summaries for on-demand queries over linked data. *International World Wide Web Conference*, 2010.

[17] A Harth, J Umbrich, A Hogan, and S Decker. Yars2: a federated repository for querying graph structured data from the web. *6th International Semantic Web Conference (ISWC)*, pages 211–224, 2007.

[18] O Hartig, C Bizer, and J C Freytag. Executing sparql queries over the web of linked data. *8th International Semantic Web Conference ISWC2009*, pages 293–309, 2009.

[19] O Hartig and R Heese. The sparql query graph model for query optimization. *4th European Semantic Web Conference*, Jun 2007.

[20] Dennis Heimbigner and Dennis McLeod. A federated architecture for information management. *ACM Transactions on Information Systems*, 3(3):253–278, July 1985.

[21] Donald E Knuth. *The Art of Computer Programming*, volume 1, page 98. Addison-Wesley, 1997.

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

[23] A Langegger and W Woss. Rdfstats - an extensible rdf statistics generator and library. *DEXA '09 Proceedings of the 2009 20th International Workshop on Database and Expert Systems Application*, 2009.

[24] A Langegger, W Wöß, and M Blöchl. A semantic web middleware for virtual data integration on the web. *Lecture Notes In Computer Science*, 2008.

[25] Y Li and J Heflin. Using reformulation trees to optimize queries over distributed heterogeneous sources. *9th International Semantic Web Conference (ISWC2010)*, Nov 2010.

[26] A Maduko, K Anyanwu, and A P Sheth. Estimating the cardinality of rdf graph patterns. *16th International World Wide Web Conference (WWW)*, May 2007.

[27] K. Miettinen. *Nonlinear Multiobjective Optimization*, volume 12 of *International Series in Operations Research and Management Science*. Kluwer Academic Publishers, Dordrecht, 1999.

[28] H Muehleisen, T Walther, A Augustin, M Harasic, and R Tolksdorf. Configuring a self-organized semantic storage service. *The 6th International Workshop On Scalable Semantic Web Knowledge Base Systems (SSWS2010)*, pages 1–16, Nov 2010.

[29] T Neumann and G Weikum. Scalable join processing on very large rdf graphs. *36th International Conference on Management of Data (SIGMOD)*, Jun 2010.

[30] E Oren, C Gueret, and S Schlobach. Anytime query answering in rdf through evolutionary algorithms. *The Semantic Web - ISWC 2008*, 5318:98–113, 2008.

[31] Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems (2nd Edition)*. Prentice Hall, 2 edition, 1999.

[32] B Quilitz and U Leser. Querying distributed rdf data sources with sparql. *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, pages 524–538, 2008.

[33] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[34] S Ramesh, O Papapetrou, and W Siberski. Optimizing distributed joins with bloom filters. *ICDCIT '08: Proceedings of the 5th International Conference on Distributed Computing and Internet Technology*, pages 145–156, 2009.

[35] S Schenck and S Staab. Networked graphs: a declarative mechanism for sparql rules, sparql views and rdf data integration on the web. *17th International World Wide Web Conference (WWW)*, Apr 2008.

[36] M Schmidt, O Gorlitz, P Haase, A Schwarte, G Ladwig, and T Tran. Fedbench: A benchmark suite for federated semantic data query processing. *International Semantic Web Conference*, 2011.

[37] A P Sheth and J A Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, Sep 1990.

[38] H Stuckenschmidt, R Vdovjak, G J Houben, and J Broekstra. Index structures and algorithms for querying distributed rdf repositories. *13th International World Wide Web Conference (WWW)*,

May 2004.

[39] Giovanni Tummarello, Richard Cyganiak, Michele Catasta, Szymon Danielczyk, Renaud Delbru, and Stefan Decker. Sig.ma: Live views on the Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(4):355–364, November 2010.

[40] Octavian Udrea, College Park, and Andrea Pugliese. GRIN : A Graph Based RDF Index. *Artificial Intelligence*, 22(2):1465–1470, 2007.

[41] Marshall Van Alstyne, Erik Brynjolfsson, and Stuart Madnick. Why not one big database? principles for data ownership. *Decis. Support Syst.*, 15(4):267–284, December 1995.

[42] C Weiss, P Karras, and A Bernstein. Hexastore: Sextuple indexing for semantic web data management. *Proc. of the 34th Intl Conf. on Very Large Data Bases (VLDB)*, 2008.

[43] M. Zeleny. Compromise programming. In J. Cochrane and M. Zeleny, editors, *Multiple Criteria Decision Making*, pages 262–301. University of South Carolina Press, Columbia, 1973.

[44] J Zemánek, S Schenk, and V Svatek. Optimizing sparql queries over disparate rdf data sources through distributed semi-joins. *7th International Semantic Web Conference (ISWC)*, Oct 2007.

Appendix A. Avalanche Endpoint Operators

Execution Operators. For brevity, example query listings will not include the prefixes already defined in the motivating example query Q_{ex}.

`getTPCardinality(`*tp*`)`

As the name suggests, this operator is responsible with returning the number of instances matching the triple pattern *tp* on the callee endpoint. This operator is SPARQL (1.1) compliant and can be implemented in several fashions depending on wether the predicate is bound and VoID is used. To illustrate how, the following triple pattern from Q_{ex} is considered:
< `?chebiDrug, chebi:image, ?chebiImage` >.

Example: getTPCardinality operator to SPARQL(1.1) mapping

```
PREFIX void: <http://rdfs.org/ns/void#>

## If predicate is bound and VoID is used
SELECT ?cardinality WHERE {
      ?dataset         void:propertyPartition   ?partition .
      ?partition       void:property            chebi:image .
      ?partition       void:triples             ?cardinality
}

## If VoID is not used but SPARQL 1.1 compliant
SELECT (COUNT(DISTINCT ?chebiDrug) as ?cardinality)  WHERE {
      ?chebiDrug       chebi:image              ?chebiImage
}
```

`getTotalTriples()`

SPARQL compliant as well, this is arguably the simplest operator. Its task being to report the total number of triples indexed by the endpoint. The overwhelming majority of modern day triple stores are aware of this fact and exposing this as a VoID statistic would be trivial.

Example: getTotalTriples operator to SPARQLmapping

```
## if VoID is used
SELECT ?dataset ?total  WHERE {
      ?dataset         void:triples             ?total  .
}
```

`executeQuery(`*bgp*`, `*vars*`, `*values*`)`

This operator is virtually implemented by all RDF triple stores. The optional *vars* and *values* arguments are mapped directly to the `VALUES` term in SPARQL 1.1. For example consider the second fragment from Q_{ex} in Listing 2 with example dummy *values* for the `?drugBankName` variable:

Example: executeQuery operator to SPARQL1.1 mapping

```
SELECT ?chebiDrug ?chebiImage WHERE {
        ?chebiDrug   chebi:image              ?chebiImage .
        ?chebiDrug   dc:title                 ?drugBankName
} VALUES (?drugBankName) {
      ("Drug A")
      ("Drug B")
      ("Drug C")
}
```

`executeDistributedJoin(`bgp_{local}`, `bgp_{remote}`, `*host*`)`

A critical part of the core functionality of any distributed database querying system is given by the ability to execute distributed joins. This operator is essentially a *proxy* operator as it relies on the ability to execute SPARQL queries both locally and remotely and functions as following: first the subquery bgp_{local} is executed locally as any regular SPARQL query. Next, the join variables (*vars*) between the two subqueries (bgp_{local} and bgp_{remote}) are determined and the partial results corresponding to them are selected (*values*). As the final step the `executeQuery(`bgp_{remote}`, `*vars*`, `*values*`)` operator is called on the remote *host*.

The following operator pair is optional and exists mainly for optimization reasons. Their role is to simply reduce the end I/O cost of executing a distributed query:

`executeDistributedReconciliation`(bgp_{local}, bgp_{remote}, *host*)

Regarded as a "cleanup" operation the set-reconciliation procedure follows the execution of a distributed n-way join in order to remove partial results in excess resulting from preceding joins. Also a *proxy* operator baring a simplistic nature, its task is that of determining the *values* of the join *vars* between the two subqueries (bgp_{local} and bgp_{remote}) and calling `executeReconciliation`(bgp_{remote}, *vars*, *values*) on the remote *host*. Various optimizations are possible at this stage. Hence, instead of sending the actual set of *values* (compressed or not), a set of their hashes or a **bloom filter** can be employed, resulting in a hash- or a bloom filter-optimized distributed join.

`executeReconciliation`(*bgp*, *vars*, *values*)

Always called as the result of executing the `executeDistributedReconciliation` operator, its scope is to select and filter the excess results corresponding to the previously locally executed *bgp* query. As mentioned earlier this operator is designed to reduce the network traffic for the final *merge* phase of the distributed query execution. Depending on the optimization mechanism used (hashing, bloom filters, or the actual set) the process can be exact or exhibit false positives (for bloom filters).

The following operators are required in the final stages of the query execution process:

`executeDistributedMerge`(bgp_{local}, bgp_{remote}, *host*)

Just like the previous `executeDistributedJoin` operator, this is also a proxy operator paired with `executeMerge`. The partial results contained in *results_table* corresponding to the previously executed query bgp_{local} are selected and sent remotely by calling `executeMerge`(bgp_{remote}, *results_table*) on *host*. This operator is outside the scope of SPARQL compliancy, however, it can be implemented as a simple HTTP GET call as described by the REST model [11].

`executeMerge`(*bgp*, *results_table*)

Called as a result of a distributed merge operation, this final operator in the execution pipeline implements the standard database INNER JOIN ($\bowtie$) operation on the incoming remote *results_table* and the local partial results table corresponding to the *bgp* query, which was previously executed during the distributed join phase.

`materialize`(*bgp*)

This operator is necessary when distributed joins are executed in a common ID space used by the remote endpoints to index RDF data-sets. As the name suggests its basic functionality is that of providing the mapping from ID to RDF literals, a necessary condition when formulating the final results.

State Management Operators. The following state management operators[43] are exposed by AVALANCHE as a means to allow query brokers to halt the distributed operations involved in answering a query when the desired results are found:

`stopPlan(pid)`

Although not strictly necessary for AVALANCHE to function, the operator ensures the "cleanup" and freeing of allocated resources while trying to satisfy a given plan denoted by the *pid* identifier (i.e. the MD5 hash of the SPARQL 1.1 query decomposition).

`stopAllPlans`(Q)

Similarly, the operator will stop the execution and free all resources allocated for the resolving of all plans pertaining to the considered query. To reduce network overhead the query string can be replaced with a simple hash of the actual query (i.e., the MD5 hash of the original SPARQL query).

[43]Both operators can be implemented as REST calls

Appendix B. LUBM Benchmark Queries

```
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
PREFIX uni0: <http://www.Department1.University0.edu/>
```

Listing 3: PREFIXES

```
SELECT ?professor WHERE {
    ?professor lubm:name "FullProfessor1"} LIMIT 100
```

Listing 4: LQ0

```
SELECT ?department ?researchGroups WHERE {
    ?researchGroups lubm:subOrganizationOf ?department.
    ?department lubm:name "Department1"} LIMIT 100
```

Listing 5: LQ1

```
SELECT ?studentName WHERE {
    ?student lubm:name ?studentName.
    ?student lubm:memberOf <http://www.Department1.University0.edu>} LIMIT 100
```

Listing 6: LQ2

```
SELECT ?property ?value WHERE {
    ?professor lubm:name "FullProfessor1".
    ?professor ?property ?value} LIMIT 100
```

Listing 7: LQ3

```
SELECT ?mail ?phone WHERE {
    ?professor lubm:emailAddress ?mail.
    ?professor lubm:telephone ?phone.
    ?professor lubm:name "FullProfessor1"} LIMIT 100
```

Listing 8: LQ4

```
SELECT ?mail ?phone ?doctor WHERE {
    ?professor lubm:emailAddress ?mail.
    ?professor lubm:telephone ?phone.
    ?professor lubm:doctoralDegreeFrom ?doctor.
    ?professor lubm:name "FullProfessor1"} LIMIT 100
```

Listing 9: LQ5

```
SELECT ?studentName ?courseName WHERE {
    ?student lubm:takesCourse ?course.
    ?course lubm:name ?courseName.
    ?student lubm:name ?studentName.
    ?student lubm:memberOf <http://www.Department1.University0.edu>} LIMIT 100
```

Listing 10: LQ6

```
SELECT ?publication ?author ?department ?university WHERE {
    ?publication lubm:name "Publication0".
    ?publication lubm:publicationAuthor ?author.
    ?author lubm:worksFor ?department.
    ?department lubm:subOrganizationOf ?university} LIMIT 100
```

Listing 11: LQ7

```
SELECT ?name ?advisor ?department WHERE {
    ?advisor lubm:worksFor ?department.
    ?student lubm:advisor ?advisor.
    ?student lubm:name ?name.
    ?student lubm:takesCourse uni0:GraduateCourse33} LIMIT 100
```

Listing 12: LQ8

```
SELECT ?name ?tel ?advisor ?department WHERE {
    ?advisor lubm:worksFor ?department.
    ?student lubm:advisor ?advisor.
    ?student lubm:name ?name.
    ?student lubm:telephone ?tel.
    ?student lubm:takesCourse uni0:GraduateCourse33} LIMIT 100
```

Listing 13: LQ9

```
SELECT ?university ?student ?name ?tel WHERE {
    ?student lubm:advisor ?advisor.
    ?advisor lubm:worksFor ?department.
    ?department lubm:subOrganizationOf ?university.
    ?student lubm:name ?name.
    ?student lubm:telephone ?tel.
    ?student lubm:takesCourse uni0:GraduateCourse33} LIMIT 100
```

Listing 14: LQ10

Appendix C. Fedbench Query Name Mapping

Table C.4: Fedbench query name mapping

Collection	**Fedbench Name**	**Name**	**Fedbench Name**	**Name**	**Fedbench Name**	**Name**
Cross Domain	CD 1a[c]	**FQ0**	CD 1b[c]	**FQ1**	CD 2	**FQ2**
	CD 3	**FQ3**	CD 4	**FQ4**	CD 5	**FQ5**
	CD 6	**FQ6**	CD 7	**FQ7**		
Life Sciences	LS 1a[c]	**FQ8**	LS 1b[c]	**FQ9**	LS 2a[c]	**FQ10**
	LS 2b[c]	**FQ11**	LS 3	**FQ12**	LS 4	**FQ13**
	LS 5	**FQ14**	LS 6	**FQ15**		
Life Sciences +[b]		**FQ33**		**FQ34**		**FQ35**
		FQ36		**FQ37**		
Linked Data	LD 1	**FQ16**	LD 2	**FQ17**	LD 3	**FQ18**
	LD 4	**FQ19**	LD 5	**FQ20**	LD 6	**FQ21**
	LD 7	**FQ22**	LD 8	**FQ23**	LD 9	**FQ24**
	LD 10	**FQ25**	LD 11	**FQ26**		
SP^2Bench	SP2Bench Q1	**FQ27**	SP2Bench Q2	**FQ28**	SP2Bench Q5	**FQ29**
	SP2Bench Q9a[c]	**FQ30**	SP2Bench Q9b[c]	**FQ31**	SP2Bench Q10	**FQ32**

[a] Original query names from the Fedbench project: http://code.google.com/p/fbench/wiki/Queries.
[b] These queries are not part of the original Fedbench benchmark and therefore do not have a corresponding denomination. They are added for their increased complexity.
[c] Queries whose names are suffixed with *a* or *b* represent Fedbench queries that contain the UNION operator. The two subqueries are executed independently.

Appendix D. Fedbench Benchmark Queries

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX kegg: <http://bio2rdf.org/ns/kegg#>
PREFIX nytimes: <http://data.nytimes.com/elements/>
PREFIX geonames: <http://www.geonames.org/ontology#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX swc: <http://data.semanticweb.org/ns/swc/ontology#>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX bench: <http://localhost/vocabulary/bench/>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX person: <http://localhost/persons/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
PREFIX drugbank-category: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugcategory/>
PREFIX drugbank-drugs: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/>
PREFIX linkedmdb: <http://data.linkedmdb.org/resource/movie/>
PREFIX chebi: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX purl: <http://purl.org/dc/terms/>
```

Listing 15: PREFIXES

```
SELECT ?predicate ?object WHERE {
    dbpedia:Barack_Obama ?predicate ?object}
```

Listing 16: FQ0

```
SELECT ?predicate ?object WHERE {
    ?subject owl:sameAs dbpedia:Barack_Obama.
    ?subject ?predicate ?object}
```

Listing 17: FQ1

```
SELECT ?party ?page WHERE {
    dbpedia:Barack_Obama dbpedia-owl:party ?party.
    ?x nytimes:topicPage ?page.
    ?x owl:sameAs dbpedia:Barack_Obama}
```

Listing 18: FQ2

```
SELECT ?president ?party ?x WHERE {
    ?president rdf:type dbpedia-owl:President.
    ?president dbpedia-owl:nationality dbpedia:United_States.
    ?president dbpedia-owl:party ?party.
    ?x nytimes:topicPage ?page.
    ?x owl:sameAs ?president}
```

Listing 19: FQ3

```
SELECT ?actor ?news WHERE {
    ?film purl:title "Tarzan".
    ?film linkedmdb:actor ?actor.
    ?actor owl:sameAs ?x.
    ?y owl:sameAs ?x.
    ?y nytimes:topicPage ?news}
```

Listing 20: FQ4

```
SELECT ?film ?director ?genre WHERE {
    ?film dbpedia-owl:director ?director.
    ?director dbpedia-owl:nationality dbpedia:Italy.
    ?x owl:sameAs ?film.
    ?x linkedmdb:genre ?genre}
```

Listing 21: FQ5

```
SELECT ?name ?location WHERE {
    ?artist foaf:name ?name.
    ?artist foaf:based_near ?location.
    ?location geonames:parentFeature ?germany.
    ?germany geonames:name "Federal Republic of Germany"}
```

Listing 22: FQ6

```
SELECT ?location ?news WHERE {
    ?location geonames:parentFeature ?parent.
    ?parent geonames:name "California".
    ?y owl:sameAs ?location.
    ?y nytimes:topicPage ?news}
```

Listing 23: FQ7

```
SELECT ?drug ?melt WHERE {
    ?drug drugbank:meltingPoint ?melt}
```

Listing 24: FQ8

```
SELECT ?drug ?melt WHERE {
    ?drug dbpedia-owl:drug/meltingPoint ?melt}
```

Listing 25: FQ9

```
SELECT ?predicate ?object WHERE {
    drugbank-drugs:DB00201 ?predicate ?object}
```

Listing 26: FQ10

```
SELECT ?predicate ?object WHERE {
    drugbank-drugs:DB00201 owl:sameAs ?caff.
    ?caff ?predicate ?object}
```

Listing 27: FQ11

```
SELECT ?Drug ?IntDrug ?IntEffect WHERE {
    ?Drug rdf:type dbpedia-owl:Drug.
    ?y owl:sameAs ?Drug.
    ?Int drugbank:interactionDrug1 ?y.
    ?Int drugbank:interactionDrug2 ?IntDrug.
    ?Int drugbank:text ?IntEffect}
```

Listing 28: FQ12

```
SELECT ?drugDesc ?cpd ?equation WHERE {
    ?drug drugbank:drugCategory drugbank-category:cathartics.
    ?drug drugbank:keggCompoundId ?cpd.
    ?drug drugbank:description ?drugDesc.
    ?enzyme kegg:xSubstrate ?cpd.
    ?enzyme rdf:type kegg:Enzyme.
    ?reaction kegg:xEnzyme ?enzyme.
    ?reaction kegg:equation ?equation}
```

Listing 29: FQ13

```
SELECT ?drug ?keggUrl ?chebiImage WHERE {
    ?drug rdf:type drugbank:drugs.
    ?drug drugbank:keggCompoundId ?keggDrug.
    ?keggDrug chebi:url ?keggUrl.
    ?drug drugbank:genericName ?drugBankName.
    ?chebiDrug dc:title ?drugBankName.
    ?chebiDrug chebi:image ?chebiImage}
```

Listing 30: FQ14

```
SELECT ?drug ?title WHERE {
    ?drug drugbank:drugCategory drugbank-category:micronutrient.
    ?drug drugbank:casRegistryNumber ?id.
    ?keggDrug rdf:type kegg:Drug.
    ?keggDrug chebi:xRef ?id.
    ?keggDrug dc:title ?title}
```

Listing 31: FQ15

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

Listing 32: FQ16

```
SELECT ?proceedings ?paper ?p WHERE {
    ?proceedings swc:relatedToEvent <http://data.semanticweb.org/conference/eswc/2010>.
    ?paper swc:isPartOf ?proceedings.
    ?paper swrc:author ?p}
```

Listing 33: FQ17

```
SELECT ?paper ?p ?x ?n WHERE {
    ?paper swc:isPartOf <http://data.semanticweb.org/conference/iswc/2008/
        poster_demo_proceedings>.
    ?paper swrc:author ?p.
    ?p owl:sameAs ?x.
    ?p rdfs:label ?n}
```

Listing 34: FQ18

```
SELECT ?role ?p ?paper ?proceedings WHERE {
    ?role swc:isRoleAt <http://data.semanticweb.org/conference/eswc/2010>.
    ?role swc:heldBy ?p.
    ?paper swrc:author ?p.
    ?paper swc:isPartOf ?proceedings.
    ?proceedings swc:relatedToEvent <http://data.semanticweb.org/conference/eswc/2010>}
```

Listing 35: FQ19

```
SELECT ?a ?n WHERE {
    ?a dbpedia-owl:artist dbpedia:Michael_Jackson.
    ?a rdf:type dbpedia-owl:Album.
    ?a foaf:name ?n}
```

Listing 36: FQ20

```
SELECT ?director ?film ?x ?y ?n WHERE {
    ?director dbpedia-owl:nationality dbpedia:Italy.
    ?film dbpedia-owl:director ?director.
    ?x owl:sameAs ?film.
    ?x foaf:based_near ?y.
    ?y geonames:officialName ?n}
```

Listing 37: FQ21

```
SELECT ?x ?n WHERE {
    ?x geonames:parentFeature <http://sws.geonames.org/2921044/>.
    ?x geonames:name ?n}
```

Listing 38: FQ22

```
SELECT ?drug ?id ?s ?o ?sub WHERE {
    ?drug drugbank:drugCategory drugbank-category:micronutrient.
    ?drug drugbank:casRegistryNumber ?id.
    ?drug owl:sameAs ?s.
    ?s foaf:name ?o.
    ?s skos:subject ?sub}
```

Listing 39: FQ23

```
SELECT ?x ?p WHERE {
    ?x skos:subject dbpedia:Category:FIFA_World_Cup-winning_countries.
    ?p dbpedia-owl:managerClub ?x.
    ?p foaf:name "Luiz Felipe Scolari"}
```

Listing 40: FQ24

```
SELECT ?n ?p2 ?u WHERE {
    ?n skos:subject dbpedia:Category:Chancellors_of_Germany.
    ?n owl:sameAs ?p2.
    ?p2 nytimes:latest_use ?u}
```

Listing 41: FQ25

```
SELECT ?x ?y ?d ?p ?l WHERE {
    ?x dbpedia-owl:team dbpedia:Eintracht_Frankfurt.
    ?x rdfs:label ?y.
    ?x dbpedia-owl:birthDate ?d.
    ?x dbpedia-owl:birthPlace ?p.
    ?p rdfs:label ?l}
```

Listing 42: FQ26

```
SELECT ?yr WHERE {
    ?journal rdf:type bench:Journal.
    ?journal dc:title "Journal 1 (1940)".
    ?journal purl:issued ?yr}
```

Listing 43: FQ27

```
SELECT ?inproc ?author ?booktitle ?title ?proc ?ee ?page ?url ?yr WHERE {
    ?inproc rdf:type bench:Inproceedings.
    ?inproc dc:creator ?author.
    ?inproc bench:booktitle ?booktitle.
    ?inproc dc:title ?title.
    ?inproc purl:partOf ?proc.
    ?inproc rdfs:seeAlso ?ee.
    ?inproc swrc:pages ?page.
    ?inproc foaf:homepage ?url.
    ?inproc purl:issued ?yr}
```

Listing 44: FQ28

```
SELECT ?person ?name WHERE {
    ?article rdf:type bench:Article.
    ?article dc:creator ?person.
    ?inproc rdf:type bench:Inproceedings.
    ?inproc dc:creator ?person.
    ?person foaf:name ?name}
```

Listing 45: FQ29

```
SELECT ?predicate WHERE {
    ?person rdf:type foaf:Person.
    ?subject ?predicate ?person}
```

Listing 46: FQ30

```
SELECT ?predicate WHERE {
    ?person rdf:type foaf:Person.
    ?person ?predicate ?object}
```

Listing 47: FQ31

```
SELECT ?subject ?predicate WHERE {
    ?subject ?predicate person:Paul_Erdoes}
```

Listing 48: FQ32

```
SELECT ?drug ?enzyme ?reaction WHERE {
    ?drug1 drugbank:drugCategory drugbank-category:antibiotics.
    ?drug2 drugbank:drugCategory drugbank-category:antiviralAgents.
    ?drug3 drugbank:drugCategory drugbank-category:antihypertensiveAgents.
    ?I1 drugbank:interactionDrug2 ?drug1.
    ?I1 drugbank:interactionDrug1 ?drug.
    ?I2 drugbank:interactionDrug2 ?drug2.
    ?I2 drugbank:interactionDrug1 ?drug.
    ?I3 drugbank:interactionDrug2 ?drug3.
    ?I3 drugbank:interactionDrug1 ?drug.
    ?drug owl:sameAs ?drug5.
    ?drug5 rdf:type dbpedia-owl:Drug.
    ?drug drugbank:keggCompoundId ?cpd.
    ?enzyme kegg:xSubstrate ?cpd.
    ?enzyme rdf:type kegg:Enzyme.
    ?reaction kegg:xEnzyme ?enzyme.
    ?reaction kegg:equation ?equation}
```

Listing 49: FQ33

```
SELECT ?drug ?drug1 ?drug2 ?drug3 ?drug4 WHERE {
    ?drug1 drugbank:drugCategory drugbank-category:antibiotics.
    ?drug2 drugbank:drugCategory drugbank-category:antiviralAgents.
    ?drug3 drugbank:drugCategory drugbank-category:antihypertensiveAgents.
    ?drug4 drugbank:drugCategory drugbank-category:anti-bacterialAgents.
    ?I1 drugbank:interactionDrug2 ?drug1.
    ?I1 drugbank:interactionDrug1 ?drug.
    ?I2 drugbank:interactionDrug2 ?drug2.
    ?I2 drugbank:interactionDrug1 ?drug.
    ?I3 drugbank:interactionDrug2 ?drug3.
    ?I3 drugbank:interactionDrug1 ?drug.
    ?I4 drugbank:interactionDrug2 ?drug4.
    ?I4 drugbank:interactionDrug1 ?drug}
```

Listing 50: FQ34

```
SELECT ?drug WHERE {
    ?drug1 drugbank:possibleDiseaseTarget <http://www4.wiwiss.fu-berlin.de/diseasome/resource/
         diseases/302>.
    ?drug2 drugbank:possibleDiseaseTarget <http://www4.wiwiss.fu-berlin.de/diseasome/resource/
         diseases/53>.
    ?drug3 drugbank:possibleDiseaseTarget <http://www4.wiwiss.fu-berlin.de/diseasome/resource/
         diseases/59>.
    ?drug4 drugbank:possibleDiseaseTarget <http://www4.wiwiss.fu-berlin.de/diseasome/resource/
         diseases/105>.
    ?I1 drugbank:interactionDrug2 ?drug1.
    ?I1 drugbank:interactionDrug1 ?drug.
    ?I2 drugbank:interactionDrug2 ?drug2.
    ?I2 drugbank:interactionDrug1 ?drug.
    ?I3 drugbank:interactionDrug2 ?drug3.
    ?I3 drugbank:interactionDrug1 ?drug.
    ?I4 drugbank:interactionDrug2 ?drug4.
    ?I4 drugbank:interactionDrug1 ?drug.
    ?drug drugbank:casRegistryNumber ?id.
    ?keggDrug rdf:type kegg:Drug.
    ?keggDrug chebi:xRef ?id.
    ?keggDrug dc:title ?title}
```

Listing 51: FQ35

```
SELECT ?d ?drug5 ?cpd ?enzyme ?equation WHERE {
    ?drug1 drugbank:possibleDiseaseTarget <http://www4.wiwiss.fu-berlin.de/diseasome/resource/
         diseases/261>.
    ?I1 drugbank:interactionDrug2 ?drug1.
    ?I1 drugbank:interactionDrug1 ?drug.
    ?drug drugbank:possibleDiseaseTarget ?d.
    ?drug owl:sameAs ?drug5.
    ?drug5 rdf:type dbpedia-owl:Drug.
    ?drug drugbank:keggCompoundId ?cpd.
    ?enzyme kegg:xSubstrate ?cpd.
    ?enzyme rdf:type kegg:Enzyme.
    ?reaction kegg:xEnzyme ?enzyme.
    ?reaction kegg:equation ?equation}
```

Listing 52: FQ36

```
SELECT ?drug5 ?drug6 WHERE {
    ?drug1 drugbank:possibleDiseaseTarget <http://www4.wiwiss.fu-berlin.de/diseasome/resource/
         diseases/319>.
    ?drug1 drugbank:possibleDiseaseTarget <http://www4.wiwiss.fu-berlin.de/diseasome/resource/
         diseases/270>.
    ?I1 drugbank:interactionDrug1 ?drug1.
    ?I1 drugbank:interactionDrug2 ?drug.
    ?drug1 owl:sameAs ?drug5.
    ?drug owl:sameAs ?drug6}
```

Listing 53: FQ37

C. Basca et al.: Querying a Messy Web of Data with Avalanche
Technical Report No. IFI-2013.03, June 2013
Dynamic and Distributed Information Systems (DDIS)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.ifi.uzh.ch/