Avalanche: Putting the Spirit of the Web back into Semantic Web Querying

Cosmin Basca and Abraham Bernstein

DDIS, Department of Informatics, University of Zurich, Zurich, Switzerland
{lastname}@ifi.uzh.ch

Abstract. Traditionally Semantic Web applications either included a web crawler or relied on external services to gain access to the Web of Data. Recent efforts have enabled applications to query the entire Semantic Web for up-to-date results. Such approaches are based on either centralized indexing of semantically annotated metadata or link traversal and URI dereferencing as in the case of Linked Open Data. By making limiting assumptions about the information space, they violate the openness principle of the Web – a key factor for its ongoing success. 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 distribution of the data – thus adhering to the openness criteria. Specifically, Avalanche can perform "live" (SPARQL) queries over the Web of Data. First, it gets on-line statistical information about the data distribution, as well as bandwidth availability. Then, it plans and executes the query in a 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. Furthermore, we propose to extend the query planning algorithm with qualitative statistical information. We empirically evaluate Avalanche using a realistic dataset, show its strengths but also point out the challenges that still exist.

1 Introduction

With the introduction of the World Wide Web, the way we share knowledge and conduct day to day activities has changed fundamentally. 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 (SW) data crawler, others rely on services that facilitate access to the Web of Data (WoD) either through the SPARQL protocol or various APIs (i.e. Sindice or Swoogle). As the mass of data continues to grow – currently Linked Open Data (LOD) [1] accounts for 4.7 billion triples – 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 query the Web of Data on-demand, without hindering the flexible openness principle of the Web – seen as the ability to query independent un-cooperative semantic databases, not controlling their distribution, their availability or having to adhere to fixed publishing guidelines (i.e. LOD). The underlying assumptions of WoD, as with the WWW, are that (a) there exists no distribution pattern of information onto servers, (b) there is no guarantee of a working network, (c) there is no centralized resource discovery system, (d) there exists a standard (HTTP) for the retrieval of information, and (e) the size of RDF data no longer allows us to consider single-machine systems feasible. With the serendipitous nature of Semantic Web [12], querying the global information space gives rise to new possibilities unthought of before.

Several approaches that tackle the problem of querying the entire Web of Data have emerged lately. One solution 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.

Other approaches use the guidelines of LOD publishing to traverse the linked data 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 bounded entities in the query. Relying on URI structure, however, may cause significant scalability issues when retrieving distributed data sets, since (1) the servers dereferenced in the URI may become overloaded, and (2) it limits the possibilities of rearranging (or moving) the data around by binding the id (i.e. URI) to storage location.

Finally, traditional database federation techniques have been applied to query WoD. They rely on statistical information from queryable endpoints that are used by a mediator to build efficient query execution plans. Their main drawback is that some query execution engine is aware of the data distribution ex-ante (i.e., before the query execution). Moreover, in most cases, data sources even need to register themselves at the query execution engine with detailed information about the data they contain.

In this paper, we propose Avalanche, a novel approach for querying the Web of Data that (1) *makes no assumptions about data distribution, availability, or partitioning*, (2) provides *up-to-date results*, and (3) is *flexible as it makes no assumption about the structure of participating triple stores*. Consequently, it addresses the shortcomings of previous approaches. To query WoD Avalanche provides a novel technique via means of a two-phase protocol: a discovery step, i.e. gathering statistical information about data distribution from involved hosts, and a planning optimization step over the distributed SPARQL endpoints. Hence, the main contributions of our approach are:

- on-demand transparent querying over the Web of Data, without any prior knowledge about its distribution
- a formal description of our approach, together with possible optimizations for each step

– a novel planning strategy and cost model for dealing with towards Web scale graph data
– a reference implementation of the AVALANCHE technique

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: (a) distributed query processing and (b) statistical information gathering over RDF sources.

Research on distributed query processing has a long history in the database field [18, 9]. Its traditional concepts are adapted in current approaches to provide integrated access to RDF sources distributed over the Web. For instance, Yars2 [6] is an end-to-end semantic search engine that uses a graph model to interactively answer queries over structured and interlinked data, collected from disparate Web sources. Another example is the DARQ engine [15], which divides a SPARQL query into several subqueries, forwards them to multiple, distributed query services and, finally, integrates the results of the subqueries. Finally, Rdfpeers [3] 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. Virtuoso [4], 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 [11] provides a mediator that distributes the execution of SPARQL queries transparently. Its main focus is to provide an integration and sharing system for scientific data. Whilst it does assume control over the instance distribution they assume perfect knowledge about it. Addressing this drawback some [20, 17] propose to extend SPARQL with explicit instructions 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. [7] describe an approach for executing SPARQL queries over spaces structured according to the Web of Linked Data rules [1]. Whilst they make no assumptions about the openness of the data space the LOD rules requires them to place the data on the URI-referenced servers—a limiting assumption for example when caching/copying data.

Research on query optimization for SPARQL includes query rewriting [8], triple pattern optimization based on selectivity estimations [13, 19, 14], and on other statistical information gathering over RDF sources [10, 5]. RDFStats [10] is an extensible RDF statistics generator that records how often RDF properties

are used and feeds automatically generated histograms to SemWIQ. Histograms on the combined values of SPO triples have proved to be especially useful to provide selectivity estimations for filters [19]. For joins, however, histograms can grow very large and are rarely used in practice. Another approach is to compute ahead 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 [13]. It is unclear how this would work in a highly distributed scenario. Finally, Neumann et. al [14] claim that selectivity estimation is a worthwhile solution for tens of millions of RDF triples, but unsuitable for billions of triples, because the size of the data and the increasing diversity in property names lead to poor estimations, thus misguiding the query optimizer.

3 Avalanche — System Design and Implementation

In this section, we describe the overall design of AVALANCHE and the underlying philosophy of the distributed query execution across large datasets spread over multiple, uncooperative servers.
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.* Hence, our approach follows neither a federated nor a peer-to-peer model, instead the statistical discovery phase that traditionally is reserved for the (parallel) mediator component in clustered approaches, has become an individual step during each query execution phase. In the remaining part of this section, we will first illustrate our approach using a motivating example. This will lead the way towards thoroughly describing the AVALANCHE components and its novelty.

The system consists of six major components working together in a parallelized pipeline: the AVALANCHE *endpoints Web Directory* or *Search Engine*, the *Statistics Requester*, the *Plan Generator*, *Plan Executor* instances, *Plan Materializer* instances and the *Query Stopper* component as seen in Figure 1.

AVALANCHE comprises of two phases: the *Query Preprocessing* phase and the parallel *Query Execution* phase. During *Query Preprocessing*, participating hosts are identified via means of a `Search Engine` such as Sindice[1] or `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, classes, labels, etc) are appropriate candidate entries to index. After query parsing, this information is immediately available and used to quickly trim down the number of potential endpoints. Then, all selected AVALANCHE endpoints are queried for the cardinality (number of instances) of each unbounded variable — statistical information that triple-stores generally posses.

In the *Query Execution* phase, first the query is broken down into the superset of all **molecules**, where a molecule is a subgraph of the overall query graph.

[1] http://sindice.com/

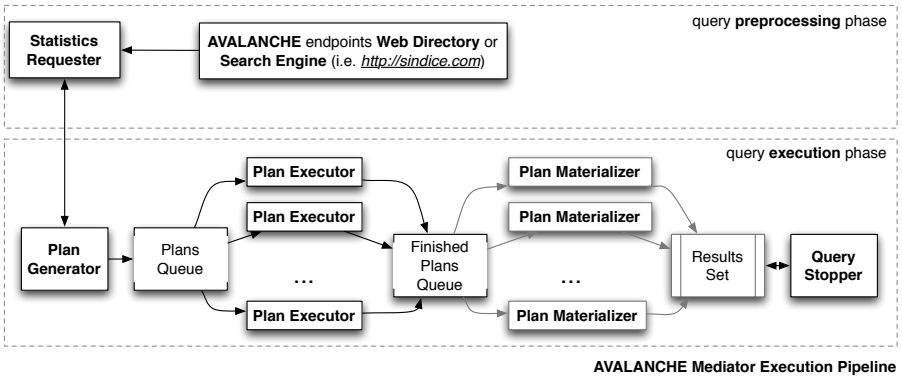

Fig. 1. The Avalanche execution pipeline

A combination of minimally overlapping molecules covering the directed query graph is referred to as a **solution**. Binding all molecules in a given solution to physical hosts (Avalanche endpoints) that may resolve them, transforms a solution into a **plan**. Given the size of the Web and the unknown distribution of the RDF data, Avalanche will try to optimize the execution of the query to quickly find the **first K** results. The proposed planning system and algorithm, though complete, will normally not be allowed to exhaust the entire search space since the cost of doing so is prohibitively expensive. Instead, the planner component strives to execute the most "promising" query plans first, while being monitored by the `Query Stopper` for termination conditions. To further reduce the size of the search space, a windowed version of the search algorithm can be employed – i.e. with each exploratory step only the first M molecules are considered, thus sacrificing completeness.

As shown in Figure 1 the `Plan Generator` relies on statistics about the data contained on the different hosts from the `Statistics Requester`. Any generated plan gets put in the `Plans Queue` regardless if the planner finished its overall tasks of exploring the plan space or not. Plans in the `Plans Queue` are fetched by `Plan Executors` that execute them generating partial results in parallel and put them in the `Finished Plans Queue`. There, they get fetched by one of the parallel executing `Plan Materializers`, who will merge and materialize the partial results.

To put Avalanche into perspective consider the following **motivating query** that executes over Linked Open Datasets describing movies and actors:

```
SELECT ?title ?photoCollection ?name WHERE {
?film dc:title ?title; movie:actor ?actor;  owl:sameAs ?sameFilm.
?actor a foaf:Person; movie:actor_name ?name .
?sameFilm dbpedia:hasPhotoCollection ?photoCollection.
?sameFilm dbpedia:studio ‘‘Producers Circle’’; }
```

The goal of AVALANCHE is to return the list of all movie titles, their photo collections and the names of starring actors, that have been produced at "Producers Circle" studios – considering that the required information is spread with an unknown distribution over several LOD endpoints.

At a given moment during the execution of a plan, a `Plan Executor` instance may find itself in the state depicted in Figure 2 (in depth description in Section 3.2). The plan is comprised of three molecules: `M1, M2, M3` and three hosts are involved: `host A, host B` and `host C`. Molecule `M1` was reported to be highly selective on host A (holding Linked Movie[2] data), while the remainder of the plan: molecule `M2` and `M3`, is distributed between hosts B and C (both holding DBPEDIA[3] data). Given that we operate in an environment where bandwidth cost is non-trivial we should not "just" transport all partial results to one central server to be joined. Instead we start with executing the highly selective (or in this case: with the lowest cardinality) molecule `M1` on host A and then limit the execution space on host B by sending over the results from host A. The process repeats itself given the number of molecules in the plan and is finalized with a merge/update operation in reverse join order.

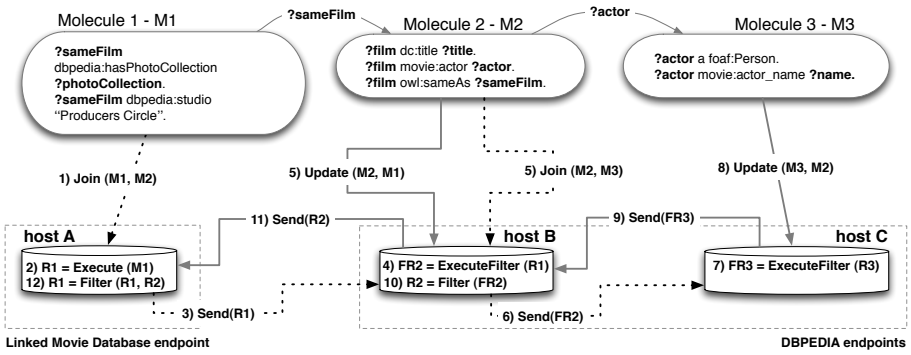

Fig. 2. Distributed Join and Update operations for a Simple Plan

It is important to note that to execute plans, hosts will need to share a common id space – a given in Semantic Web via URIs. Naturally, using RDF strings can be prohibitively expensive. To limit bandwidth requirements, we chose to employ a single global id space in the form of the `SHA` family of hash functions on the URIs.

The remainder of this section will detail the functionality of the most important elements: the `Plan Generator`, `Plan Executor` and `Plan Materializer` as well as explain how the overall pipeline stops.

[2] http://www.linkedmdb.org/
[3] http://dbpedia.org/About

3.1 Generating Query Plans

The planner's main focus is to generate query plans that are likely to produce results fast with a minimum of cost. As shown in Algorithm 1 the planner will try to optimize the construction of plans using a multi-path informed (best-first) search strategy by maximizing the OBJECTIVE function of a plan. Therefore, all plans are generated in descending order of their objective function.

Algorithm 1 The plan generator algorithm

```
PLAN-GENERATOR(Molecules, Hosts, Cardinalities)
 1  fringe = []
 2  for each molecule M ∈ Molecules, host H ∈ Hosts
 3       partialPlan = {M, H, NULL, Cardinalities}
 4       APPEND(fringe, partialPlan)
 5  SORT(fringe)
 6  while !fringe.empty() // Loop through fringe
 7       best = GETFIRSTELEMENTWITHPOSITIVEOBJECTIVE(fringe)
 8       if PLANISCOMPLETE(best) // all molecules assigned to host
 9            SORT(fringe)
10            yield GETPLAN(best) // returns results but continues planning
11       else // plan is incomplete
12            remMol = GETREMAININGCONNECTEDMOLECULES(Molecules, best)
13            planFringe = []
14            for each molecule M ∈ remMol, host H ∈ Hosts
15                 partialPlan = {M, H, best, Cardinalities}
16                 APPEND(planFringe, partialPlan)
17            SORT(planFringe)
18            CONCATENATE(fringe, planFringe)
```

In defining the OBJECTIVE function we use the statistical information gathered beforehand (result set cardinality). To ensure the generation of most productive plans, our function models the chance of finding a solution, utility U, divided by the cost of executing the query, C. Hence:

$$Objective = \frac{U}{C} \tag{1}$$

An emergent challenge from preserving the openness of the query process and the flexibility of semantic data publishing, is denoted by the exponential complexity class of the plan composition space. Thus the space complexity of the problem is $O(N^3)$, considering that the problem size increases by $M * H$ with each step towards a complete plan, where H represents the total number of hosts involved and M is a measure of the query complexity (i.e. the number of unique *molecules* that can be extracted from the given query graph). A simple calculation for the scenario where 1000 hosts are involved and a rather large

query ($\approx$ 15 unbounded variables) might generate 500 molecules with the average depth of a plan of 10 (molecules), results in 5 million possible combinations to form *plans*. Not all combinations produce viable plans, so pruning low or no utility plans early is desired as seen in line 7 of the planning algorithm.

We follow the assumption that selective molecules – with low cardinalities – will help the plan to converge faster. In the bootstrap phase the utility of the first plan node is equal to the inverse of its cardinality: CNT_{N1} (where $N1$ is node 1 and CNT is the cardinality) factored by the size of the plan ($Edges(N1)$). Further on, we consider a join where the best-case cardinality is the minimum of the involved result set cardinalities (see Equation 2). We define the cost, C for executing queries in Equation 3. The cost of the first node is assumed to be constant. For all other nodes we combine:

- the network latency L (between two nodes)
- a measure of the time required to send the results from node $N1$ to node $N2$ given the bandwidth B
- the cost of executing on $N1$ and $N2$ as approximated by their cardinalities

Finally, we scale this result with a measure of the current molecule size (molecule assigned to $N2$) relative to the size of the whole solution, in order to encourage the choice of nodes that aid convergence.

$$U_{N1,N2} = \begin{cases} \frac{Edges(N1)}{CNT_{N1}}, & \text{first node} \\ min(CNT_{N1}, CNT_{N2}), & \text{otherwise} \end{cases} \tag{2}$$

$$C_{N1,N2} = \begin{cases} 1, & \text{first node} \\ (L + \frac{CNT_{N1}}{B} + CNT_{N1} + \frac{CNT_{N2}}{CNT_{N1}}) \frac{Edges(Solution)}{Edges(N2)}, & \text{otherwise} \end{cases} \tag{3}$$

Extended Utility Function The main drawback of this utility function is that it assumes the lower cardinality of the two nodes is representative—an assumption that is quite wrong when searching for "rare" results given a large number of "promising" hosts. Therefore it disregards the actual join probabilities. Consider the previous example query that goes out to two almost disjoint RDF servers: one with DBPEDIA data and another with public social network data. Assuming we found an actor through some other host, the utility function will not be able to favor DBPEDIA over the other host, as it cannot evaluate the actual number of joins. Hence, if the public social network host happens to be using a better network connection, the planer will be lead astray. To overcome this effect we need a measure of join-quality. Following [16] we employ *bloom-filters*, which are space-efficient set representation bit vectors composed of multiple hash functions. As stated by [2] bloom-filters allow for a statistically solid estimation of the cardinality of the join between two sets:

$$JOIN_{BF_1,BF_2} \approx -\frac{1}{k} \frac{\ln(m \frac{Z_1 + Z_2 - Z_{12}}{Z_1 Z_2})}{\ln(1 - \frac{1}{m})} \tag{4}$$

where BF is a bloom filter, m is the number of bits in the bloom filter, k represents the number of hash functions, Z_i represents the number of zero bits in BF_i, and Z_{12} represents the number of zero bits in the magnitude of their inner product.

Since computing bloom filters for large sets is a costly operation, we propose the use of bloom filters as an extension to the previously proposed *utility* function only for highly selectivity molecules — where the cardinality is below a manually set threshold. Given implementation specific, execution considerations we empirically set the threshold to 1000 partial results (ids) for the given set. Consequently the *extended utility EU* is now defined as follows:

$$EU_{N1,N2} = \begin{cases} w1 \cdot JOIN_{BF(N1),BF(N2)} + w2 \cdot U_{N1,N2}, & N1,\ N2 \text{ selective} \\ w2 \cdot U_{N1,N2}, & \text{otherwise} \end{cases} \quad (5)$$

where $w1$ and $w2$ are weights that define the importance of the employed estimation methods. We chose $w1 = 0.8$ and $w2 = 0.2$ for our experiments, which means that for selective molecules we favor a more expensive, but more realistic estimation.

Algorithm 2 The plan execution algorithm

```
EXECUTE-PLAN(Plan)
 1  nodes = SortedList() // initialize
 2  update = Queue()
 3  for each node N ∈ Plan
 4      PUSH(nodes, N) // Note: sorting according to selectivity gets preserved
 5  while !nodes.empty() // While joins to perform, do so
 6      best = POP(nodes)
 7      if nodes.empty()
 8          break
 9      for N ∈ nodes, where GETMOLECULE(best) ∩ GETMOLECULE(N) ≠ ∅
10          joinVariables = GETMOLECULE(best) ∩ GETMOLECULE(N)
11          selectivity = DOJOIN(join[0], best, N)
12          APPEND(update, [join[0], N, best])
13          if selectivity == 0
14              exit and stop
15          else
16              N.selectivity = selectivity
17              UPDATE(nodes, [N.selectivity, N])
18  REVERSE(update) // Now inform all hosts of elements without join partners
19  for every [join, N1, N2] ∈ update
20      UPDATE(join, N1, N2)
```

Algorithm 3 Materialing a resolved plan

```
MERGE-MATERIALIZE(Plan, Solution, Query)
 1  graph = GETGRAPH(Solution) // the molecule graph
 2  resultVariables = GETPROJECTIONS(Query) // the result variables
 3  resolved = [] // the bound result variables
 4  results = [][] // the final table of results
 5  while !resultVariables.empty()
 6      v1 = POP(resultVariables) // get next unbound result variable
 7      if resolved.empty() // no currently bound result variables
 8          v2 = GETNEARESTRESULTVARIABLE(v1, resultVariables, graph)
 9          REMOVE(projections, v2); PUSH(resolved, v2)
10      else // there are currently bound result variable
11          v2 = GETNEARESTRESULTVARIABLE(v1, resolved, graph)
12          PUSH(resolved, v2)
13      resultsTable = GETMERGETABLE(v1, v2, graph) // merge partial results (id's only)
14      if results.empty()
15          results = resultsTable
16      else
17          results = EXTEND(results, resultsTable)
18      REMOVEDUPLICATES(results)
19  MATERIALIZE(results) // turn id's into actual strings
20  return results
```

3.2 Executing Plans

Specifically, following Algorithm 2 we start by executing the most selective molecule in the plan (steps 1 and 2 in Figure 2). To perform the join (lines 10-12 in Algorithm 2) we send the results to host B and execute the join there (steps 3 and 4 in Figure 2). Similarly we join the remainder of the molecules. After all join operations have ended, we need to let hosts A and B know of all the elements that did not have a join-partner by updating its structure (lines 18-20 in Algorithm 2; steps 8 to 12 in Figure 2).

To increase execution performance, since many plans contain overlapping subqueries, we employ a *memoization* strategy by keeping partial results on the respective hosts for the duration of the query execution, while at the same time database caching strategies are in effect. As a further improvement, site-level memory caches can be employed, bypassing the database altogether for "popular" result sets.

3.3 Materializing Plans

Once a plan has finished its execution, the `Plan Executor` monitoring the process will signal the AVALANCHE mediator by pushing the executed plan onto the `Finished Plans` queue. Note that the executed plans do not contain the results yet, since the matches are kept as partial tables on their respective hosts. Hence,

plans in the `Finished Plans Queue` will be handled by a `Plan Materializer` that materializes the partial results as described in Algorithm 3. First, we get an unbound result variable $v1$ (line 6). We then try to find the next possible result variable that will produce the lowest number of merge operations (procedure getNearestResultVariable in lines 8 or 11). Having chosen the next result variable we create a partial result table (line 13) and merge it with the global result table (lines 14-17). We finish by removing duplicates and replacing all ids with the actual strings (lines 18 and 19). To further reduce the overhead of sending the results between hosts, we use RLE compression.

3.4 Stopping the query execution

Since we have no control over distribution and availability of RDF data and SPARQL endpoints, providing a complete answer to the query is an unreasonable assumption. Instead, the `Query 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
- 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. Using Chebyshev's inequality we stopped when $1 - \frac{1}{\sigma^2} > 0.9$.

4 Preliminary Evaluation

In this section we describe the experimental evaluation of the Avalanche system. We first succinctly introduce the setup and then discuss the two evaluated properties: the query execution and plan convergence.

4.1 Experimental setup

We tested Avalanche using a five-node cluster. Each machine had 2GB RAM and an Intel Core 2 Duo E8500 @ 3.16GHz. We chose this small number of nodes to better illustrate Avalanche's query handling strategies, but did not measure its ability to scale.

The data was gathered directly from the LOD cloud. Specifically, we employed the IEEE (66K triples), DBLP (22 millions) and ACM (13 millions) publication data. The datasets were distributed over a five-node cluster, split by their origin and chronological order (i.e. ACM articles till 2003 on host A) as shown in Table 4.1. Recall that as stated above Avalanche makes no assumptions about the data distribution over the nodes.

For the purpose of evaluating Avalanche we selected 3 SPARQL queries as listed in Appendix A. The queries were chosen in increasing order of complexity

Host	# Triples	# S / O	# DBLP P	# ACM P	# IEEE P
Host A	7058949	1699554	0	18	0
Host B	6549326	1554767	0	18	14
Host C	6547513	2153509	20	0	17
Host D	8319504	2773740	19	0	0
Host E	7399881	2680160	19	0	0

Table 1. Number of triples, unique subject S, object O, and predicate P distributions on the hosts. Predicates are shown by dataset.

(in terms of number of unbound variables and triple patterns). We conducted all query executions with the following parameters: 1) *timeout* set to 20 seconds, 2) a *stop sliding window* of size 5, 3) a *saturation threshold* of 0.9, and 4) a *selectivity threshold* for bloom filter construction of 1000 while searching for a maximum of 200 results.

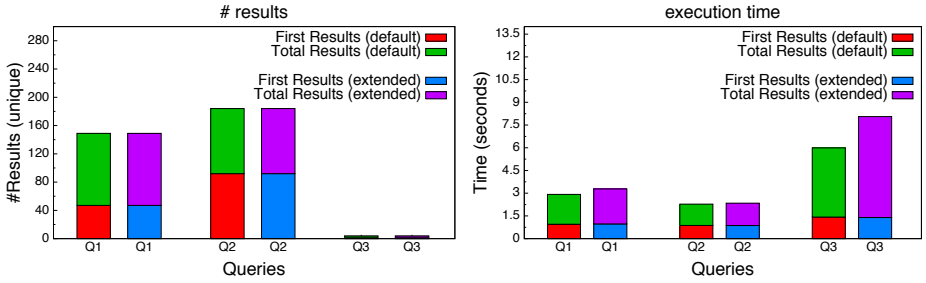

Fig. 3. Number of retrieved results and query execution times

Query execution Figure 3 graphs the number of query results (left) and the execution time (right) for both the default utility U and the extended utility EU introduced in Section 3. Note that the query execution time for the extended utility is somewhat higher (lower than the timeout), but it does find more answers to the queries. The time used for the extended utility is higher since gives the better plans a higher priority and executes them earlier. The execution of "useful" plans does take longer, since a non-useful plan is stopped as soon as an empty join is discovered. Hence, the saturation condition will stop the default utility earlier after having executed fewer useful plans. Given a large number of hosts we expect that the overhead of cancelling non-useful plans will overcome the cost of executing useful plans. Hence, the extended utility planer should converge faster.

As we see in this experiment, Avalanche is able to successfully execute query plans and retrieves many up-to-date results without having any prior

knowledge of the data distribution. We, furthermore, see that different objective functions have a significant influence on the outcome and should play a critical role when deployed on the Semantic Web.

Planner convergence A second issue we planned to investigate is the usefulness of the convergence criteria introduced in Section 3.4. Figure 4 graphs the total number of results against the number of new results where the data points represent newly arriving—possibly empty—answer sets whilst disabling the stopping condition.

As an example consider query $Q1$. At first, the number of new results grows to a certain level. But, after having gathered ≈ 140 results, no more new results are received. A similar behavior can be seen for each of the three queries. Hence, given the experimental results the choice of a stopping condition is pertinent. The current stopping conditions would stop both queries $Q1$ and $Q3$ at the right point when the correct plateau is reached. When considering the number of results found (see also Figure 3), query $Q2$, however, is stopped somewhat early in one of the local plateaus.

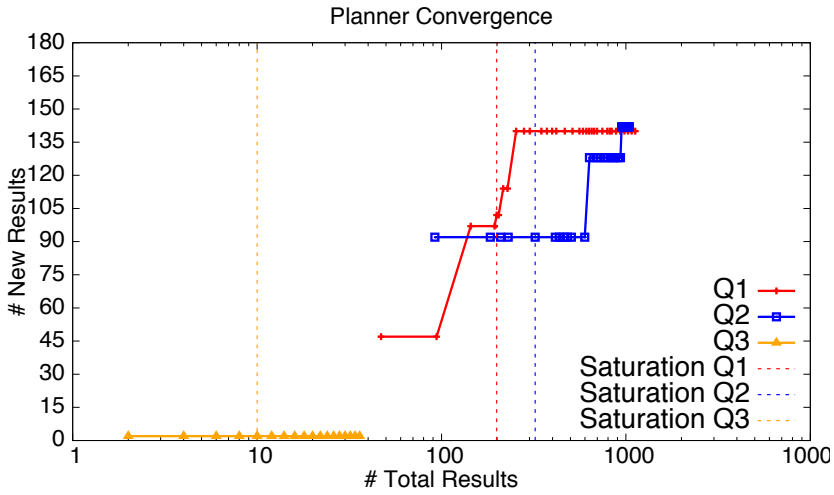

Fig. 4. Query planner convergence

5 Limitations, Optimizations and Future Work

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 truly scales to large number of hosts. Here we discuss each of these issues in turn.

The core optimization of the Avalanche system lies in its cost and utility function. The basic utility function only considers possible joins with no information regarding the probability of the respective join. The proposed utility extension UE estimates the join probability of two highly selective molecules. Although this improves the accuracy of the objective function, its limitation to highly selective molecules is often impractical, as many queries (such as our example query) combine highly selective molecules with non-selective ones. Hence, we need to find a probabilistic distributed join cardinality estimation for low selectivity molecules. One approach might be the usage of bloom-filter caches to store precomputed, "popular" estimates. Another might be investigating sampling techniques for distributed join estimation.

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

- report statistics: **cardinalities**, bloom filters, other future extensions
- support the execution of **distributed joins** (common in distributed databases), which could be delegated to an intermediary but would be inefficient
- share the same **same key space** (can be URIs but would result in bandwidth-intensive joins and merges)

Whilst these requirements seem simple we need to investigate how complex these extensions of triple-stores are in practice. Even better would be an extension of the SPARQL standard with the above-mentioned operations, 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—may not be applicable as the size of the state (i.e., the partial results) may be huge and overburden the available bandwidth.

We designed Avalanche with the need for high scalability in mind. 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 1, *symmetry* by allowing each endpoint to act as the initiating Avalanche node for a query caller, and *fault tolerance* through a number of time-outs and stopping conditions. Nonetheless, an empirical evaluation of Avalanche with a large number of hosts is still missing—a non-trivial shortcoming (due to the lack of suitable, partitioned datasets and the significant experimental complexity) we intend to address in the near future.

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, (2) provides up-to-date results, and (3) is flexible since it assumes nothing 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 two possible utility functions to guide the planning and execution over the distributed data-sources—the basic simple model and an extended model exploiting join-estimation. We found that whilst the simple model found some results faster it did find less results than the extended model using the same stopping criteria. We believe that if we were to query huge information spaces the overhead of badly selected plans will be subdued by the better but slower plans of the extended utility function.

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 a first important towards bringing the spirit of the web back to triple-stores—a major condition to fulfill the vision of a truly global and open Semantic Web.

Acknowledgements This work was partially supported by the Swiss National Science Foundation under contract number 200021-118000. We would like to acknowledge Cathrin Weiss and Rob H. Warren for their help and contribution in the development and evolution of the ideas behind Avalanche.

References

1. C. Bizer, T. Heath, and T. Berners-Lee. Linked data - The story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 2009.
2. A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.
3. M. Cai and M. R. Frank. Rdfpeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *13th International World Wide Web Conference (WWW)*, pages 650–657, 2004.
4. O. Erling. Virtuoso. In *http://openlinksw.com/virtuoso/*.
5. A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, and J. Umbrich. Data summaries for on-demand queries over linked data. In *19th International World Wide Web Conference (WWW)*, May 2010.
6. A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: a federated repository for querying graph structured data from the web. In *6th International Semantic Web Conference (ISWC)*, pages 211–224, 2007.
7. O. Hartig, C. Bizer, and J.-C. Freytag. Executing SPARQL queries over the Web of linked data. In *8th International Semantic Web Conference (ISWC)*, page 293309, October 2009.
8. O. Hartig and R. Heese. The SPARQL query graph model for query optimization. In *4th European Semantic Web Conference*, June 2007.

9. D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.
10. A. Langegger and W. Wöß. RDFStats - An extensible RDF statistics generator and library. In *8th International Workshop on Web Semantics, DEXA*, September 2009.
11. A. Langegger, W. Wöß, and M. Blöchl. A Semantic Web middleware for virtual data integration on the web. In *5th European Semantic Web Conference*, June 2008.
12. O. Lassila. Programming Semantic Web applications: a synthesis of knowledge representation and semi-structured data Doctoral. *Doctoral dissertation*, 2007.
13. A. Maduko, K. Anyanwu, and A. Sheth. Estimating the cardinality of RDF graph patterns. In *16th International World Wide Web Conference (WWW)*, May 2007.
14. T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *36th International Conference on Management of Data (SIGMOD)*, June 2010.
15. B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. *The Semantic Web: Research and Applications*, pages 524–538, 2008.
16. S. Ramesh, O. Papapetrou, and W. Siberski. Optimizing distributed joins with bloom filters. In *ICDCIT '08: Proceedings of the 5th International Conference on Distributed Computing and Internet Technology*, pages 145–156, 2009.
17. S. Schenck and S. Staab. Networked graphs:a declarative mechanism for SPARQL rules, SPARQL views and RDF data integration on the web. In *17th International World Wide Web Conference (WWW)*, April 2008.
18. A. P. Sheth and J. A. Larson. Federated databases systems for managing distributed, heterogeneous and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.
19. M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *17th International World Wide Web Conference (WWW)*, April 2008.
20. J. Zemánek, S. Schenk, and V. Svatek. Optimizing SPARQL queries over disparate RDF data sources through distributed semi-joins. In *7th International Semantic Web Conference (ISWC)*, October 2007.

A Appendix

Query 1:
```
SELECT  ?title ?author ?date   WHERE {
        ?paperDBLP <http://www.aktors.org/ontology/portal#has-title> ?title .
        ?paperDBLP <http://www.aktors.org/ontology/portal#has-author> ?author .
        ?paperDBLP <http://www.aktors.org/ontology/portal#has-date> ?date .
        ?author <http://www.aktors.org/ontology/portal#full-name> "Abraham Bernstein" .
    }
```
Query 2:
```
SELECT ?name ?title WHERE {
        ?paper <http://www.aktors.org/ontology/portal#has-author> ?author .
        ?author <http://www.aktors.org/ontology/portal#full-name> ?name .
        ?paper <http://www.aktors.org/ontology/portal#has-author> ?avi .
        ?paper <http://www.aktors.org/ontology/portal#has-title> ?title .
        ?avi <http://www.aktors.org/ontology/portal#full-name> "Abraham Bernstein" .
    }
```
Query 3:
```
SELECT ?title ?date WHERE {
        ?author <http://www.aktors.org/ontology/portal#full-name> "Abraham Bernstein" .
        ?paper <http://www.aktors.org/ontology/portal#has-author> ?author .
        ?paper <http://www.aktors.org/ontology/portal#has-title> ?title .
        ?paper <http://www.aktors.org/ontology/portal#has-date> ?date .
        ?paper <http://www.aktors.org/ontology/portal#article-of-journal> ?journal .
        ?journal <http://www.aktors.org/ontology/portal#has-title> "ISWC/ASWC".
    }
```