Undefined 1 (2009) 1–5
IOS Press

x-Avalanche: Optimisation Techniques for Large Scale Federated SPARQL Query Processing

Editor(s): Name Surname, University, Country
Solicited review(s): Name Surname, University, Country
Open review(s): Name Surname, University, Country

Cosmin Başca [a,*] and Abraham Bernstein [a]
[a] *Department of Informatics, University of Zürich, Switzerland*
E-mail: {basca,bernstein}@ifi.uzh.ch

Abstract. Attributes like ease of linking and integration, flexibility and standardisation are making the RDF data model more popular. As a consequence, more RDF data gets published across different domains. This distributed publication of RDF data ethos embodies the spirit of the Web of Data. While centralised RDF storage has gotten more scalable in order to keep up with the increase of published data, the problem of querying large federations of RDF datasets has not received as much attention.

In this paper we extend our existing AVALANCHE federation engine to address some of the most pertinent issues with federated RDF query processing. First, we add *support for disjunctions* by employing a distributed union operator capable of scaling to hundreds or thousands of endpoints. Second, we *enhance the distributed state management with remote caches* aimed to reduce the high latency typical of SPARQL endpoints. Finally, we introduce a *novel and parallel-friendly optimisation paradigm* designed not only to offer an optimal tradeoff between total query execution time and fast first results, but to also consider an extended planning space unexplored so far.

Our results show that combined, these capabilities improve our system's performance by up to 70 times over the best performing SPARQL federation engine and find an optimal performance tradeoff between delivering first results and total query execution time under external constraints.

Keywords: Federated SPARQL processing, Web of Data, Query Optimisation, Dynamic Programming, Planning Space Reduction, k-Segmentation

1. Introduction

In recent years, the RDF data model has received more attention; primarily due to factors that revolve around the data models' flexibility and standardisation. Linking RDF datasets as well as extending them wether with new data, annotations, or new versions is easy. Additionally, the semi-structured format is a natural fit for storing and representing graph data. As a consequence , the amount of published RDF continues to grow steadily. To cope with the growth of individual datasets—for example computational biology RDF datasets can amass to billions of triples such as *uniprot.org*, which has 6.95 billion triples—centralised indexing and storage solutions are becoming more scalable. At the same time the number of RDF datasets also continues to grow, as partly shown by the evolution of the *Linked Open Data* (LoD) cloud[1]. However, unlike centralised storage systems, federated RDF engines have not seen much attention while often pro-

*Corresponding author. E-mail: basca@ifi.uzh.ch

[1] `http://lod-cloud.net/`

viding limited support for the SPARQL 1.1 federation extensions.

1.1. Motivation

Over the years, substantial research has been carried out to address performance issues, location transparency, and to improve the SPARQL 1.1 federation specification [27,11,22,1,6,5,23]. These systems have primarily focused on addressing performance issues that are endemic to the LoD ecosystem. However, not all problematic aspects have been addressed to the same extent. One such issue stems from the LoD's schema richness and broad semantic diversity. In this setup, typical real-world and benchmark queries like the ones from FedBench [26] are *semantically selective* — i.e., the vocabularies bound to the query restrict the execution of the query to only a few endpoints, considerably reducing the size of the problem. Having to deal with only a handful of endpoints at a time simplifies the position of typical SPARQL federation engines. The limitation to of investigations to semantically selective situations can lead to a lack of attention and optimisations that target more difficult scenarios.

There are cases in which the implicit assumption of *semantic selectivity* does not hold. First, it is foreseeable that as the size of the published RDF data continues to grow so is the number of endpoints that are *semantically homogenous*, i.e., store data with the same schema. Second, given the "messiness" of the LoD, which stems from the use of similar yet overlapping vocabularies, it is not uncommon to rewrite SPARQL queries in order to capture more of the potentially relevant data. For these scenarios, the large size of the problem requires:

a) novel and scalable system designs that are not addressed by current methods and standards,
b) novel query optimisation strategies, and
c) updated and comprehensive benchmarks designed to capture the issues of large RDF federations.

Equally important, flexibility has to be taken into account. A flexibly designed RDF federation engine must be compatible to a large degree with the existing SPARQL 1.1 standard and make few or no assumptions about the underlying RDF storage technology.

1.2. Contributions

In this paper we present novel methods, architectural enhancements, and optimisations for federated RDF engines, which, when combined, offer dramatic performance improvements over existing approaches while at the same time maintaining a flexible design.

To ascertain the validity of our hypotheses we fully implemented the methods by extending the AVALANCHE SPARQL federation engine [6]. We refer to the extension as X-AVALANCHE. Specifically, the technical contributions of this paper can be grouped into *Query Execution & Operator Design*, *Optimization*, and *Implementation & Evaluation*. They are as follows:

Optimisation

1. We propose a novel approach to optimally reduce and explore an *extended planning space* for large federations of SPARQL endpoints (that has a parametric and non-parametric variant), where data is partitioned. We also show how to optimally find the largest partial result-set that can be retrieved in the shortest possible time under user / domain defined constraints and given the cost model.
2. We identify a new class of easily parallelizable plans we call *fragmented bushy plans* – the top level logical node is a disjunction of standard plan subtrees.

Query Execution & Operator Design

3. We introduce a novel parallel union operator scalable to hundreds or thousands of endpoints.
4. We present an extended distributed state management protocol with support for disjunctions. Each operator is designed to execute directly or by proxy, i.e., delegate the operator's execution to a remote endpoint. All query executiuon X-AVALANCHE operators rely only on the SPARQL 1.1 protocol.
5. We show that a distributed caching strategy tailored for federated SPARQL queries is able to mitigate to a significant extent the high latency typical of SPARQL endpoints.

Implementation & Evaluation

6. We propose a simple synthetic benchmark based on LUBM [12] and the design of the Waterloo SPARQL Diversity Test Suite or WatDiv [3] with support for different data distributions. We provide an open source implementation of that

benchmark in the *rdftools*[2] project, also containing a description of the queries[3].

7. We present the implementation of the X-AVALANCHE system and *performance measurements* against FedX a state of the art top performing federated SPARQL engine [23], with support for *location transparency*.

The remainder of this paper is structured as follows. Section 2 describes state of the art federated SPARQL query approaches and optimality guarantee optimisation methods. A scalable union operator is introduced in Section 4, while the design decisions for X-AVALANCHE's extended query execution protocol are discussed in Section 5. A detailed evaluation of X-AVALANCHE follows in Section 6. We discuss limitations and future work directions in Section 7 and conclude in Section 8.

2. Background

In the following section we describe related and similar works to our system X-AVALANCHE. They can be grouped into: *federated SPARQL processing* and *query optimisation*. We also briefly describe the original AVALANCHE federation engine.

2.1. Related Work

2.1.1. Federated SPARQL Processing

The continuous growth of the Web of Data (WoD) has given rise to new opportunities and challenges in querying this global repository of distributed but interlinked datasets. The Linked Open Data (LoD)[4] alone amassed over 60 billion assertions spread over more than 1000 datasets spanning a broad spectrum of domains. Typically, data on the LoD is shared either by following W3C's Linked Data[5] guidelines, indexed and exposed via a SPARQL endpoint, or simply available as compressed data dumps. While querying the WoD has seen much attention, in the following we will focus only on federations of SPARQL endpoints, such as those querying the indexed LoD, but not limited to.

One of the earliest approaches to offer *location transparency* materialised in DARQ [22]. Since the SPARQL 1.1 federation extensions were standardised much later, the authors relied on their own RDF-based representation of *service descriptions*. These provided a declarative way to describe the indexed data alongside useful statistics, which were valuable during query optimisation. A second wave of research has given birth to several more SPARQL federation engines. In FedX [27], another virtual integrator of SPARQL endpoints, the authors develop new join execution strategies designed to minimise the number of requests sent to participating endpoints. Unlike FedX which makes use of a *rule-based* or *heuristic* query optimiser, SPLENDID [11] features a Dynamic Programming (DP) cost based optimiser able to guarantee plan optimality – within the confines of the cost model. The authors overcome one of the major impediments to using traditional database techniques for federated SPARQL processing by extracting advanced statistics from voID[6] endpoint descriptors. When voID statistics are not available, SPLENDID reverts to using ASK queries when selecting source endpoints.

A series of factors endemic to the WoD such as *i*) uncontrollable network conditions, i.e., no guarantees can be made about latency, bandwidth or availability, *ii*) inaccurate statistics, i.e., continuous data growth in both number of datasets and size, as well as *iii*) dynamic data and workload, have prompted the adoption to various degrees of *adaptive query processing* methods. For example, ADERIS [17] a mediator based federation, utilises adaptive join reordering given a predefined cost model. ANAPSID [1] is adaptive during query execution as well as during source selection. Exhibiting an intra-operator flavour of adaptivity the system features a non-blocking operator design. In contrast, AVALANCHE [6,5] features an inter-operator adaptive query execution design. Statistics about cardinalities and data distribution are obtained before query execution and used during optimisation. The system uses a fragmented execution model were top-k partial plans are executed concurrently, until user defined termination conditions are met or the plan fragment space is exhausted.

Given the sheer size of the LoD, recent research into federated SPARQL querying has focused more on the parallelism aspect of query processing. For instance, LHD [30] like previous systems makes use of a variant of the popular selectivity based cost model, coupled with a parallel execution system that exploits stream-

[2] https://github.com/cosminbasca/rdftools

[3] https://github.com/cosminbasca/rdftools/blob/master/doc/DESCRIPTION.md

[4] http://stats.lod2.eu/

[5] http://www.w3.org/TR/ldp-bp/

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

ing in order to minimise query execution time. An extension of FedX, FedSearch [19] a hybrid federation search engine, is designed to execute combined structured SPARQL queries with full-text search. The system employs on-the-fly adaptation of the query plan and is optimised to execute top-k hybrid search queries over multiple data sources. Finally, in [24] the authors focus on the problem of duplicate data on the WoD. The proposed method, DAW which is used to extend the DARQ, SPLENDID and FedX federation engines, shows great promise in reducing the number of queries sent to endpoints.

2.1.2. Query Optimisation

The ideal query optimiser would feature the lowest optimisation time (a small search space) and optimal plans. In the centralised case, the size of the search space is primarily governed by the number of joins in the query. Of secondary concern, the shape of the query can be used to further reduce the search space, i.e., leverage the fact that the query contains star-patterns.[7] When resolving complex federated SPARQL queries, query optimisers typically switch to a rule-based mode of operation in order to cope with the large planning space and answer the query in reasonable time. This is undesirable since *1*) the optimiser is forced to *drop any optimality guarantees* and *2*) using heuristics worsens the problem of *accurately estimating the cost* of complex queries [15].

As analysed in previous works [20], the time complexity of a DP optimiser in a centralised DBMS is $\mathcal{O}(3^n)$, where n is the number *triple patterns*. Similarly, the space complexity is $\mathcal{O}(2^n)$. One way to reduce the optimisation time is to adapt the general DP approach as described in [16] by applying DP several times iteratively, while optimising the query. The method is known as Iterative Dynamic Programming (IDP) and features a reasonable polynomial time complexity, but does not guarantee overall optimal plans when more than one iteration of DP is performed. It does, however, find the optimal plan under the imposed resource constrains. The number of iterations can be controlled by the database administrator or adjusted automatically considering resource allocation (e.g., memory or time). In a distributed context, when data is replicated at different sites, the size of the search space explodes in the worst case. In this case the time complexity of a classic DP optimiser is $\mathcal{O}(s^3 * 3^n)$, while its space complexity falls into the $\mathcal{O}(s * 2^n + s^3)$ class, where s represents the number of sites that hold data.

For partitioned setups, however, traditional DP optimisers consider partitions usually at the leaf nodes as physical unions between sites, conveying the advantage of a reduced planning space over the replicated setup. In doing so, however, an *extended planning space* that can contain better join and union orderings is left unexplored, as this would again lead to an explosion of the search space. A first attempt to partially explore this *extended planning space* is presented in [13], where the authors propose a method suitable for both centralised and parallel relational DBMS'. The central innovation of the algorithm is the introduction of a clustering phase aimed at discarding unnecessary child table (partitions) joins during planning, when information about which partitions can join is present. While this method is applicable for *range, list* or *hash* partitioning schemes typically encountered in such systems, it is not suitable for the global and uncontrollable nature of the Web of Data.

2.2. Avalanche

Here, we give a brief overview of our previous AVALANCHE federated SPARQL engine. AVALANCHE's architecture is organised to accommodate a three phase execution model. First, relevant endpoints are identified. Second, query specific cardinalities are retrieved and finally the query planning and execution phases follow. While finding out the cardinality of a basic graph pattern (BGP) can be expensive operation for an RDF store, aggressive indexing techniques like the ones implemented by RDF-3X [18] and Hexastore [31] or high performance implementations such as Virtuoso[8] allow for fast retrieval of *triple pattern cardinalities*. Furthermore, voID[9] [2] descriptions of the indexed data can accomplish the same. If no catastrophic SPARQL endpoint failures occur, the system is *eventually complete*.

Tailored to address the semantic heterogeneity and lack of guarantees that are characteristic to the WoD, AVALANCHE employs a *fragmented* execution model. Here, the query is decomposed into the union of all *query fragments*. A query fragment, or fragment in short, is defined as the conjunction of all query triple patterns with the restriction that a triple pattern can

[7] star-patterns are common when retrieving resource attributes

[8] https://github.com/openlink/virtuoso-opensource

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

be resolved by one host, i.e., no disjunctions allowed inside a fragment. AVALANCHE enumerates all fragments using a priority queue based repeated depth first search traversal algorithm. This allows AVALANCHE to *1*) generate fragments in a given order, i.e., favour faster and productive fragments and *2*) execute all or K fragments concurrently at any given moment, i.e., dynamically adapting to network conditions and endpoint availability.

3. Optimisation

Currently, the LoD exhibits high semantic selectivity and limited dataset partitioning. This is primarily due to its schema richness and broad semantic diversity, which leads to a drastic reduction of the number of participating SPARQL endpoints when querying. Designed to target the current state of the LoD, the original AVALANCHE federation engine exhibits a number of shortcomings. First, while it features a multi-fragment concurrent execution model, it does not provide support for disjunctions. Second, AVALANCHE does not statically optimise each plan fragment and instead it employs a non-optimal *greedy execution strategy* (GRDY), where the order of each join is decided on the fly. Finally, AVALANCHE makes use of a selectivity based cost model to decide on the order in which fragments are generated, and does not attempt to reduce the size of the planning space.

However, as the size of the LoD continues to grow, we expect to see a decrease in semantic selectivity and an increase of dataset partitioning. Consequently, new solutions are warranted and therefore, this section introduces new optimisation approaches that are more suitable for these scenarios, optimisations which are built into X-AVALANCHE – an extension of AVALANCHE.

3.1. Cost Model and Optimisation Strategies

X-AVALANCHE is designed to address large RDF federations where data is horizontally partitioned between many endpoints and where *semantic selectivity* has a negligible or low impact. In other words it is designed to deal with large and semantically homogenous distributed datasets. In order to proceed further, we relax the notion of a *plan fragment* as used by AVALANCHE and redefine it as follows:

Definition 3.1 *A **plan fragment** is a query plan for which only a subset of all participating sites* $\mathbf{s}$ *are considered.*

In other words, a (conjunctive) plan fragment can also contain disjunctions or unions. Defined as such (Definition 3.1), a query can have between 1 ($\forall$ triple pattern bound to all sites) and s^n fragments ($\forall$ triple pattern bound to one site).

A first enhancement over AVALANCHE is that each fragment can now be statically optimised in contrast to the greedy execution strategy. For this purpose we employ the classic *dynamic programming* (DP) [7] method. Since DP features a worst case time complexity of $\mathcal{O}(3^n)$ [20] for n triple patterns, we consider the following simplifying assumptions in order to reduce the plan space:

- Like in System R [4] we explore only *left-deep* plan trees and avoid cross-products whenever possible.
- The order of the join operands is ignored during the planning phase and determined at runtime, i.e., always ship the smallest bindings set.

Furthermore, since network communication introduces the highest latency during query execution, we rely on the simplifying assumption that *the number of partial results has the highest impact on performance*. Therefore, we base the cost model used to optimise each plan fragment on the estimation of the query's selectivity [29]. Equations 1 and 2 show how the cardinality of joining and unioning two triple patterns tp_1 and tp_2 is estimated, where Θ represents the total number of triples.

$$|tp_1 \bowtie tp_2| = |tp_1| \times |tp_2| \times \frac{|tp_1| + |tp_2|}{2 \times \Theta}, \quad (1)$$

$$|tp_1 \cup tp_2| = \max\left(|tp_1|, |tp_2|\right) \quad (2)$$

While naïve, this model of partial result size estimation has the advantage that no other statistics are needed aside from triple pattern cardinalities or estimates of. However, this comes at a cost: less accurate estimations can, in practice, render some plans much more expensive than estimated. As results from Section 6 show, this model, while simple, was able to dramatically improve performance over a top performing state of the art approach.

3.2. Extended Space Reduction

To show how the optimal partition-aware union grouping method works, we reuse the notion of *plan matrix* from [6]. The *plan matrix* or $\mathcal{PM}$ is a compact representation of the cardinalities of all query triple patterns on all sites as follows:

$$\mathcal{PM} = \begin{bmatrix} card_{0,0} \cdots card_{0,n} \\ \vdots \quad \ddots \quad \vdots \\ card_{s,0} \cdots card_{s,n} \end{bmatrix} \tag{3}$$

where s and n represent the number of sites and triple patterns respectively, while $card_{i,j}$ is the cardinality[10] of triple pattern i on site j. For simplicity, the remainder of this paper only considers plans that are constructed with conjunctions ($\bowtie$) and disjunctions ($\cup$). While not a trivial matter and outside the scope of this work, support for OPTIONAL and FILTER graph patterns could be provided by relying for example on their mapping to relational algebra operators [10].

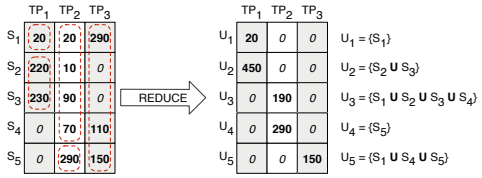

Fig. 1. Example $\mathcal{PM}$ and a possible reduced $\mathcal{PM}^*$. S_i represent the sites holding data, while TP_j represent *triple-patterns*.

Note that a plan matrix $\mathcal{PM}$ of size (s, n) may lead to up to s^n plan fragments. Consider for a moment the unlikely case, where each triple pattern tp_i can be matched on every site s_j (i.e., $\mathcal{PM}$ contains no zeros). A valid plan fragment can now be constructed by choosing one of the sites for each triple pattern. As there are s sites to choose from, there are s valid choices for each of the n triple patterns resulting in s^n possible fragments. Obviously, this plan space is too large in the worst case. Heuristics-based algorithms circumvent the problem of a large $\mathcal{PM}$ by employing specific rules to prune the majority of possible plans.

In this paper, in contrast, we propose to employ *partition-aware union grouping* to reduce $\mathcal{PM}$, resulting in fewer plans to consider. The spirit of the solution is to use union operations to merge the data from different sites thereby reducing the plan space. Specifically, the method maps $\mathcal{PM} \mapsto \mathcal{PM}^*$, where $\mathcal{PM}^*$ is the *reduced plan matrix* of the *extended planning space*. This gives rise to the following research question: ***How can $\mathcal{PM}$ be reduced, and how can it be done optimally?***

As an example (used in the remainder of this section), consider the simple $\mathcal{PM}$ illustrated in Figure 1 alongside a possible reduction $\mathcal{PM}^*$. The reduced plan matrix $\mathcal{PM}^*$ will always have the same number of columns, but fewer or equal number of rows. In essence, this transform introduces 0s in some of the reduced matrix's cells to limit the number of possible fragments that can be constructed. In other words the general idea is to *reduce* the size of each column individually, i.e., by grouping together sites given a criteria for group *fitness*.

In the remainder of this section, we introduce the novel concept of *fragmented bushy-plans* and proceed with detailing two approaches to reducing the plan matrix $\mathcal{PM}$, inspired from data-analysis, a non-parametric and a parametric method.

3.2.1. Fragmented Bushy Plans

Traditionally, a logical query plan is represented as a tree where the non-leaf nodes are algebraic operators while the leaf nodes represent data. As stated in the introduction, the time complexity of a DP optimiser which does not consider disjunctions during the logical planning phase is $\mathcal{O}(3^n)$. In the worst case, between any two joining triple patterns that are fully partitioned on all sites, there would be $n * s$ partition joins. There are an exponential number of possible ways in which a union leaf node in a logical plan can be split into a combination of sub-unions and there are n such union nodes. The higher order exponential space complexity of the *extended planning space* prevents us from exploring all possible plans. Instead, to benefit from parallelism, we consider to explore only a special class of bushy plans, which we call *fragmented bushy plans*.

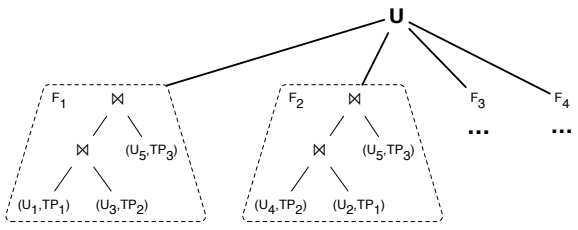

Fig. 2. A possible *fragmented bushy plan* for the example $\mathcal{PM}^*$ from Figure 1. The plan consists of 4 fragments, each equivalent to a left-deep logical plan tree.

[10] exact cardinalities are not required, estimations suffice.

Given the construct of a *plan fragment* as outlined in Definition 3.1, a *fragmented bushy plan* is defined as follows:

Definition 3.2 *A **fragmented bushy plan** is a logical plan whose root node is a disjunction between multiple **plan fragments**.*

As an example, Figure 2 is a partial illustration of a possible fragmented bushy plan extracted from the reduced plan matrix $\mathcal{PM}^*$ from Figure 1. The primary benefits of fragmented bushy plans are twofold:

- the size of the *reduced extended planning space* (explained by $\mathcal{PM}^*$) can be directly controlled by varying the maximum number of desired unique fragments or ϕ, and
- they are easily parallelisable, since each fragment is independent and can be executed concurrently.

Consequently, in order to explore the new *extended planning space* X-AVALANCHE's optimiser pipeline consists of two general phases:

1. *reduce* the plan matrix $\mathcal{PM}$ and
2. *optimise* each fragment in parallel.

As a result, the optimiser's overall time complexity is $\mathcal{O}(k*n^2*s^2+p*3^n)$, where $p = \frac{\phi}{\#CPU}$ is a constant factor regarding parallelism.

3.2.2. Non-Parametric Optimal Reduction

The primary appeal of these methods is that they do not require user-intervention to decide how to best reduce $\mathcal{PM}$. In the remainder of this section, we first detail how a column can be reduced and then show how this method can be used to reduce $\mathcal{PM}$.

A class of methods which can be used to achieve the reduction of each column in $\mathcal{PM}$ are *change point* or *step detection* methods [21]. Widely used in statistical analysis, these methods try to identify when the probability distribution of a series of events changes, resulting in a *change-point*. Given this information, the original set of events can be approximated by a piece-wise constant model, a process we refer to as *segmentation*. One such method is *bayesian blocks*, detailed in [25], which achieves an optimal reduction of its input (in our case a $\mathcal{PM}$ column) by employing dynamic programming or DP in short. Applied to each column in $\mathcal{PM}$, it features a time complexity of $\mathcal{O}(n*s^2)$ with an $\mathcal{O}(s)$ space complexity, where s is the number of sites and n the number of columns (or triple patterns in query).

While the primary advantage of such methods is that they are parameter agnostic, they do however require ex-ante knowledge about the prior distribution of the data to be segmented. In our case, they require knowledge about the prior probability distribution of triples to participating sites. This can be problematic as data distributions may change, requiring re-learning the prior in order to produce higher quality plans.[11]

Algorithm 1 Non-Parametric Plan Matrix Reduction

Precondition: $\mathcal{PM}$: the cardinalities matrix of size $s \times n$, p_0: the prior probability distribution

```
1: function AUTOREDUCE(PM)
2:     cols ← ∅                          ▷ the columns of PM*
3:     s, n ← SHAPE(PM)                  ▷ shape: (rows, columns)
4:     for δ ∈ PM do                     ▷ iterate over all columns ∈ PM
                                         ▷ optimum segmentation of column given p0
5:         S ← BAYESIANBLOCKS(δ, p0)
6:         cols ← cols ∪ {Σ δ[σ] | ∀σ ∈ S}
                                         ▷ outer join of all reduced columns
7:     return ⟗ cols
```

The bayesian-blocks algorithm can be used to reduce $\mathcal{PM}$ as seen in Algorithm 1. Iterating over all columns in $\mathcal{PM}$ (line 4), the method retrieves the optimal segments $\mathbb{S}$ for the current column δ (line 5). It than constructs the reduced or segmented column by replacing all cardinalities within each segment $\sigma \in \mathbb{S}$ with their sum (line 6). Other aggregate functions could be used to get better estimates of the size of the resulting union over the given segment. We chose Σ since it represents the upper bound of the estimated cardinality of the union. Finally, the newly reduced columns are concatenated in matrix $\mathcal{PM}^*$, by employing a *full outer join* (line 7).

3.2.3. Parametric Optimal Reduction

Methods in this class expect the user to pass domain knowledge encoded as parameters. While tedious, this form of *loose coupling* exhibits the major advantage of ease of adaptation when the domain changes or when encoding this knowledge is difficult or expensive. In our case, the domain knowledge is represented by the distribution of cardinalities (or selectivities) of query triple patterns to sites, which is expected to change as data diversifies and its volume increases over time.

To this end, we adapt the traditional 1D *k-segmentation* method. Unlike the non-parametric bayesian-blocks method which was applied locally to reduce each column in $\mathcal{PM}$, the global parameter ϕ

[11] we used the same prior p_0 as in [25].

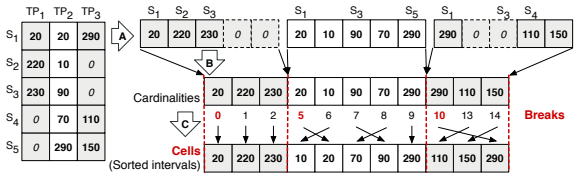

Fig. 3. Preparing $\mathcal{PM}$ for reduction. $\mathcal{D}$ (*cells*) is the array of non-zero cardinalities from $\mathcal{PM}$ in **column major order** form, while B (*breaks*) encodes the position of the columns in $\mathcal{PM}$.

(the number of segments) requires that the method be applied to the entire $\mathcal{PM}$ and not individually to its columns. To achieve this, $\mathcal{PM}$ is represented as an array in *column major order*, i.e., as a 1D array comprised of the concatenation of all columns in the order in which they appear in $\mathcal{PM}$. Figure 3, illustrates the process of representing the plan matrix as a 1D array $\mathcal{D}$ in *column major order* form. In order to avoid the creation of segments that would span across multiple columns and therefore invalidating the semantics of the original SPARQL query, we introduce a helper structure referred to as *breaks* ($\mathcal{B}$), which is a list holding the starting index of each $\mathcal{PM}$ column in $\mathcal{D}$. The structure is used by the DP algorithm to set the cost on any segment spanning over multiple columns to ∞ (line 4 in Algorithm 2).

A DP algorithm, *ksegb* finds the optimal set of segments of $\mathcal{PM}$ – in *column major order* form, $\mathcal{D}$ – with the restriction that any segmentations with segments containing elements from $\mathcal{B}$ are ignored. The algorithm's time complexity is $\mathcal{O}(k*n^2*s^2)$, where k is the number of segments and a parameter of *ksegb*. The method is exhaustive as it explores all possible segmentations of $\mathcal{D}$. To find the optimal segmentation, the fitness function (line 1) computes the max-min delta of a segment. We base this formulation on the simplifying assumptions that:

1. all unions are executed in parallel, and
2. the time to execute a union is primarily dependent on the selectivity of the given triple pattern.

Therefore, the fitness function of a segment is intended as a measure of *wasted time*. In cardinality-homogenous segments all sites finish around the same time, while heterogeneous segments incur waiting times on sites with lower cardinalities.

The transition from the original parameter ϕ representing the number of plan fragments in $\mathcal{PM}$ to the number of segments required by *ksegb* is performed using the formula from Equation 4.

Algorithm 2 k-Segmentation with Breaks

Precondition: $\mathcal{D}$: numeric array containing data to be segmented, k: desired number of segments, $\mathcal{B}$: integer array with index bounds of non-breakable segments from $\mathcal{D}$

```
1:  function COST(D, j, i, B)
2:      for b ∈ B do
3:          if j < b ≤ i then
4:              return ∞
5:      return MAX(D[j : i]) - MIN(D[j : i])

6:  function KSEGB(k, D, B)
7:      N ← LENGTH(D)
        ▷ matrix of size (k, N), elements initialised to ∞ cost
8:      DP ← MATRIX(k,N,∞)
        ▷ matrix of size (k, N); for solution reconstruction
9:      PT ← MATRIX(k,N,0)

10:     for 0 ≤ j < k do                          ▷ initialisation
11:         DP[j, j] ← 0
12:     for 0 ≤ i < N do
13:         DP[0, i] ← COST(D,0,i,B)
14:     for 1 ≤ j < k do
15:         for j + 1 ≤ i < N do
16:             C ← {DP[j − 1, l] + COST(D, l + 1, i, B) |
        ∀l ∈ [0, i)}
17:             best ← ARGMIN(C)
18:             DP[j, i] ← C[best]
19:             PT[j, i] ← best
                                          ▷ final solution reconstruction
20:     return SOLUTION(PT, k, N)
```

$$k = n * \sqrt[n]{\phi} \tag{4}$$

The parametric reduction method detailed in Algorithm 3 starts by preparing the input for the *ksegb* method. It first represents $\mathcal{PM}$ in *column major order* form (line 5) after computing the number of segments k. It then records the start positions of the original columns in $\mathcal{B}$ (line 6). Next, it obtains the optimum segmentation of the transformed $\mathcal{PM}$ (line 7). Afterwards, it proceeds to constructing the reduced columns, by replacing all cardinalities within each segment $\sigma \in \mathbb{S}[i : j]$ with their sum (line 10), following the same rationale as in Algorithm 1. Finally, the newly reduced columns are concatenated in matrix $\mathcal{PM}^*$, by employing a *full outer join* (line 11).

Algorithm 3 Parametric Plan Matrix Reduction

Precondition: $\mathcal{PM}$: the cardinalities matrix of size $s \times n$, ϕ: the maximum number of plan fragments to reduce to

1: **function** REDUCE($\mathcal{PM}$, ϕ)
2: $cols \leftarrow \emptyset$ ▷ *the columns of* $\mathcal{PM}^*$
3: $s, n \leftarrow$ SHAPE($\mathcal{PM}$) ▷ *shape: (rows, columns)*
4: $k \leftarrow n \times \sqrt[n]{\text{MIN}(\phi, s^n)}$
5: $\mathcal{D} \leftarrow$ TOCOLUMNMAJORORDERFORM($\mathcal{PM}$)
6: $\mathcal{B} \leftarrow$ COLUMNPOSITIONS($\mathcal{PM}$, $\mathcal{D}$)
7: $\mathbb{S} \leftarrow$ KSEGB(k, $\mathcal{D}$, $\mathcal{B}$) ▷ *optimum k-segmentation*
8: ***for*** $(i, j) \in \mathcal{B}$ ***do*** ▷ *(begin, end) of each column*
9: $\delta \leftarrow \mathcal{D}[i : j]$
10: $cols \leftarrow cols \cup \{\sum \delta[\sigma] \mid \forall \sigma \in \mathbb{S}[i : j]\}$
▷ *outer join of all reduced columns*
11: ***return*** $\bowtie cols$

3.3. Parametric vs. Non-Parametric Fragmentation

Naturally, an automatic or non-parametric reduction of $\mathcal{PM}$ is preferred, given that the optimiser or administrator does not have to be concerned with specifying extra parameters. Such a choice, however, leads to the following question: *how does the non-parametric bayesian blocks method perform in general?* or more specifically, *how does bayesian blocks perform in automatically choosing the number of segments ϕ?* To find out the answer to this question, we conducted an exploratory analysis of the bayesian blocks method over synthetically generated data. Hence, we randomly generated synthetic $\mathcal{PM}$ data which simulates the case of a medium-sized 10 triple pattern SPARQL query, with a random distribution of triples to endpoints.

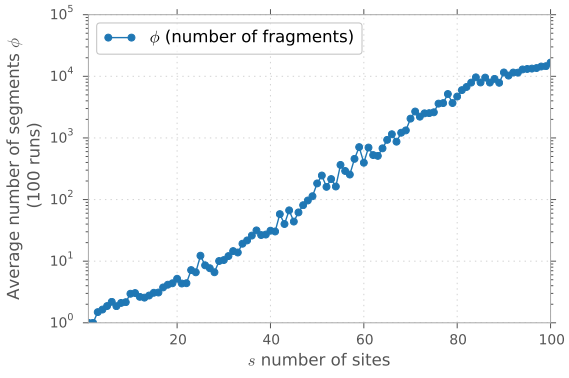

Fig. 4. Number of fragments ϕ automatically selected by *bayesian blocks* function of number of sites s, for a 10 triple pattern query.

Figure 4 shows the relationship between the number of fragments ϕ and the number of sites s, when s increases from a centralised setup to a large federation of 100 SPARQL endpoints. Each datapoint represents the average of 100 runs over randomly generated cardinalities while incrementing the number of sites. A clear observation is that when the number of sites increases over a particular threshold, ≈ 50 for this analysis, the number of fragments automatically chosen by the *bayesian blocks* method starts to increase exponentially. This is undesirable for two reasons:

1. the optimiser cannot choose ϕ in order to control resource wastefulness, and
2. ϕ can, on average, grow very large which diminishes the tractability of the query execution, e.g., more than 10000 fragments for ≈ 90 sites.

In contrast, parametric methods hand control over to the optimiser or the administrator, allowing for the choice of a value that also encompasses resource availability.

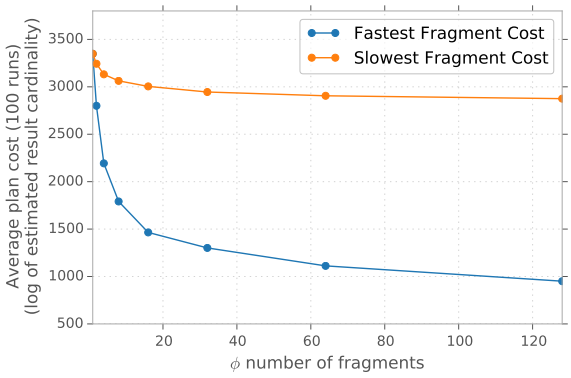

Fig. 5. Quality of plans function of ϕ (maximum number of fragments) for a 10 triple pattern SPARQL query. The cost is equivalent to that of the traditional DP planner when $\phi = 1$.

Another interesting aspect of the $\mathcal{PM}$ reduction process has to do with with the complex relationship between: (*a*) the quality (or cost) of the fastest and slowest fragments and (*b*) the number of chosen fragments ϕ. Figure 5 illustrates this relationship, by comparing the quality of the *fastest* and *slowest* fragments when ϕ is incremented (for parametric methods). We compute the cost o a fragment given the simplifying assumption that query performance is mostly affected by the number of partial results generated by that respective fragment. One can clearly see that the higher the number of fragments, the better the plan fragment quality (lower cost is better). This is no surprise since by fragmenting the original plan, the optimiser ends up dividing the work optimally between participating

sites. However, just like before, having a larger number of fragments incurs resource wastefulness leading to a larger overall system load. This leads the way to the following questions: *What is an optimal number of fragments ϕ and how can it be computed?*

Results from Figures 4 and 5 show that both methods have their own plusses and minuses. Automatic, non-parametric methods like *bayesian blocks* suffer from the need of precise fine tuning to data. Even in such cases, there is no guarantee that the appropriate number of fragments is low enough for query execution to become tractable. In contrast, parametric methods offer the administrator or query optimiser just this: control over the number of segments. On the down side, finding out the appropriate ϕ can be an expensive trial and error process, considering the complex relationship between the characteristics of each participating SPARQL endpoint and the a fragmented query execution.

Consequently, we chose to employ the parametric *k-segmentation* space reduction method as part of X-AVALANCHE's optimisation, primarily due to its intrinsic control over the number of fragments.

3.4. Total/First Results Tradeoff

Since *fragmented bushy plan* are a variant of bushy plans where the top subtrees represent disjoint partitions or fragments of the query plan, they are easily parallelizable given the fact that each fragment is independent. In consequence, they offer control over executing only a portion of the query if needed. This can be advantageous, for example, in multi-query optimisation situations, when the scheduler can choose to interleave the execution of fragments belonging to different queries in an informed way, avoiding the starvation of clients waiting for results from expensive queries. Another situation where plan fragmentation can be beneficial is when FAST FIRST results constraints are imposed by some application, e.g. a search engine requiring results for the first page.

The fragmented execution of any query plan ultimately offers the caller a tradeoff between t, the time until first results are found and T, the total query completion time. Naturally, minimising both performance metrics is desired. To obtain a clear and quantifiable view of this tradeoff we combine both time measurements within the unified performance metric τ. We express τ using the *euclidean* norm to compute the distance to the ideal, $(0, 0)$:

$$\tau = \sqrt{t^2 + (\delta)^2}, \quad \delta = T - t \tag{5}$$

It is our hypothesis that there exists a number of fragments $\phi > 1$ where the τ performance metric is optimal. Additionally, we expect τ to degrade as fragmentation increases over a given threshold due to the fact that overall system occupancy increases in addition to the overhead and interactions introduced by orchestrating the execution of a large number of fragments.

4. Scalable Distributed Unions

When data pertinent to a triple-pattern or subquery is physically partitioned among several sites, the optimiser will have to consider a disjunction between all relevant endpoints in order to guarantee result-set completeness. To simplify matters, most state of the art query optimisers will not consider different grouping strategies during the logical planning phase. Consequently, unions are only applied to the relevant leaf nodes of the plan. If the number of endpoints is large, the physical design of the operator can have a dramatic effect on the overall query execution performance. Consider for example a setup similar to the one illustrated in Figures 1 and 2, with the difference that instead of 5 sites data is partitioned over 100 sites. Such a scenario could lead to unioning triples matching for example TP_1, from 100 endpoints.

While there are many ways in which a distributed union can be carried out, in the following we will focus on methods where computation occurs remotely and not at the client site. Specifically, we investigate parallel execution while considering the naïve serial method as the baseline. The main advantage of the serial method lies in its inherent simplicity: it does not require advanced support, aside from the basic assumption that *2-way unions* can be carried out by simply *shipping* the smaller result set of bindings to the target server and performing the union in-place. Obviously, this serial approach forgoes any performance benefits from parallelism since unioning n sites require in the order of $\mathcal{O}(n)$ union operations.

On the other spectrum from serial execution all binding sets or partial results can be shipped to a previously elected *master* site and 'unioned' in-place. In this case, since all union operations are executed in parallel, the cost of the union falls in the $\mathcal{O}(1)$ com-

plexity class and would theoretically be equivalent with the cost of the most expensive of the union operations. In practice, the *master* site can become a bottleneck when there are many sites, by having to keep n remote connections open at the same time. Furthermore, if duplicate partial bindings are dropped either to reduce traffic or due to the UNIQUE modifier, local contention can mitigate the benefit of parallelism and would require more complex handling strategies that are not implemented by most federated RDF stores.

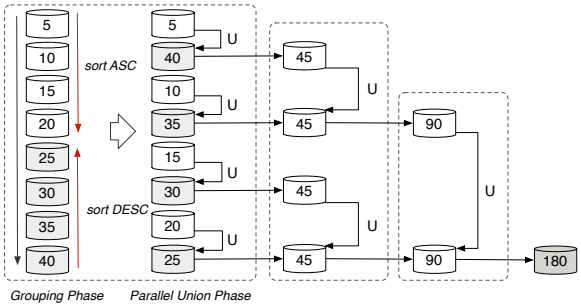

Fig. 6. Example *parallel tree union* for 8 sites. Numbers are subquery cardinalities on each site.

Algorithm 4 parallel tree union

Precondition: $\mathcal{S}$: the participating sites, given subquery sq

```
1: function PARALLELUNION(sq, S)
2:     SORTASC(S)   ▷ sort S on cardinality (ascending)
3:     while ¬EMPTY(S) do
4:         n ← |S|/2
5:         slaves ← S[: n]            ▷ left side of union
6:         masters ← S[n :]           ▷ right side of union
           ▷ sort master sites on cardinality (descending)
7:         SORTDESC(masters)
       ▷ parallel execution of each (slave,master) union pair
8:         S ← PARMAP({s ∪ m | ∀s ∈ slaves, m ∈ masters})
           ▷ the root of the union-tree holds all partial results
9:     return S[0]
```

In the following, we propose a simpler distributed union execution strategy which enjoys both: the benefit of parallelism while at the same time requiring only the simple *2-way union* capability from a participating site. Called *parallel tree-union*, the method uses the topology of a balanced binary tree with endpoints as nodes. The algorithm traverses the tree bottom-up towards the *master* endpoint, by iteratively pairwise unioning each level of leaf nodes. The time complexity for this operator is $\mathcal{O}(log(n))$ for n sites.

As illustrated in Figure 6, within each iteration the sites are divided into two groups: *slave* and *master* sites, where the latter are the ones performing the union. To load-balance the amount of traffic that is generated, the slave with the smallest binding-set ships to the master with the most partial bindings in each iteration (lines 5 - 8 in Algorithm 4). This has the advantage of producing more balanced later stage unions.

5. Distributed State Management & Caching

One of the major factors contributing to X-AVALANCHE's increased performance is the distributed management of partial query results. The SPARQL 1.1 federation extensions are stateless and therefore operate at a lower level than X-AVALANCHE. They are however instrumental building blocks, since X-AVALANCHE relies on: *i*) the COUNT aggregate, to obtain statistics about triple patterns (equivalent statistics can be retrieved using W3C's VoID), *ii*) the SERVICE keyword, to execute a subquery agains a remote endpoint, and *iii*) the VALUES clause, to constrain the results another endpoint. During execution, network traffic is minimised, by keeping materialised BGP views in memory for the duration of the current query as detailed in [6].

Parallel Multicast Joins The introduction of support for disjunctions triggered the addition of support for the execution of parallel joins. Each X-AVALANCHE endpoint can multicast and coordinate a join operation between multiple remote endpoints. All join operations are *bind semi-joins*, where a set of partial bindings is shipped remotely to reduce the execution of the subquery using the SPARQL 1.1 VALUES clause. Consider for example the case of the star query $LQ5$ (Listing 9), where during the execution process bindings for the `?name` variable are restricted to two values: *"GraduateStudent1"* and *"GraduateStudent2"*. Hence, the execution of the remainder triple patterns is bounded on all relevant remote sites of the semi-join, by the two values.

In addition, the source partial results table from which the bindings for the join variable are shipped, can be reconciled using either a *bloom filter* of the remote set of partial bindings if the set is large, or the (compressed) set otherwise. Consider for example that the remote side, or destination, of the semi-join oper-

ation produces partial results only for the *"GraduateStudent1"* binding of the `?name` variable. Therefore all partial records from the source endpoint matching *"GraduateStudent2"* can be safely discarded.

Execution by Proxy In addition, just like p2p systems, all X-AVALANCHE operators can be executed directly or by proxy. Proxy based execution helps the endpoint orchestrating the overall query execution to offload part of the execution orchestration to remote sites while still managing the overall process. This is a particularly useful design since it allows an RDF federation engine more flexible management of remote resources and significantly aids in introducing more parallelism into the query execution pipeline.

For example, consider the *bind semi-join* operation for query $LQ5$ on variable `?name` described earlier. The query execution coordinator can manage the process in two ways. It could orchestrate the process directly by managing each of the phases of the semi-join operation, or it could delegate the management of the entire semi-join operation to the designated source endpoint, therefore benefiting from more I/O and computational resources to coordinate the execution of potentially other concurrent operations.

SPARQL Endpoint Caching Often the performance of the underlying SPARQL endpoint has a negative impact over the RDF federation engine. To mitigate some of the performance penalties incurred, in X-AVALANCHE we enhance the wrapped SPARQL endpoint with a simple cache. The cache cannot be used to store the results of all SPARQL query types. Cacheable queries include: COUNT queries and simple SELECT queries that do not have a VALUES clause. For obvious reasons, queries which contain VALUES variable binding sets cannot be cached. In such cases the key would have to uniquely identify not only the BGP or subquery but also the supplied binding sets. Creating a unique key in this case can be expensive for large binding sets. We employed a typical LRU cache eviction strategy with expiration for records. In practice, the expiration duration should not be larger than the endpoint's dataset update frequency.

6. Evaluation

In this section we present and discuss the results we obtained from evaluating X-AVALANCHE in a controlled setup in order to observe the impact that different external and internal factors have on system performance. Specifically, we first investigate the impact of the parallel union operator followed by a enquiry of the impact of SPARQL endpoint caching. We then explore X-AVALANCHE's performance whilst varying problem size and data distribution. Additionally, we evaluate X-AVALANCHE against the current top performing federated SPARQL engine: FedX, as identified in [23].

Technical setup: We used the latest freely available version of FedX, v3.1.[12] All experiments were run on a cluster of 11 machines, each having 128 GB RAM and two E5-2680v2 @2.8GHz processors, with 10 cores per processor, i.e., equivalent to 20 execution units when *HyperThreading* enabled. Nodes run 64 bit linux (kernel version 3.2.0) and are interconnected using standard 1Gb ethernet. We used Python 2.7.8 and all SPARQL endpoints were powered by Virtuoso v7.1 open source.

6.1. Benchmark Design

X-AVALANCHE is designed to improve query performance in large federations of SPARQL endpoints. However, as mentioned in Section 1.1, the present day LoD's schema richness and broad semantic diversity create a *semantically selective* benchmarking setup, i.e., where the vocabularies used in the query restrict the execution to a handful of endpoints. To address this notion, we distinguish between the selectivity of a query based on the number of result tuples, which we call *result-set selectivity*, and the *source selectivity* of the query. The latter kind of selectivity is the decisive factor during the *source selection* phase and can dramatically improve performance and recall.

Unfortunately, semantically selective benchmarks do not shed any light into how the federation engine performs in worst case scenarios, where hundreds of endpoints are actively engaged in query answering. These scenarios can occur when: *a*) large numbers of sites operate within the same domain, a clear future development as the LoD continues to grow and *b*) the query is *re-written* to use different but similar schemas (i.e., overlapping semantics). In both situations the federated engine has to coordinate the query execution over a large number of endpoints.

In order to observe X-AVALANCHE's performance improvements compared to state of the art federated engines as well as to better understand the impact of internal (configuration) and external (data distribution

[12] `http://www.fluidops.com/`

/ workload) factors in a large federation setting, the benchmark must be able to:

1. scale to as many endpoints as required,
2. allow for data distribution control,
3. emulate a *semantically homogenous* setup over a large number of endpoints,
4. provide a diverse and comprehensive set of queries.

The most comprehensive federated SPARQL benchmark to date is FedBench [26]. It features a mix of synthetic and real-world LoD data. In addition, it offers a set of cross-domain and domain-specific queries. While highly useful, it does not adhere to the above requirements. First concerning *points 4* and *1*, it provides only a fixed data set size, while queries do not systematically cover a defined design space. Second, *point 3* is not addressed, as it, for example has only three sources for life sciences queries. Finally, regarding *point 2*, the data distribution is not specified.

To mimic this worst-case scenario, we modified the popular LUBM [12] benchmark to generate data from a single domain: academia. Both scale and data distribution are user controllable. While there are many possible data distributions, in our evaluation we adopted *horizontal partitioning*. Highly popular, these strategies often provide an excellent tradeoff between performance and ease of use. For example, Huang et. al. [14] show substantial performance improvements by employing a partitioning scheme based on the idea that *star shaped* queries are common and therefore star shaped sub-graphs should not be split. Finally, horizontal partitioning schemes are a natural fit for federations of RDF data, as it is unlikely for triples to be randomly assigned to sites that belong to different administrative entities, but very likely for triples sharing a common provenance criteria to stay together.

Given its popularity, we adopt this partitioning scheme and choose 5 horizontal splits with increasing levels of *distribution messiness*. In the first distribution $U1$, data specific to one LUBM university is allocated to one site, similarly, distributions $U3$, $U5$ and $U7$ split the triples of each university to 3, 5 and 7 sites respectively.[13] Finally, we complemented these with distribution UH which represents the traditional horizontal split of the data based on the *subject* of a triple. While UH is not a natural fit for federated setups it offers valuable insight.

Table 1
federated LUBM queries

Query	Shape	Selectivity	Scaling
$LQ1$	LINEAR	LOW	SCALING
$LQ2$	LINEAR	HIGH	CONSTANT
$LQ3$	LINEAR	HIGH	CONSTANT
$LQ4$	LINEAR	LOW	SCALING
$LQ5$	STAR	HIGH	CONSTANT
$LQ6$	STAR	LOW	SCALING
$LQ7$	STAR	LOW	SCALING
$LQ8$	STAR	LOW	SCALING
$LQ9$	FLAKE	HIGH	CONSTANT
$LQ10$	FLAKE	LOW	CONSTANT
$LQ11$	FLAKE	HIGH	CONSTANT
$LQ12$	COMPLEX	HIGH	CONSTANT
$LQ13$	COMPLEX	HIGH	CONSTANT
$LQ14$	COMPLEX	LOW	CONSTANT

Accompanying these distributions we developed 14 SPARQL queries (cf. detailed in Appendix C and Table 1) with different shapes as specified by the Waterloo SPARQL Diversity Test Suite (WatDiv) [3].[14] In addition to shape, the queries are also split into *high* and *low* result-set selectivity given a threshold on the total number of result tuples. Furthermore, we differentiate between *constant* and *scaling* queries when their result sets stay constant or increase with total dataset size. For this evaluation we fixed the result-set selectivity threshold to **5000** tuples.

6.2. Union Operator Performance and Scaling

To ascertain how much faster the parallel tree union operator is when compared to the baseline serial union we constructed four queries each containing only a single triple-pattern: $LU1$ - $LU4$ (see Appendix B). Note that these single triple pattern queries have the advantage over $LQ1 - LQ14$ that they solely measure the impact of different kinds of unions. Specifically, we measured the time it takes to union all partial bindings spread over 100 sites, while relying on the same experimental setup detailed earlier.

As seen in Table 2, the final result-set cardinality for each of the union-queries varies between ≈ 20 and 160000 tuples. As expected, when the cardinality of the result set is low the methods fare comparably in terms of performance. For example query $LU1$ pro-

[13] We released the LUBM generator wrapper under an open source licence at `https://github.com/cosminbasca/rdftools`

[14] The queries are also available publicly at`https://github.com/cosminbasca/rdftools/blob/master/doc/DESCRIPTION.md`

Table 2
Union LUBM queries

Query	Cardinality	T_{SERIAL}	$T_{PARALLEL}$
$LU1$	19	0.26	**0.20**
$LU2$	8000	6.48	**1.02**
$LU3$	25600	7.09	**1.23**
$LU4$	160006	10.27	**1.97**

* T: time for all results (seconds)

duces only 19 result tuples, a much smaller number than the number of participating sites. In consequence, and assuming no data replication in our setup, not all endpoints can contribute to the final result. This leads to a low number of disjunctions for both operators, and hence similar performance: ≈ 0.2 seconds.

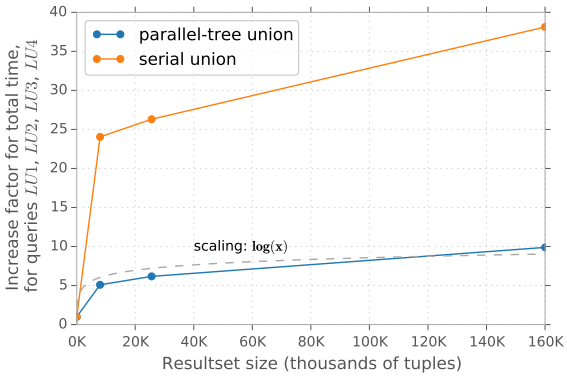

Fig. 7. Parallel tree vs serial union performance function of varying triple-pattern cardinality. Partial bindings horizontally partitioned over 100 sites.

However, when more data is involved i.e., for queries which produce more partial results, the performance difference can be dramatic. Just as expected (and graphed in Figure 7), the parallel tree union operator exhibits a scaling characteristic closely following a logarithmic performance degradation (blue versus dashed line). It is however interesting to observe that the naïve serial operator also scales better than linear when result-set cardinality increases. This can be explained by the fact that larger result-sets use the network more efficiently, by saturating bandwidth, unlike smaller result-sets which do not utilise the entire available bandwidth.

As seen, for queries $LU2$, $LU3$ and $LU4$, the parallel tree union algorithm leads to a **6.3x**, **5.7x** and **5.2x** performance boost over the naïve serial case. Even more so, such performance gains are typically cumulative, since even simpler queries may require several union operations – proportional with the number of (partitioned) triple patterns in the query. Additionally, the same general principle can be applied not only to union but to merge operations as well.

6.3. Impact of SPARQL Endpoint Caching

In order to observe the extent by which the SPARQL endpoint caching strategy (outlined in Section 5) impacts overall system performance, we measured the geometric mean over the entire benchmark, of the time spent while waiting for RDF store results. We differentiated between the two optimisers employed by X-AVALANCHE. Furthermore, we used the same experimental setup detailed before and controlled for fragmentation by setting the number of fragments $\phi = 1$.

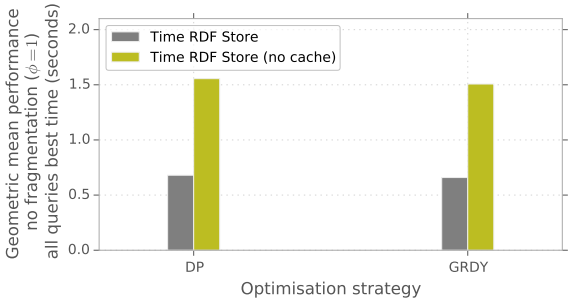

Fig. 8. SPARQL endpoint cache impact.

Figure 8 graphs the geometric mean of the SPARQL endpoint wait times incurred (i.e., the time that x-Avalanche waits for results) for all queries in the benchmark. As can be seen, caching has a significant impact on overall performance. The hit ratio varies from **52%** to **66%** with an average of 55% cache hits. The impact is significant even for high performance SPARQL endpoints, like Virtuoso v7.1 (used in this evaluation), and resulted in an $\approx$ **10%** reduction of the benchmark overall geometric mean query completion time.

Note that these results are based on the simple strategies that only cache the results of BGP and COUNT queries. More elaborate strategies are likely to have a higher impact.

6.4. System Scalability

Performance scalability is critical to any distributed DBMS query processing engine when more data is indexed. To this end, we varied the size of the generated LUBM datasets from 500 universities totalling $\approx$ 67 million triples to 8000 universities totalling $\approx$

1.1 billion triples. Naturally, the scaling characteristic of the underlying SPARQL endpoint, which X-AVALANCHE wraps, has an impact on parts of the federated engine's execution pipeline. In the worst case, if all endpoints would be powered by an RDF store that scales poorly, the maximum number of SPARQL operations that need to be executed serially, i.e., the *critical execution path*, will be the primary performance impacting factor. X-AVALANCHE mitigates the effect of a low performing RDF store to a certain degree by: *1*) caching the results of queries without VALUES bindings (Section 5) and *2*) keeping materialised views in memory for the current executing query.

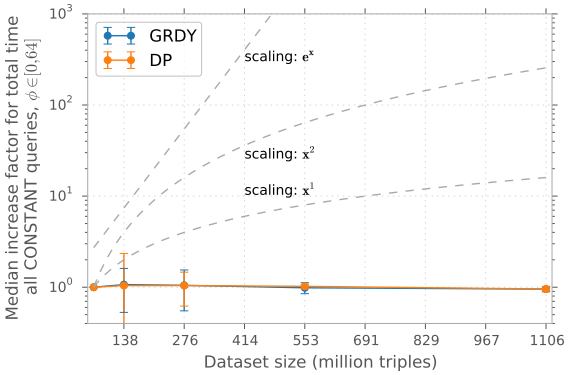

Fig. 9. Performance scaling by strategy, when dataset size increases for *constant* queries (error bars indicate standard deviation).

First, we examine how X-AVALANCHE's performance scales when processing queries whose number of results do not change when the total number of triples stored across all endpoints varies. Figure 9 graphs the median ratio between the total query time across all *constant* benchmark queries for the current dataset size and the smallest dataset: LUBM 500. As observed, X-AVALANCHE exhibits average constant scaling for queries whose number of results stay the same at all dataset sizes i.e., **is unaffected by dataset size variation for constant queries**.

Similarly, we examine how X-AVALANCHE's performance scales when dealing with queries whose number of results increase with the dataset size. In our evaluation queries from the *scaling* group (see Table 1) exhibit the same scaling factor as that of the dataset, e.g., if the dataset size doubles so does the number of results for the respective query. As can be observed in Figure 10, X-AVALANCHE's median scaling characteristic is better than the theoretical *linear scalability* threshold (bottom most dotted line in Figure labeled:

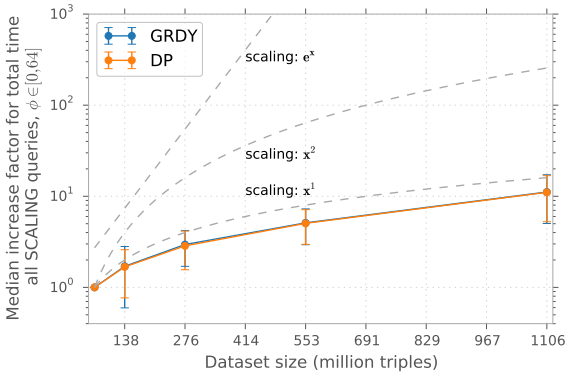

Fig. 10. Performance scaling by strategy, when dataset size increases for *scaling* queries (error bars indicate standard deviation).

scaling: x^1). While there are cases that lead to performance degradation as dataset size increases (error bars in Figure) they still follow a linear scaling characteristic as depicted. In conclusion, X-AVALANCHE **exhibits better than linear performance scaling for scaling queries**.

6.5. Data Distribution

To see how the X-AVALANCHE optimisation strategies are impacted by varying distribution messiness, we generated distributions $U1$, $U3$, $U5$, $U7$ and UH for the LUBM 8000 scaling factor, a dataset totalling more than 1.1 billion triples.

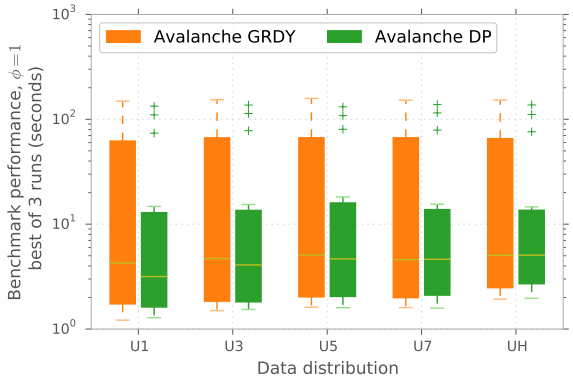

Fig. 11. Overall benchmark performance function of data distribution. The boxplot graphs the quartiles (box), the largest and smallest non-outliers (little T-shaped extensions, as well as possible outliers (crosses).

Figure 11 illustrates X-AVALANCHE's top performance distribution over all benchmark queries by optimisation strategy. We controlled for the effects of

fragmentation and disabled it by setting $\phi = 1$. Each query was run 3 times and the best run time was considered. Results show that both the greedy (GRDY) and dynamic programming (DP) optimisers exhibit a comparable average performance as a university's triples spread further from the source, i.e., distributions $U1 \longrightarrow UH$. However, as expected the optimal DP optimiser fares better in general than GRDY. The average best performance ranges in the $[3.1, 5]$ and $[4.2, 5]$ seconds intervals for the DP and GRDY respectively. An interesting observation is that while for the messiest distribution UH both show the same average performance, DP is 1 second faster on average for the less messy distribution $U1$.

Performance differences become quite visible however by the time 75% of the benchmark queries have executed. The GRDY optimiser exhibits an average performance between 62 and 66 seconds depending on distribution while the DP optimiser takes only between 12.8 and 15.9 seconds to achieve the same. The DP optimiser is on average ≈ 4.5 times faster than the GRDY approach.

In order to measure the effect of distribution variation on X-AVALANCHE, i.e., to see how robust X-AVALANCHE is to distribution change, we performed *pairwise t-tests*[15] between the obtained measurements of all distribution pairs. We use the more rigorous *three σ rule*, i.e., having a P value threshold of 0.001, to determine if the observed effect is due to distribution variation and not due to chance alone. Distribution variation has no effect on the GRDY optimiser. The P value ranges from 0.003 to 0.93. The smaller P values are obtained when comparing distribution $U1$ to any other distribution. A similar conclusion can be drawn for the DP optimiser with one exception, the effect that distribution $U1$ has on X-AVALANCHE compared to distribution UH is statistically significant with $P = 0.0004$. In conclusion we can safely say that even though performance degrades slightly, the DP optimiser is robust to distribution variation as triples are spread further from the source with the exception of the less messy distribution $U1$, where performance is best for both optimisers.

A similar trend can be seen in Figure 12 which plots the geometric performance of both strategies in addition to FedX for the entire benchmark. Again, while the distribution has a general but limited impact on performance, except for $U1$ where as expected it performs best, it does not affect the relative differences between the two optimisation strategies, with the greedy optimiser consistently performing worst.

[15] We tested for normality using the SciPy normality test, which is based on D'Agostino and Pearson.

6.6. Versus State of the Art

In order to get a better grasp of X-AVALANCHE's performance gains, this section compares its results against the top performing state of the art SPARQL federated engine, which supports *location transparency*. We chose to evaluate only against FedX, since a recent fine grained and comprehensive study [23] found that overall FedX outperformed all other state of the art SPARQL federation engines.

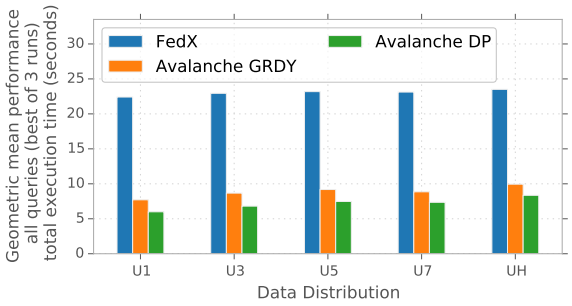

Fig. 12. Geometric benchmark performance function of data distribution, for both systems.

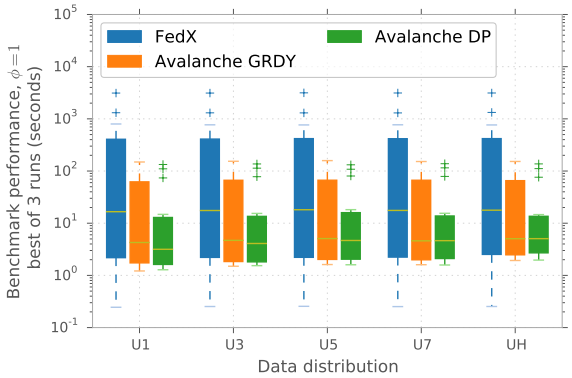

Fig. 13. Overall benchmark performance function of data distribution, for both systems. Note that y-axis is logarithmic.

As seen in Figure 12, both optimisation strategies outperform FedX in the geometric mean over the entire benchmark. Like before, we controlled for fragmentation by setting $\phi = 1$, and considered the best out of 3 runs for each query. Both federated engines perform

better for less messy distributions, where the triples are spread out on fewer or no endpoints. A detailed statistical breakdown of the performance difference between X-AVALANCHE and FedX is illustrated in Figure 13. Results clearly show that X-AVALANCHE is **more than one order of magnitude faster** than FedX over the entire benchmark. In addition it is interesting to observe that:

- the slowest X-AVALANCHE query finishes well before FedX completes the benchmark's 75^{th} percentile,
- the 75^{th} percentile of the benchmark queries are completed by the DP optimiser before FedX completes the benchmarks 50^{th} percentile, and
- most expensive non-outlier query for GRDY is comparable with the outlier queries for DP. Furthermore, DP completes the benchmark's 75^{th} percentile significantly sooner than GRDY.

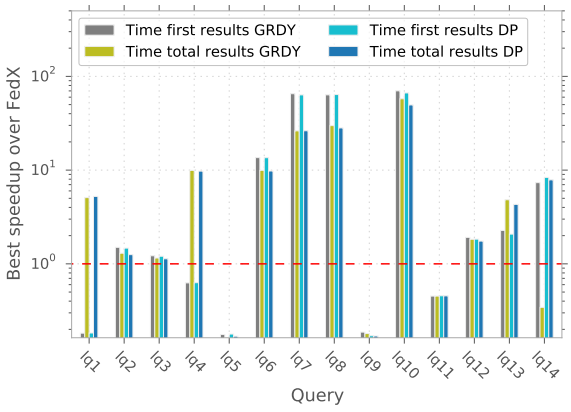

Fig. 14. Best speedup for $U7$, any configuration.

Speedup Figure 14 illustrates X-AVALANCHE's speedup over FedX, by optimiser strategy. In this setup we consider the messiest natural distribution we evaluated namely $U7$, where the triples of a university are spread to 7 other endpoints. We did not control for fragmentation and choose the best response time for per system and per query. The dotted line in Figure represents the *point of equal best performance* between the two systems.

The GRDY optimiser obtains a maximum performance **speedup factor of 70x respectively 57.7x over FedX** for first results retrieval respectively total query completion. While better performing in general the DP optimiser obtains a maximum **speedup factor of 66.5x respectively 49.5x over FedX** for first results respectively total time.

GRDY is faster than FedX for total query performance in 10 of the 14 queries while DP performs better for 11 queries. For getting first results both optimisers are faster for 9 of the 14 queries. FedX is faster in 3 respectively 4 out of the 14 queries over the DP respectively GRDY optimisers, and in 5 queries when retrieving first results. However, as seen in Table 3, the difference between the two systems for queries where FedX is faster is between 1.1 and 1.7 seconds with the notable exception of query $LQ14$ where GRDY completes in 140.5 seconds compared to 48.2 seconds for FedX. This is an expected result since the query is part of the COMPLEX group and the greedy optimiser does not guarantee optimality. The DP optimiser however, finishes query execution in 6.1 seconds, an expected conclusion. We attribute FedX's speedup over X-AVALANCHE for queries $LQ5$, $LQ9$ and $LQ11$ to the following:

a) the queries are highly selective with 7, 3, and 133 results respectively, and
b) FedX's local cache, which can greatly improve performance by discarding sources known not to contribute to the current query.

Table 3
Best query performance for each system

Query	t^{GRDY}_{AVA}	t^{DP}_{AVA}	t_{FEDX}	T^{GRDY}_{AVA}	T^{DP}_{AVA}	T_{FEDX}
lq1	2.3	2.3	0.4	**4.3**	**4.2**	**22.1**
lq2	1.5	1.5	2.2	**1.8**	**1.8**	**2.3**
lq3	1.6	1.6	1.9	**1.9**	**2.0**	**2.2**
lq4	46.7	46.5	29.3	**77.4**	**78.9**	**769.4**
lq5	1.4	1.4	0.2	**1.6**	**1.5**	**0.3**
lq6	35.7	35.7	485.0	**130.7**	**132.8**	**1299.3**
lq7	5.9	6.0	382.3	**15.5**	**15.5**	**407.5**
lq8	45.3	44.9	2873.7	**105.0**	**111.1**	**3130.8**
lq9	1.8	1.9	0.3	**1.9**	**2.0**	**0.3**
lq10	4.5	4.7	313.7	**7.3**	**8.4**	**418.7**
lq11	2.0	2.0	0.9	**2.0**	**2.0**	**0.9**
lq12	2.3	2.4	4.4	**2.3**	**2.4**	**4.2**
lq13	2.6	2.8	5.8	**2.7**	**3.0**	**13.1**
lq14	5.6	5.0	41.5	**140.5**	**6.1**	**48.2**

* t: time for first results (seconds)
+ T: time for all results (seconds)

For cases where X-AVALANCHE is faster than FedX, the performance difference ranges from near similar, e.g., 0.5 seconds, to **dramatic improvements of over 3000 seconds**, as observed for query $LQ8$ a low selectivity start shaped query with more than 70000 results.

6.7. Fragmentation

In Section 3.4 we introduced the τ performance metric as defined in equation 5. It offers a unified measure of the tradeoff between time to first results and total query execution time. Considering the τ metric, in the following, we investigate X-AVALANCHE's average benchmark performance when plan fragmentation is considered. We varied the number of fragments $\phi \in [0, 64]$ by powers of 2 increment.

In the following we investigate the average benchmark performance of X-AVALANCHE in terms of the τ performance tradeoff.

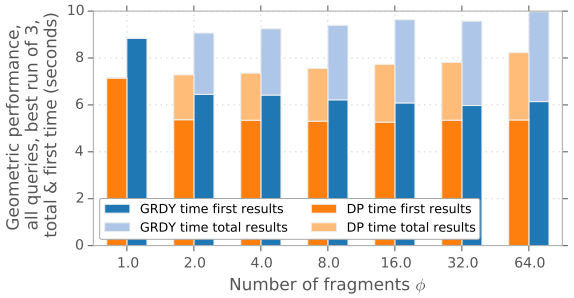

Fig. 15. Overall benchmark performance by number of fragments and optimisation strategy.

Figure 15 depicts the geometric benchmark performance split by total query completion time and time to first results for both optimisation strategies function of number of fragments ϕ. We varied $\phi \in [1, 64]$ by powers of 2 increments. All generated fragments were executed concurrently in parallel on the orchestrating node's 10 physical cores. In general we can see that fragmentation helps deliver FAST FIRST results at the cost of introducing a small penalty for overall completion time. The trend appears to be more accentuated on average for the GRDY optimiser.

A quantifiable view of the trade-off between total and first results is graphed in Figure 16. The dashed line represents τ when $\phi = 1$, equivalent to total execution time when fragmentation is disabled. Both GRDY and DP strategies benefit from fragmentation if the desired goal is to get first results fast with some penalty in increasing query execution time. For the give experimental setup, an optimal tradeoff is obtained for DP when $\phi \in [2, 4]$, while for GRDY when $\phi \in [2, 32]$. It is interesting to note that on average the greedy approach can offer a better tradeoff when fragmentation is enabled than DP with no fragmentation.

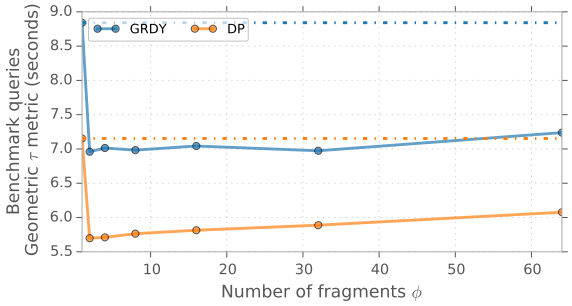

Fig. 16. Overall benchmark performance by number of fragments and optimisation strategy.

6.8. Query Shape and Selectivity

To get a clearer view of the impact of workload on performance, in the following we control for query shape and result set selectivity.

High selectivity queries They are primarily characterised by low number of results. We consider a query to be highly selective if it has ≤ 5000 results. Consequently, such queries are expected to have better execution performance, leading to the hypothesis that the impact of any optimisation is less visible than for low selectivity queries. This fact is easily observed in Figures 17 through 20, where the range of the τ metric is between 1.4 and 2.75 seconds overall.

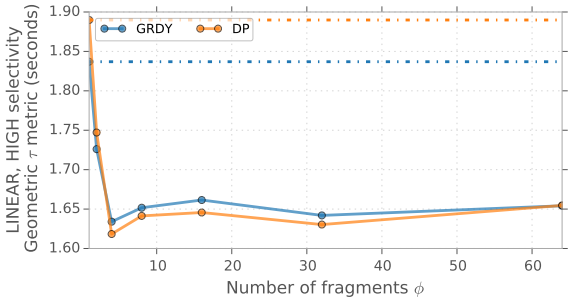

Fig. 17. High selectivity LINEAR queries

For LINEAR, STAR and COMPLEX queries there exists an optimal tradeoff for $\phi > 1$. In general the DP optimiser fares better, however, this is not the case for COMPLEX queries where GRDY offers better performance (Figure 20) although by a very small margin of 0.1 seconds on average. It is interesting to observe that for FLAKE queries (Figure 19), only the DP optimiser benefits from fragmentation with an optimal $\phi \in [2, 4]$ seconds. At the same time the GRDY optimiser shows

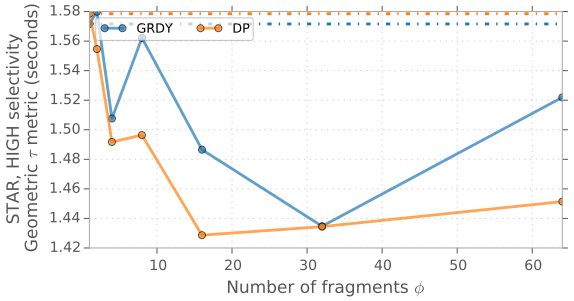

Fig. 18. High selectivity STAR queries

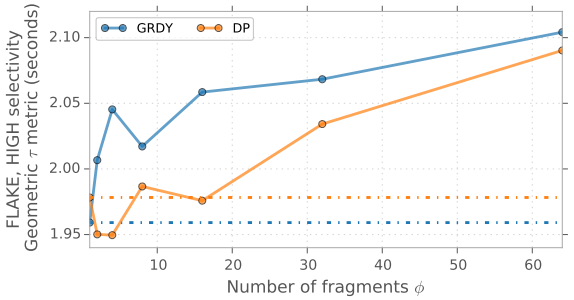

Fig. 19. High selectivity FLAKE queries

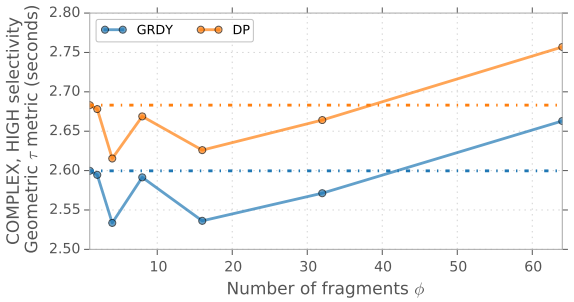

Fig. 20. High selectivity COMPLEX queries

a steady degradation characteristic although by a very small margin of 0.15 seconds on average.

Low selectivity queries Low selectivity queries are naturally more expensive since they usually produce a larger number of partial results during execution. Therefore, the effects of fragmentation on the different optimisation strategies and query shapes is more visible, as seen in Figures 21 through 24, where the range of the τ metric is between 5.5 and $\approx$ 160 seconds overall.

LINEAR shaped queries show a clear benefit (Figure 21) when fragmentation is enabled. While both

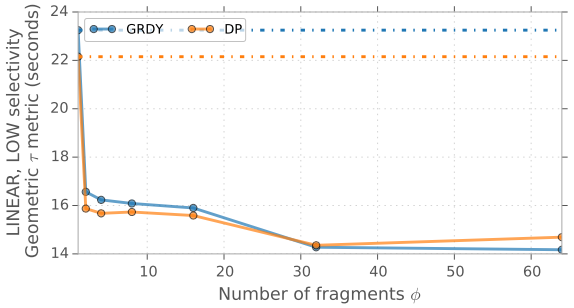

Fig. 21. Low selectivity LINEAR queries

optimisers fare similarly in performance, an optimal tradeoff is obtained when $\phi \in [32, 64]$. We believe this to be due to the fact that when fragmented this class of queries leads to less interactions between executing fragments than in other situations.

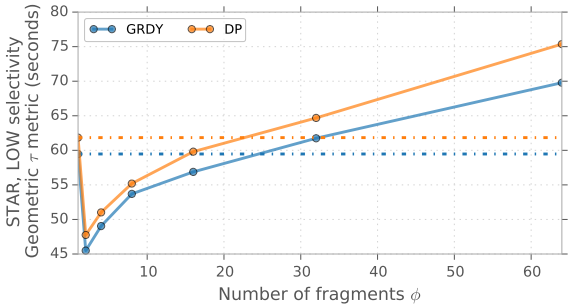

Fig. 22. Low selectivity STAR queries

For STAR shaped queries, both the GRDY and DP optimisers benefit from fragmentation with an optimum trade-off when $\phi = 2$. It is interesting to note that both strategies follow a similar performance trend with GRDY being faster by up to 5 seconds on average. In addition more than 16 fragments leads to performance degradations in our experimental setup.

Perhaps the most dramatic performance improvements are observed for FLAKE shaped queries which benefit both optimisers for any number of fragments in the chosen range. Here the greedy optimiser (GRDY) seems to benefit the most by achieving the highest performance tradeoff for $\phi = 64$. In this case the total time stays relatively stable while the time for first results drops from ca. 7.5 seconds to ca. 4.75 seconds.

For COMPLEX queries the choice of number of fragments has a positive effect on the GRDY optimiser whose time for first results drops from ca. 155 seconds to ca. 40 seconds. The DP strategy appears to be less

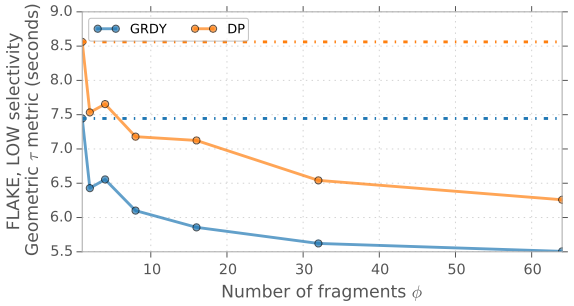

Fig. 23. Low selectivity FLAKE queries

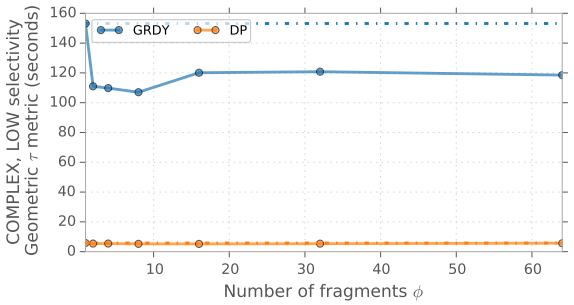

Fig. 24. Low selectivity COMPLEX queries

affected by fragmentation in this case. We attribute this to the fact that GRDY ends up choosing suboptimal plans where the effect of fragmentation is more dramatic, in contrast to DP which chooses optimal plans.

In conclusion, we can observe that in general the asynchronous GRDY and DP strategies, where each fragment is optimised either greedily or via dynamic programming in isolation and executed concurrently, do generally benefit up to a point from an increased number of fragments. The most impact can be observed for LINEAR and FLAKE low selectivity queries. We believe that this is the case due to the more flexible scheduling of resources, a direct consequence of the concurrent and asynchronous execution paradigm that X-AVALANCHE employs.

7. Limitations and Future Work

In the following we are going to detail X-AVALANCHE limitations as well as those of the system's optimisation methods and operator design. In addition, based on these limitations and findings, we will briefly discuss possible future work directions.

The work presented in this paper exhibits two kinds of limitations. First, X-AVALANCHE could be extended and/or optimised further and second, the external validity of our empirical evaluation is limited.

A first limitation stems from the fact that when employing *plan fragmentation* to derive an optimal trade-off between total query execution time and time to first results the choice between parametric and non-parametric space reduction algorithms is not automatic. Potential future work directions could include automatic learning of the parameters. This would entail learning which method to use and what is a good prior or number of fragments using of methods such as *Bayesian Optimisation* [28] or self-tuning database methods [9].

Additional limitations stem from the *mismatch* between real and predicted plan performance. Traditional query optimisation algorithms like bottom up DP approaches assume that the cost model is optimal. In reality, plan cost estimations vary widely from their true cost. Consequently, fragmentation derived performance gains are diminished and depend on the estimative power of the cost model. Improving the cost model's accuracy will allow X-AVALANCHE to make better optimisation decisions and improve performance.

To further improve X-AVALANCHE's performance a number of research avenues and potential solutions stand out. While X-AVALANCHE extends and enhances the distributed state management protocol of AVALANCHE, it does not address all sources of limitation. One such limitation is derived from the level of impact that low performing SPARQL endpoints have on the system. While this is addressed to a certain degree by caching of result-sets, X-AVALANCHE does not cache SPARQL queries with VALUES bindings. A future extension could entail investigating how to use bloom filters [8] to reduce the number of bindings sent to remote endpoints and therefore remote workload. Furthermore, X-AVALANCHE union operator is not optimised to take duplicates into account. On the Web of Data records are duplicated leading to a more optimisation possibilities by investigating the applicability of bloom filters to these cases or employing methods similar to the ones described in [24].

Finally, the experimental setup relies on a limited number of physical resources. A physical machine is typically tasked with accommodating more than a dozen SPARQL and X-AVALANCHE endpoints. The resulting resource contention, generated by the competition for shared resources such as RAM, disk & network I/O, and CPU-time, can have a negative impact

on measured system performance, a fact that can be mitigated by the choice of physical machines.

8. Conclusions

To conclude, in this paper we present an extension of our original AVALANCHE SPARQL federation engine, which we call X-AVALANCHE. First, we introduce support for disjunctions when data is partitioned, by employing a novel parallel union algorithm called: *parallel tree union*. Results show that the parallel algorithm is able to perform up to **5x** faster than a naïve serial one. Second, we enhanced the distributed state management specific to our federated SPARQL protocol. To this end, each X-AVALANCHE operator is enhanced with support for execution by proxy allowing for orchestration effort offloading to other participating endpoints. In addition, we make use of parallel multicast bind-joins to minimise network traffic. At the same time we remotely cache query results (given allotted memory for cache) when no VALUES bindings are present in the query. This strategy alone, reduced overall query processing time by $\approx 10\%$.

Furthermore, we introduce a first novel approach to optimally reduce and traverse the *extended planning space*, that is suitable for large federations of RDF stores. We identify a new class of easily parallelizable plans we call *fragmented bushy plans* and we show how to optimally find the largest partial results set retrievable in the shortest possible time given external constraints. We implement and compare two exemplars of the non-parametric and parametric optimal extended planning space reduction methods: *bayesian-blocks* and *k-segmentation* and conclude in favour of the parametric approach, given its intrinsic control of the number of fragments. Finally, to support our hypothesis we also introduced a new synthetic benchmark designed with the difficult case of large homogenous RDF federations in mind. Released as open-source, the benchmark relies on LUBM to generate the data, which is than distributed to a given number of sites based on a user specified distribution. In addition, it borrows from the Waterloo SPARQL Diversity Test Suite (WatDiv) for query design.

Combined, X-AVALANCHE's enhancements and optimisations can lead to dramatic performance improvements over one of the top federated SPARQL engines to date: FedX – as shown in [23]. While, in the best case, X-AVALANCHE is up to **70x** times faster, on average, the system exhibits **more than one order of magnitude performance improvements** for total query execution time.

In summary, x-Avalanche shows that federated SPARQL processing can still be substantially improved both by focusing on low-level elements such as operator design and high-level system architecture considerations. As such we believe that the insight we gained from x-Avalanche provide an important building block for building the Web of Data.

9. Acknowledgments

This work was partially supported by the Swiss National Science Foundation under contract number 200021-118000. We would also like to thank Timo Mennle and Ioana Giurgiu for their invaluable feedback in improving this paper.

References

[1] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: An adaptive query processing engine for SPARQL endpoints. In *The Semantic Web – ISWC 2011*, pages 18–34. Springer Science + Business Media, 2011.

[2] K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. Describing Linked Datasets - On the Design and Usage of voiD, the 'Vocabulary of Interlinked Datasets'. In *WWW 2009 Workshop: Linked Data on the Web (LDOW2009)*, Madrid, Spain, 2009.

[3] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of RDF data management systems. In *The Semantic Web – ISWC 2014*, pages 197–212. Springer Science + Business Media, 2014.

[4] M. M. Astrahan, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, V. Watson, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, and P. R. McJones. System r: relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, jun 1976.

[5] C. Başca and A. Bernstein. Avalanche: putting the spirit of the web back into semantic web querying. In *Proceedings Of The 6th International Workshop On Scalable Semantic Web Knowledge Base Systems (SSWS2010)*, pages 64–79, November 2010.

[6] C. Başca and A. Bernstein. Querying a messy web of data with avalanche. *Web Semantics: Science, Services and Agents on the World Wide Web*, 26:1–28, may 2014.

[7] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[8] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003.

[9] S. Chaudhuri and V. Narasayya. Self-tuning database systems: A decade of progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 3–14. VLDB Endowment, 2007.

[10] R. Cyganiak. A relational algebra for SPARQL. Technical Report HPL-2005-170, HP Laboratories, Sep 2005.
[11] O. Görlitz and S. Staab. SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In *Proceedings of the Second International Workshop on Consuming Linked Data (COLD2011), Bonn, Germany, October 23, 2011*, 2011.
[12] 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, oct 2005.
[13] H. Herodotou, N. Borisov, and S. Babu. Query optimization techniques for partitioned tables. In *Proceedings of the 2011 international conference on Management of data - SIGMOD '11*. Association for Computing Machinery (ACM), 2011.
[14] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *Proceedings of the VLDB Endowment*, 4(11):1123–1134, 2011.
[15] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. *ACM SIGMOD Record*, 20(2):268–277, apr 1991.
[16] D. Kossmann and K. Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. Database Syst.*, 25(1):43–82, mar 2000.
[17] S. Lynden, I. Kojima, A. Matono, and Y. Tanimura. ADERIS: An adaptive query processor for joining federated SPARQL endpoints. In *Lecture Notes in Computer Science*, pages 808–817. Springer Science + Business Media, 2011.
[18] T. Neumann and G. Weikum. x-RDF-3x. *Proceedings of the VLDB Endowment*, 3(1-2):256–263, sep 2010.
[19] A. Nikolov, A. Schwarte, and C. Hütter. FedSearch: Efficiently combining structured queries and full-text search in a SPARQL federation. In *The Semantic Web – ISWC 2013*, pages 427–443. Springer Science + Business Media, 2013.
[20] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 314–325. Morgan Kaufmann, 1990.
[21] E. S. PAGE. A test for a change in a parameter occurring at an unknown point. *Biometrika*, 42(3-4):523–527, 1955.
[22] B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. In *The Semantic Web: Research and Applications*, pages 524–538. Springer Science + Business Media, 2008.
[23] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, and A.-C. Ngonga Ngomo. A fine-grained evaluation of SPARQL endpoint federation systems. *Semantic Web Journal*, 2014.
[24] M. Saleem, A.-C. N. Ngomo, J. X. Parreira, H. F. Deus, and M. Hauswirth. DAW: Duplicate-AWare federated query processing over the web of data. In *The Semantic Web – ISWC 2013*, pages 574–590. Springer Science + Business Media, 2013.
[25] J. D. Scargle, J. P. Norris, B. Jackson, and J. Chiang. STUDIES IN ASTRONOMICAL TIME SERIES ANALYSIS. VI. BAYESIAN BLOCK REPRESENTATIONS. *ApJ*, 764(2):167, feb 2013.
[26] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. FedBench: A benchmark suite for federated semantic data query processing. In *The Semantic Web – ISWC 2011*, pages 585–600. Springer Science + Business Media, 2011.
[27] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization techniques for federated query processing on linked data. In *The Semantic Web – ISWC 2011*, pages 601–616. Springer Science + Business Media, 2011.
[28] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012.
[29] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceeding of the 17th international conference on World Wide Web - WWW '08*. Association for Computing Machinery (ACM), 2008.
[30] X. Wang, T. Tiropanis, and H. C. Davis. LHD: optimising linked data query processing using parallelisation. In *Proceedings of the WWW2013 Workshop on Linked Data on the Web, Rio de Janeiro, Brazil, 14 May, 2013*, 2013.
[31] C. Weiss, P. Karras, and A. Bernstein. Hexastore. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, aug 2008.

Appendix

For brevity the prefix declaration for LUBM[16] was omitted.

A. Detailed Results and Statistics

The average time to retrieve the cardinality of a triple-pattern was **0.246** seconds with $\sigma = 0.01$ seconds.

B. Union Benchmark Queries

```
SELECT * WHERE{
?student lubm:takesCourse <http://www.Department12.
      University1.edu/Course1> }
```

Listing 1: LU1

```
SELECT * WHERE{
?department lubm:name "Department1" }
```

Listing 2: LU2

```
SELECT * WHERE{
?student lubm:undergraduateDegreeFrom <http://www.
      University0.edu> }
```

Listing 3: LU3

[16] www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#

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

Listing 4: LU4

C. Benchmark Queries

C.1. Linear Queries

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

Listing 5: LQ1

```
SELECT * WHERE {
?department lubm:subOrganizationOf ?university .
?professor lubm:worksFor ?department .
?student lubm:advisor ?professor .
?student lubm:memberOf <http://www.Department1.University0.
      edu> .}
```

Listing 6: LQ2

```
SELECT * WHERE {
?resgroup lubm:subOrganizationOf ?department .
?professor lubm:worksFor ?department .
?student lubm:advisor ?professor .
?student lubm:memberOf <http://www.Department1.University0.
      edu> .}
```

Listing 7: LQ3

```
SELECT * WHERE {
?advisor lubm:emailAddress ?email .
?advisor lubm:worksFor ?department .
?department lubm:name "Department1" .}
```

Listing 8: LQ4

C.2. Star Queries

```
SELECT * WHERE {
?student lubm:advisor ?advisor .
?student lubm:name ?name .
?student lubm:undergraduateDegreeFrom ?university .
?student lubm:takesCourse <http://www.Department1.
      University0.edu/GraduateCourse33> .}
```

Listing 9: LQ5

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

Listing 10: LQ6

```
SELECT * WHERE {
?student lubm:memberOf ?department .
?student lubm:takesCourse ?course .
?student lubm:advisor ?advisor .
?student lubm:teachingAssistantOf ?tacourse .
?student lubm:emailAddress ?email .
?student lubm:name ?name .
?student lubm:telephone ?telephone .
?student lubm:undergraduateDegreeFrom <http://www.
      University0.edu> .}
```

Listing 11: LQ7

```
SELECT * WHERE {
?student lubm:memberOf ?department .
?student lubm:takesCourse ?course .
?student lubm:advisor ?advisor .
?student lubm:teachingAssistantOf ?tacourse .
?student lubm:emailAddress ?email .
?student lubm:name "GraduateStudent71" .
?student lubm:telephone ?telephone .
?student lubm:undergraduateDegreeFrom ?university .}
```

Listing 12: LQ8

C.3. Snow Flake Queries

```
SELECT * WHERE {
?student lubm:advisor ?advisor .
?advisor lubm:worksFor ?department .
?department lubm:subOrganizationOf ?university .
?student lubm:name ?name .
?student lubm:telephone ?tel .
?student lubm:takesCourse <http://www.Department12.
      University1.edu/Course1> .}
```

Listing 13: LQ9

```
SELECT * WHERE {
?department lubm:name ?name .
?resgroup lubm:subOrganizationOf ?department .
?department lubm:subOrganizationOf <http://www.University0.
      edu> .
?student lubm:memberOf ?department .
?student lubm:advisor ?professor .
?student lubm:takesCourse ?course .}
```

Listing 14: LQ10

```
SELECT * WHERE {
?department lubm:name ?name .
?resgroup lubm:subOrganizationOf ?department .
?department lubm:subOrganizationOf ?university .
?student lubm:memberOf ?department .
?student lubm:advisor ?professor .
?student lubm:takesCourse <http://www.Department1.
      University0.edu/GraduateCourse33> .}
```

Listing 15: LQ11

C.4. Complex Queries

```
SELECT * WHERE {
?department lubm:subOrganizationOf ?university .
?resgroup lubm:subOrganizationOf ?department .
?student lubm:memberOf ?department .
?department lubm:name ?name .
?student lubm:advisor ?professor .
?publication lubm:publicationAuthor ?professor .
?publication lubm:publicationAuthor <http://www.Department1.
      University10.edu/AssociateProfessor1>.}
```

Listing 16: LQ12

```
SELECT * WHERE {
?department lubm:subOrganizationOf ?university .
?resgroup lubm:subOrganizationOf ?department .
?student lubm:memberOf ?department .
?student lubm:advisor ?professor .
?student lubm:takesCourse ?course .
?publication lubm:publicationAuthor ?professor .
?publication lubm:publicationAuthor <http://www.Department1.
      University10.edu/AssociateProfessor1> .
?publication lubm:name ?title .}
```

Listing 17: LQ13

```
SELECT * WHERE {
?student lubm:advisor ?advisor .
?advisor lubm:worksFor ?department .
?department lubm:subOrganizationOf <http://www.University0.
      edu> .
?head lubm:headOf ?department .
?head lubm:emailAddress ?email .
?head lubm:doctoralDegreeFrom ?alma .
?student lubm:name ?name .
?student lubm:telephone ?tel .
?student lubm:takesCourse ?course .}
```

Listing 18: LQ14