Processing Aggregate Queries in a Federation of SPARQL Endpoints

Dilshod Ibragimov[1,2], Katja Hose[2], Torben Bach Pedersen[2], and Esteban Zimányi[1]

[1] Université Libre de Bruxelles, Brussels, Belgium
{dibragim|ezimanyi}@ulb.ac.be
[2] Aalborg University, Aalborg, Denmark
{diib|khose|tbp}@cs.aau.dk

Abstract. More and more RDF data is exposed on the Web via SPARQL endpoints. With the recent SPARQL 1.1 standard, these datasets can be queried in novel and more powerful ways, e.g., complex analysis tasks involving grouping and aggregation, and even data from multiple SPARQL endpoints, can now be formulated in a single query. This enables Business Intelligence applications that access data from federated web sources and can combine it with local data. However, as both aggregate and federated queries have become available only recently, state-of-the-art systems lack sophisticated optimization techniques that facilitate efficient execution of such queries over large datasets. To overcome these shortcomings, we propose a set of query processing strategies and the associated Cost-based Optimizer for Distributed Aggregate queries (CoDA) for executing aggregate SPARQL queries over federations of SPARQL endpoints. Our comprehensive experiments show that CoDA significantly improves performance over current state-of-the-art systems.

1 Introduction

In recent years, we have witnessed the growing popularity of the Semantic Web and the Open Data movement. Nowadays a plethora of data is available in RDF format, published as Linked Open Data [6], accessible free of charge, and often queryable via SPARQL endpoints. Using these data in combination with the SPARQL 1.1 standard [24], organizations can build novel and powerful analytics applications that integrate their private data with web RDF datasets, enabling analyses that were not possible before. For example, a company wants to analyze its revenue in different countries against macro-economic indicators of these countries. Such information is unavailable locally, but can instead be obtained from the World Bank (`http://www.worldbank.org/`), accessed as Linked Open Data (`http://worldbank.270a.info/`) and queried via a SPARQL endpoint. Thus, the company has efficient access to up-to-date information without the costs of local maintenance, and as the company is accessing Linked Data, more information (geographical, census, etc.) for further analyses can efficiently be retrieved from linked sources, such as GeoNames [22] and DBpedia [4]. Such analytical queries, however, are based on complex queries involving grouping and aggregation as well as subqueries that need to be evaluated at remote sources. Formulating this in a single SPARQL statement has only recently become possible with the SPARQL 1.1 standard, which supports grouping, aggregation, and SERVICE subqueries.

Motivating example. Analytical queries are not only beneficial for companies, but also in other scenarios. In March 2011, an earthquake in the Pacific triggered a powerful tsunami and led to a huge devastation at the Japanese coast, which eventually caused a nuclear accident (`http://goo.gl/AcqLpe`). After these events, the Japanese government made daily announcements of radioactivity statistics observed hourly at 47 prefectures. These observations from March 16, 2011 to March 15, 2012 were converted to RDF data by Masahide Kanzaki and made publicly available via a SPARQL endpoint (`http://www.kanzaki.com/works/2011/stat/ra/`). An example observation in RDF format is given below.

```
#observation
<http://www.kanzaki.com/works/2011/
    stat/ra/20110414/p13/t08>
  rdf:value "0.079"^^ms:microsv ;
  ev:place <http://sws.geonames.org/
    1852083/> ;
  ev:time <http://www.kanzaki.com/
    works/2011/stat/dim/d/
    20110414T08PT1H> ;
  scv:dataset <http://www.kanzaki.com/
    works/2011/stat/ra/set/moe> .
#dimension - place
<http://sws.geonames.org/1852083/>
  vcard:region "Tokyo"@en ;
  vcard:locality "Shinjuku"@en ;
  gn:lat "35.69355" ;
  gn:long "139.70352" .
#dimension - time
<http://www.kanzaki.com/works/2011/stat
    /dim/d/20110414T08PT1H>
  rdfs:label "2011-04-14T08";
  tl:at "2011-04-14T08:00:00+09:00"
    ^^xsd:dateTime ;
  tl:duration "PT1H"^^xsd:duration .
```

The places that the observations were recorded at are represented by a URI from GeoNames. With the observations of radioactivity in multiple geographical locations (cities in our case) and information about their upper administrative divisions (prefectures in Japan) retrievable from GeoNames, interesting analyses become possible. For instance, we can compute the average radioactivity separately for each prefecture in Japan to find out which prefectures were more affected than others. Or we can compute the minimum and maximum radioactivity for each prefecture and hence identify the changes in radioactivity over the one-year observations. Formulating such queries involves grouping and aggregation as well as combining information from two SPARQL endpoints. Listing 1.1 shows an example query that computes the average radioactivity for all prefectures in Japan. This query could be executed at a triple store with information about radioactivity and uses the LOD Cloud Cache SPARQL endpoint (`http://lod2.openlinksw.com/sparql`) to query GeoNames data remotely.

```
SELECT ?regName (AVG(?radioValue) AS ?average)
WHERE { ?s ev:place ?placeID; ev:time ?time; rdf:value ?radioValue .
  SERVICE <http://lod2.openlinksw.com/sparql> {
    ?placeID gn:parentFeature ?regionID . ?regionID gn:name ?regName . }
} GROUP BY ?regName
```

Listing 1.1: Aggregate Query over Radioactivity Observations

This looks like a simple query but current state-of-the-art triple stores supporting SPARQL 1.1, such as Virtuoso v07.10.3207, Sesame v2.7.11, and Jena Fuseki v1.0.0 (based on ARQ) timed out while trying to answer this query. Inspecting a query execution plan was not possible for Virtuoso, Jena, and Sesame since they do not support a comfortable explain function for SPARQL queries as known from relational database systems, so we used Wireshark (`http://www.wireshark.org`) to analyze the network traffic. We found out that Virtuoso and Fuseki query the GeoNames endpoint for every single radioactivity observation, while Sesame is trying to download all triples that match the pattern from the remote endpoint. In the first case, a triple store needs to

send more than 400,000 requests to answer the query, and in the second case it needs to download more than 7.8 million triples from GeoNames.

The strategies implemented by these state-of-the-art triple stores are obviously insufficient in the scenario we consider in this paper. As the SPARQL 1.1 standard is not yet completely supported by all SPARQL endpoints [9], there is only little research regarding the evaluation of queries involving aggregation and grouping. To the best of our knowledge, this is the first paper to investigate aggregate queries in the context of federations of SPARQL endpoints and their optimization. In summary, the contributions of this paper are:

- the Mediator Join, SemiJoin, and Partial Aggregation query processing strategies for this scenario
- a cost model and techniques for estimating constants and result sizes for triple patterns, joins, grouping and aggregation
- the combination of these with the processing strategies into the Cost-based Optimizer for Distributed Aggregate queries (CoDA) approach for aggregate queries in federated setups that is generally able to choose the best execution strategy among a number of alternatives
- a comprehensive experimental evaluation showing that CoDA is efficient, scalable, and robust over different scenarios, and significantly faster than state-of-the-art triple stores

The remainder of the paper is structured as follows. Related work is discussed in Section 2. Section 3 identifies several alternative strategies for processing aggregated SPARQL queries in a federated setup. Section 4 introduces a cost-based query optimizer for aggregate queries over federations of SPARQL endpoints. The results of our evaluation are presented in Section 5; Section 6 concludes the paper.

2 Related Work

Federated query processing in database management systems (DBMS) has been a topic of research for several decades. In contrast to well-structured classic data models, federated RDF systems support arbitrary RDF datasets (even without explicit schema) and allow the use of special constructs to perform joins and express bindings (such as VALUES) not present in SQL-based systems.

The literature proposes a number of approaches for querying federated RDF sources. Some of these approaches require the availability of VoID [23] statistics. SPLENDID [15], for instance, uses VoID statistics to select a query execution plan for a federated query. For triple patterns not covered in the VoID statistics, the system requests the information by issuing SPARQL ASK queries. The system makes use of a cost-based model and cardinality estimations for selecting a query plan. However, the SPLENDID system and its cost-model do not cover the combination of grouping, aggregation, and SERVICE subqueries.

FedX [20] uses SPARQL ASK queries for triple patterns in a query to collect basic information that can be used for source selection. It implements bound joins with SPARQL UNION keyword (similar to a semi-join) to group triple patterns related to one source and, thus, reduces the number of queries that are sent. FedX has originally

been developed based on the SPARQL 1.0 standard and does not use cost-based query optimization. Hence, it does not provide any particular optimization techniques for our use case and would always use a semi-join based strategy, which is only one of the options our optimizer (CoDA) chooses from.

ANAPSID [1] uses a catalog of endpoint descriptions to decompose a user query into subqueries that can be executed by separate endpoints. The query engine implements a technique based on the symmetric hash join [12] and the XJoin [21] to execute subqueries in a non-blocking fashion. SIHJoin [18] also uses a hash join implementation to enable pipelining in combination with a lightweight cost-model with weight factors calibrated for remote systems. Both approaches were not designed with regard to aggregate queries and use a hash join implementation so that results from a join can already be forwarded to other operators in the query execution tree. However, pipelining is not helpful for analytical queries since the complete result of the query is needed for the aggregation.

Avalanche [5] and WoDQA [2], on the other hand, do not maintain data source registrations. Avalanche depends on third parties such as search engines to find a proper data source for executing a query. Statistics about cardinalities and data distributions are considered for breaking a query into a set of subqueries that in combination provide a full query answer. Then, these subqueries are executed in parallel against several endpoints. WoDQA uses VoID directories such as CKAN (`http://ckan.net`) and VoIDStore (`http://void.rbkexplorer.com`) to find possible sources of data. The system uses VoID statistics to group triple patterns into subqueries in a federated form and executes it by Jena ARQ.

An RDF data processing system that supports simple transactional queries as well as complex analytical queries is proposed in [25]. Aggregate queries are efficiently resolved by the system by using special look-up mechanisms. However, the system does not consider aggregate queries in a federated environment.

SPARQL-DQP [7] on the other hand, discusses semantics of the SPARQL 1.1 federation extension on a theoretical level and introduces the notion of well-defined patterns. It focuses on the optimization of federated queries in the presence of OPTIONAL subqueries but it was not designed to optimize and support analytical queries. Different strategies to implement federated queries in SPARQL 1.1 are discussed in [10]. Several limitations that may cause incorrect results and the potential validity restrictions are identified and fixes are proposed.

In summary, only very few approaches consider analytical queries [7, 25] but not in the context of a federated setup. Most state-of-the-art approaches for federated query processing are designed with a focus on SPARQL 1.0 [1,2,5,15,20] and lack full support of the more recent SPARQL 1.1 standard or do not offer support or particular optimizations for analytical queries. In contrast, this paper proposes a cost-based approach to optimize and execute aggregate SPARQL queries over federations of endpoints.

3 Federated Processing of Aggregate Queries

In this section, we will systematically outline several strategies that can be used to evaluate aggregated queries in federations of SPARQL endpoints. Section 4 will then introduce a cost-based approach to choose the best strategy for a query.

For ease of presentation, this section focuses on queries with a single `SERVICE` subquery. But the discussed principles can be extended to the general case of well-designed patterns with strongly bound variables [8]. The proposed approach can be combined with rule-based rewriting so that subpatterns, and especially joins, are evaluated in a cost-minimizing order. If an endpoint imposes limits on result sizes, then additional techniques, such as pagination [10], are used.

In the following, we use P_{AGG} to represent the original user query and P_e denotes the SERVICE subquery evaluated at SPARQL endpoint e. P_M represents the subquery that is created from the original query P_{AGG} by extracting P_e, adding a join on their common variables $var(P_e) \cap var(P_M)$, and, depending on the strategy, preserving grouping and aggregation. P_M is evaluated on the same endpoint M that P_{AGG} was sent to. Note that this section focuses on the implementation of the joins combining the partial results of the subqueries evaluated by remote endpoints. We do not make any restrictions on the local implementations that the remote endpoints use to evaluate joins contained in the subqueries they receive.

Mediator Join Strategy (MedJoin). The first strategy we describe is based on the mediator join technique that is used by many approaches for federated SPARQL query processing. The mediator/federator is the SPARQL engine that receives a query P_{AGG} from the user. The query optimizer at the mediator M defines P_e and P_M and sends P_e to endpoint e whereas P_M is processed on the endpoint m. Parallelization can be exploited by processing P_M and P_e at the same time. The main principle is to find all solutions to P_e and P_M first and then compute the remaining operations at the mediator, including the join (on `?placeID` in the example below) that combines the partial results as well as grouping and aggregation. Listings 1.2 and 1.3 illustrate P_M and P_e for our running example query (Listing 1.1).

```
SELECT ?placeID ?radioValue WHERE {
  ?s ev:place ?placeID; ev:time ?time.
  ?s rdf:value ?radioValue.
}
```

Listing 1.2: MedJoin: Query P_M

```
SELECT ?placeID ?regName WHERE {
  ?placeID gn:parentFeature ?regionID.
  ?regionID gn:name ?regName.
}
```

Listing 1.3: MedJoin: Query P_e

Note that due to the fact that SPARQL does not remove duplicate results, we do not need to keep all variables in the select clauses of P_e and P_M. If duplicates were removed (like in SQL), we would have to keep all variables in the subqueries to ensure that the number of tuples that form the result are preserved, otherwise the average function in our example query would not return the correct result.

In principle, constructs such as `OPTIONAL` and `FILTER` are assigned to the subqueries that their variables refer to. If there is a complex expression, e.g., a `FILTER` is defined on a condition involving variables from different subqueries (e.g., $?a < ?b$), then the `FILTER` is evaluated after the partial results are combined at the mediator. The strength of this strategy is that partial queries can be evaluated in parallel. However, it can easily become expensive if the intermediate results are very large or when the datasets are very big.

Semi Join Strategy (SemiJoin). This strategy is based on the bound join or semi-join technique [14,20], which was already available based on `UNION` or `FILTER` constructs in SPARQL 1.0. The recent SPARQL 1.1 standard, however, supports the `VALUES`

clause, which allows for a much more elegant solution.

The main principle of this strategy is to execute the subquery with the smallest result first and use the retrieved results as bindings for the join variables in the other subquery. The intuition is that for selective joins, sending a few partial results to an endpoint is much faster than receiving the complete result for the more general subquery. It is then the task of the cost optimizer to identify the most promising order of execution of subqueries. Constructs, such as `FILTER` and `OPTIONAL`, can be assigned to subqueries as discussed for MedJoin. Let us consider an example query with a `FILTER`.

```
SELECT ?regName (AVG(?radioValue) AS ?average) WHERE {
  ?s ev:place ?placeID . ?s ev:time ?time . ?s rdf:value ?radioValue .
  SERVICE <http://lod2.openlinksw.com/sparql>{
    ?placeID gn:parentFeature ?regionID . ?regionID gn:name ?regName .
  } FILTER(?radioValue < 0.08) . } GROUP BY ?regName
```

This query can be evaluated efficiently by evaluating query P_M (Listing 1.4) and then using the obtained bindings for the join variable `?placeID` in the `VALUES` clause of the query P_e (Listing 1.5).

```
SELECT ?placeID ?radioVal
WHERE {
  ?s rdf:value ?radioVal ;
  ev:place ?placeID; ev:time ?time.
  FILTER (?radioValue < 0.08) . }
```

Listing 1.4: SemiJoin: Query P_M

```
SELECT ?placeID ?regName
WHERE { ?placeID gn:parentFeature ?rgID.
  ?rgID gn:name ?regName.
  VALUES (?placeID) {
  <http://sws.geonames.org/1852083/>...} }
```

Listing 1.5: SemiJoin: Query P_e

In contrast to MedJoin, this strategy evaluates the subqueries sequentially and is particularly efficient for selective joins. However, as the `VALUES` clause is not yet widely supported by existing endpoints [9], the SPARQL 1.0 compliant alternatives of `UNION` (or `FILTER`) must often be used.

Partial Aggregation Strategy (PartialAgg). For queries where the grouping attributes of the original query contain a subset of the variables of the subquery that is executed first and the aggregate values are contained in the subquery that is evaluated second, further optimization is possible. The Partial Aggregation Strategy (PartialAgg) builds upon MedJoin by extending the subquery executed second with a `GROUP BY` clause and aggregate functions. The goal is to reduce the size of the partial result and compute partial aggregate values early so that P_{AGG} can be evaluated more efficiently.

Using PartialAgg our running example query (Listing 1.1) is decomposed into P_M (below) and P_e (Listing 1.5). First, P_M is computed, the result bindings are fed into the `VALUES` clause of P_e, and P_{AGG} combines the partial results via a join and computes final grouping and aggregation.

```
SELECT ?placeID (SUM(?radioValue) AS ?sum) (COUNT(?radioValue) AS ?count)
WHERE { ?s ev:place ?placeID; ev:time ?time; rdf:value ?radioValue . }
GROUP BY ?placeID
```

Note that P_M here groups by `?placeID` whereas the original query (Listing 1.1) groups by `?regName`, this is because P_M uses the join attributes $var(P_e) \cap var(P_M)$ in the `GROUP BY` clause. Whereas a particular placeID would occur in many results for P_M in the MedJoin strategy, the additional grouping here guarantees that the result set contains only one. Hence, the size of the intermediate result is reduced.

When performing such an optimization, however, we need to take into account whether the aggregate function in the original query is algebraic or distributive [16].

Computing aggregates for distributive functions (SUM, MIN, MAX, COUNT) is straightforward, while for computing AVG we first need to compute both SUM and COUNT in separate and in the final step divide the sum of all intermediate SUMs by the sum of all intermediate COUNTs, i.e., $AVG = \frac{\sum_{i=1}^{N} SUM_i}{\sum_{i=1}^{N} COUNT_i}$.

4 Cost-Based Query Optimization

For each user query, the query optimizer needs to decide which of the strategies that we discussed in the previous section to use. In this section, we present CoDA (**Co**st-based Optimizer for **D**istributed **A**ggregate Queries). A cost-based optimizer, finds the best strategy by computing query execution costs for different alternative query execution plans and choosing the one with minimum costs. In the remainder of this section, we first sketch how the query optimizer works, then we introduce the cost model. Finally, we present details regarding cardinality estimation and processing costs.

Query Optimizer. To find the best query execution plan, we need to systematically examine alternative query execution plans that produce the same result. We first decompose the original query into multiple subqueries as described in Section 3. We obtain a query P_M and endpoint queries $P_{e_1}, \ldots, P_{e_n}$. We then optimize the subqueries in separate, e.g., reordering the triple patterns based on a cost model so that the execution costs are minimized. Afterwards, we enumerate all possible plans that combine these subqueries using the strategies introduced in Section 3. For each of these alternative plans, we estimate execution costs (as described in the remainder of this section) and choose the plan with the minimum costs for query execution.

4.1 Cost Model

The overall costs of a distributed query execution plan (C_Q) consist of the costs for communication between endpoints and mediator (C_C) and the costs for processing the query on the data endpoint (C_P), i.e.: $C_Q = C_P + C_C$. To simplify the cost model, we estimate the costs for all subqueries in the same way. By calibrating the cost factors for each involved endpoint separately, the cost model can consider different sytem characteristics and estimate subqueries at the mediator and remote subqueries alike.

The cost model estimates C_Q for each subquery in separate and computes the costs of the complete query plan by combining the costs of its subqueries with the additional operators in P_{AGG} that compute the final result. For subqueries that are executed in parallel, as for the MedJoin strategy, the cost model needs to consider parallel execution. As the subquery that takes the longest determines the time when the result is available, we take the maximum time of these parallel subqueries, e.g., $C_Q(S_1, S_2) = \max(C_Q(S_1), C_Q(S_2))$, where $C_Q(S_i)$ denotes the costs of subquery S_i.

The communication costs C_C for a subquery S_i are estimated as: $C_C(S_i) = C_O + c_{S_i} \cdot C_{map}$, where C_O denotes the overhead to establish communication, c_{S_i} denotes the estimated number of transmitted solution mappings contained in the subquery, and C_{map} denotes the costs of transferring a single solution mapping. For SemiJoin $c_{S_i} \cdot C_{map}$ includes the costs for transferring data in both directions.

Processing costs (C_P) are determined by I/O and CPU costs and are very specific to the particular triple store and available indexes, current load, hardware characteristics, implemented algorithms, etc. As such details are not available for endpoints, we

estimate processing costs based on the amount of data that the query is evaluated on. We assume, however, that indexes are used to access triples matching a triple pattern efficiently. We obtain $C_P = \sum_{t=1}^{M}(c_{tp} \cdot C_G)$, where c_{tp} is the estimated number of solution mappings selected by triple pattern t contained in the subquery, and C_G denotes the costs of processing a single triple.

Finally, the costs for processing grouping and aggregation costs for P_{AGG} are estimated as $c_{tp_{AGG}} \cdot C_G$, where $c_{tp_{AGG}}$ represents the number of observations involved in aggregation and C_G represents the costs for processing a single observation.

4.2 Estimating Cost Factors

The cost estimation formulas introduced above rely on several system-specific constants, i.e., C_O, C_{map}, and C_G. As each endpoint has different characteristics, we need to obtain estimates for every endpoint involved in a query. CoDA estimates these values based on several probe queries. The estimates are reused for future queries and repeated regularly to account for changes at the endpoints.

C_{map} is estimated using template queries such as: `SELECT * WHERE {?s #p ?o . FILTER(?o=#o)} LIMIT #L`. This query is executed several times with different values for `#L`, `#o` and `#p` and measures the time it takes to receive an answer from the endpoint. Values for `#o` are taken from a query such as `SELECT DISTINCT(?o) WHERE {?s #p ?o} LIMIT #L`. This is done to measure C_{map} on real values present in the dataset. Based on the pairwise difference between the queries' execution times and the number of retrieved results, we estimate the average time for a single result C_{map}.

C_O is estimated based on queries that do not retrieve data from triple stores such as: `SELECT(1 AS ?v){}` or `ASK{}`. Multiple queries are executed to determine an average.

C_G is estimated based on queries such as: `SELECT COUNT(*) WHERE {?s ?p ?o} GROUP BY #g`. Again, multiple queries with different valid values for `#g` and `#c` are used to build an average. By measuring the time it takes to receive the results and substracting the message overhead C_O and the costs of transferring the result based on C_{map}, we can estimate C_G. Note that C_G represents the costs to process a single input triple. Hence, before computing the average over multiple queries, we need to divide by the number of triples that the aggregate query was computed on – this can conveniently be derived from the query result (`COUNT(*)` is the number of input triples for each group).

Note that these estimates might not be perfectly accurate but this is acceptable for our purposes because we do not aim at accurately predicting execution costs but only to find out which execution plan is more efficient than the others.

4.3 Result Size Estimation

Another important part of the cost model is estimating the size of partial results (result cardinality). Similar to [15, 17], we base our estimations on VoID statistics [3, 23] as this is a standardized format and is most commonly used. Nevertheless, not all SPARQL endpoints offer such statistics. In such cases, we send a series of SPARQL queries with `COUNT` functions to the endpoint to compute the statistics.

VoID statistics can logically be divided into three parts: dataset statistics, property partition, and class partition. The dataset statistics describe the complete dataset: the total number of triples (`void:triples`, c_t), the total number of distinct subjects (`void:distinctSubjects`, c_s), and the total number of distinct objects (`void:distinctObjects`, c_o). The property partition contains such values for each property of the dataset ($c_{p,t}, c_{p,s}, c_{p,o}$). Finally, the class partition shows the number of entities of each class (`void:entities`).

Estimating Result Sizes for Basic Triple Patterns. To estimate result sizes for complex queries, we first need to estimate the result size of basic queries (a single triple pattern and, optionally, a condition expressed by a `FILTER`).

Based on statistics, we estimate the result size c_{res} of a triple patterns as follows: `(?s ?p ?o)` is directly given by c_t, `(s ?p ?o)` is estimated as $\frac{c_t}{c_s}$, `(?s ?p o)` as $\frac{c_t}{c_o}$, and `(s ?p o)` as $\frac{c_t}{c_s \cdot c_o}$. When the predicate of the triple pattern is specified, `(?s p ?o)` is given by $c_{p,t}$, `(s p ?o)` is estimated as $\frac{c_{p,t}}{c_{p,s}}$, `(?s p o)` as $\frac{c_{p,t}}{c_{p,o}}$, and `(s p o)` is assumed to be 1. Tighter estimates based on VoID statistics are possible when the property `rdf:type` is used [17].

We further introduce several optimizations that are often used in relational database systems [13]. As distributions are skewed, we assume a Zipfian distribution of values and multiply c_{res} with the correction coefficient of 1.1 (close to Zipfian ideal). In case a `FILTER` involves an inequality comparison (e.g. $?x >= 10$), we assume that one third of the triples satisfy the requirements and divide c_t or $c_{p,t}$ in the above formulas by a factor of 3. If a `FILTER` contains an expression with the inequality operator (e.g. $?x! = 10$), we need to replace $\frac{1}{c_s}$ with $\frac{c_s-1}{c_s}$ because we select all except 1 out of c_s different values. The same consideration holds for c_o, $c_{p,s}$, and $c_{p,o}$.

Estimating Result Sizes for Joins. To estimate the sizes of join results, we need to distinguish between different shapes of joins: (1) star-shaped joins are characterized by multiple triple patterns joining on the same variable (e.g., `?s1 p1 ?o1 . ?s1 p2 ?o2`) and (2) path-shaped joins are characterized by multiple triple patterns that join on different variables (e.g., `?s1 p1 ?o1 . ?o1 p2 ?o2`).

To estimate the result size, we use the cardinality estimation model proposed in [17]. The model proposes formulas for different types of joins. For example, for queries such as `SELECT ?y WHERE { ?x p1 ?y . ?x p2 ?o1 . FILTER(?o1=10) }` (star-shaped join) the cardinality is calculated as $c_{res} = \frac{\frac{c_{p2,t}}{c_{p2,o1}} \cdot c_{p1,t}}{max(c_{p2,x}, c_{p1,x})}$, while for queries such as `SELECT ?x WHERE { ?x p1 ?y . ?y p2 ?o1 . FILTER(?o1=10) }` (path-shaped join) the cardinality is calculated as $c_{res} = \frac{\frac{c_{p2,t}}{c_{p2,o1}} \cdot c_{p1,t}}{max(c_{p1,y}, c_{p2,y})}$.

Estimating Result Sizes for Grouping and Aggregation. The upper bound for the cardinality of grouping and aggregation is the size of the input, i.e., for a non-restrictive grouping we have $c_{res} = c_{in}$. If the `GROUP BY` clause contains only a subset ($?x_1, \ldots ?x_n$) of the variables contained in the query, then c_{res} (or more specifically c_{AGG}) is bound by the product of the variables' distinct bindings $\prod_{i=1}^{n} distinct(?x_i)$.

When solution reducers are present in the query, such as FILTER statements and/or triples with literals, that are connected to grouping variables through joins, we assume

that the number of distinct values is reduced proportionally: $distinct(?x) = \frac{c_{p_x,x}}{c_{p_y,y} \cdot N}$ where $c_{p_x,x}$ is the number of distinct bindings for variable *?x*, $c_{p_y,y}$ the number of distinct bindings for variable ?y, which is connected to *?x* through star-shaped or path-shaped joins, and N is the reduction factor, which is equal to 1 in case of a solution reducer with equality, 1/3 in case of a solution reducer with inequality, and $\frac{c_{p_y,y}-1}{c_{p_y,y}}$ in case of the a solution reducer with negation [13].

5 Evaluation

In this section, we present the results of evaluating the strategies presented in this paper. Our solution uses the .NET Framework 4.0 and dotNetRDF (`http://dotnetrdf.org/`) to implement a mediator that accepts queries, optimizes their execution using the proposed strategies (SemiJoin, PartialAgg, and MedJoin), and sends subqueries to the SPARQL endpoints, which are using Virtuoso as local triple store.

5.1 Experimental Setup

We evaluate our strategies based on a standard benchmark originally designed to measure the performance of aggregate queries in relational database systems: the Star Schema Benchmark (SSB) [19]. This benchmark is well-known in the database community and was chosen for its simple design (refined decision support benchmark TPC-H [11]) and its well-defined testbed.

RDF Dataset. The data in SSB is generated as relational data. We used different scale factors (1 to 5 – 6M to 30M observations) to generated multiple datasets of different sizes. We translated the datasets into RDF using a vocabulary that strongly resembles the SSB tabular structure. For example, a lineorder tuple is represented as a star-shaped set of triples where the subject (URI) is linked via a property (e.g., `rdfh:lo_orderdate`) to a an object (e.g., `rdfh:lo_orderdate_19931201`) which in turn can be subject of another star-shaped graph. Values such as quantity and discount are connected to lineorder entities as literals. A simplified schema of the RDF structure is illustrated in Fig. 1. Converted datasets contain 110,5M (scale factor 1) to 547,5M (scale factor 5) triples.

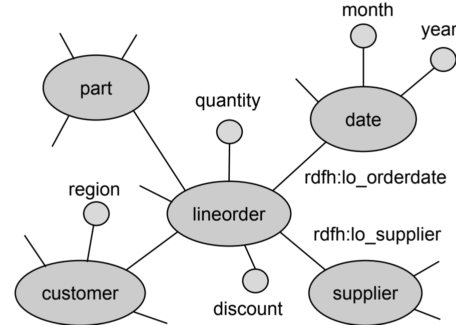


Fig. 1: Simplified Description of the SSB dataset

Queries. SSB defines 13 queries. They represent 4 "prototypical" queries with different selectivity factors. A brief description of the queries is given in Table 1. We converted all 13 queries into SPARQL and used the `SERVICE` keyword to query federated endpoints.

Configuration. To test the queries in a federation of SPARQL endpoints, we partitioned the datasets as follows:

- To simulate two endpoints (one endpoint containing main observation data and one `SERVICE` endpoint containing supporting data), we created two partitions: partition 1 (lineorders, parts, customers, and suppliers) and partition 2 (dates).

Query Prototypes	Query No	Query Parameters for Various Selectivities
Prototype 1. Amount of revenue increase that would have resulted from eliminating certain company-wide discounts.	Q1.1 Q1.2 Q1.3	Discounts 1, 2, and 3 for quantities less than 25 shipped in 1993. Discounts 1, 2, and 3 for quantities less than 25 shipped in 01/1993. Discounts 5, 6, and 7 for quantities less than 35 shipped in week 6 of 1993.
Prototype 2. Revenue for some product classes, for suppliers in a certain region, grouped by more restrictive product classes and all years.	Q2.1 Q2.2 Q2.3	Revenue for 'MFGR#12' category, for suppliers in America Revenue for brands 'MFGR#2221' to 'MFGR#2228', for suppliers in Asia Revenue for brand 'MFGR#2239' for suppliers in Europe
Prototype 3. Revenue for some product classes, for suppliers in a certain region, grouped by more restrictive product classes and all years.	Q3.1 Q3.2 Q3.3 Q3.4	For Asian suppliers and customers in 1992-1997 For US suppliers and customers in 1992-1997 For specific UK cities suppliers and customers in 1992-1997 For specific UK cities suppliers and customers in 12/1997
Prototype 4. Aggregate profit, measured by subtracting revenue from supply cost.	Q4.1 Q4.2 Q4.3	For American suppliers and customers for manufacturers 'MFGR#1' or 'MFGR#2' in 1992 For American suppliers and customers for manufacturers 'MFGR#1' or 'MFGR#2' in 1997-1998 For American customers and US suppliers for category 'MFGR#14' in 1997-1998

Table 1: SSB Queries

- To simulate three endpoints (two `SERVICE` endpoints containing supporting data), we created three partitions: partition 1 (lineorders, parts, customers), partition 2 (dates), and partition 3 (suppliers).
- To simulate four endpoints (three `SERVICE` endpoints containing supporting data), we created four partitions: partition 1 (lineorders, parts), partition 2 (dates), partition 3 (suppliers), and partition 4 (customers).

All the queries and the datasets used for the experiments are available at `http://extbi.cs.aau.dk/coda`.

We used four different machines for our experiments depending on the configuration. We used the most powerful machine (CPU Intel(R) Core(TM) i7-950, RAM 24 GB, HDD 1.5TB RAID5, 1TB SATA, 600GB SAS RAID0) for partition 1. We used three identical machines (CPU AMD(R) Opteron(TM) 285 2.6GHz, RAM 8GB, HDD 80GB) for serving data of partitions 2 to 4. 64-bit Ubuntu 14.04 LTS operating system was installed on all computers. As a mediator, we used a virtual machine with one dedicated core of Xeon E3-1240V2 3.4 GHz (2 threads), 10 GB RAM, 100 GB HDD, and 64-bit Windows Server 2008 Service Pack 1 as operating system. All machines were located on the same LAN. All benchmark queries were executed 5 times following a single warm-up run. During this warm-up run, all statistics and system measurements were obtained, stored in the system, and later used for the subsequent query executions. Statistics were gathered with the help of COUNT queries. Statistics collection took between 18 (scale factor 1) to 129 (scale factor 5) seconds. The execution time for each query is measured on the mediator from the time the query is received from a user till the time the complete results are reported back. We used a timeout of 1 hour for the experiments.

5.2 Experimental Results

As discussed in Section 1, we initially experimented with three systems (Virtuoso, Sesame, and Jena Fuseki). Sesame is always trying to download all triples that match the patterns defined in the `SERVICE` subquery from the remote endpoint and is timing out even for small datasets. Jena Fuseki and Virtuoso are using the same strategy to evaluate `SERVICE` subqueries with grouping and aggregation. We chose Virtuoso

v07.10.3207 as representative for this strategy in our experiments and include results for a native Virtuoso setup, in which Virtuoso is optimizing the distributed execution of the aggregate query.

In our first line of experiments, we measured the runtime for the benchmark queries in the configuration with one SPARQL endpoint. For the SemiJoin strategy, due to issues with large numbers of bindings in the `VALUES` clause in existing endpoints [9], we often have to partition the set of bindings that we aim to pass in a `VALUES` statement into smaller partitions and send a separate messages for each of the partitions.

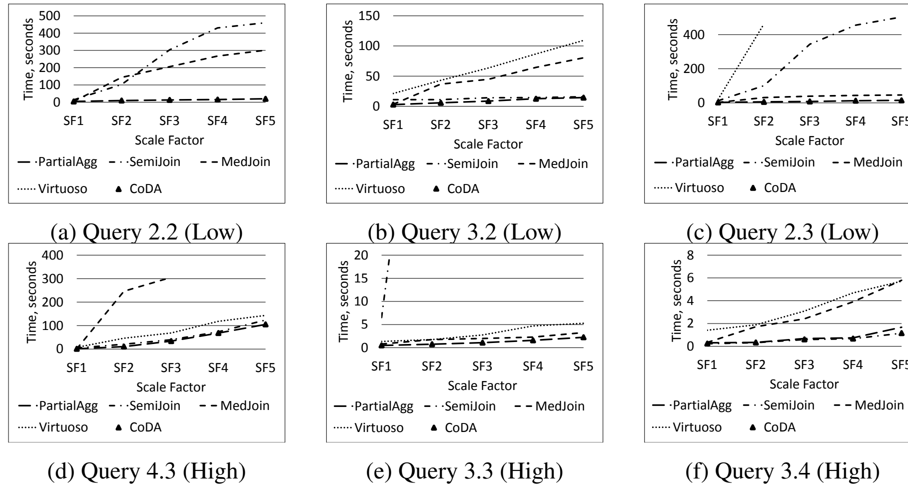
	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3	Q3.1	Q3.2	Q3.3	Q3.4	Q4.1	Q4.2	Q4.3
Scale Factor 1													
Virtuoso	T/O	T/O	760	500,3	107,8	21,3	215,8	21,2	1,4	1,4	863	969	7,3
SemiJoin	**1,3**	**0,2**	**0,1**	12,7	13,5	12,6	**14,1**	11,0	6,5	**0,2**	4,8	**8,1**	4,5
PartialAgg	1,6	1,4	0,8	**9,4**	**4,5**	3	17,5	**2,8**	**0,5**	0,3	**4**	18,5	**1,0**
MedJoin	249,5	213,4	82,9	11	5,2	**2,9**	98,9	3,4	0,8	0,3	26,4	32	1,1
CoDA	**1,3**	**0,2**	**0,1**	**9,4**	**4,5**	**2,9**	**14,1**	**2,8**	**0,5**	**0,2**	**4**	**8,1**	**1,0**
Scale Factor 2													
Virtuoso	T/O	T/O	T/O	950,9	T/O	462,9	992,2	42,9	1,8	1,9	T/O	1054	46,5
SemiJoin	**3,6**	**0,9**	**0,5**	25,7	102,8	101	**15,4**	11,1	89,7	**0,32**	**30,6**	**35,5**	20,6
PartialAgg	17,1	16,5	7,3	**16,2**	**9,5**	**5,9**	18,4	**5,8**	**0,8**	0,34	77,3	37,4	**10,5**
MedJoin	T/O	T/O	T/O	T/O	143,7	31,5	612,7	36,7	1,8	1,7	T/O	T/O	246,7
CoDA	**3,6**	**0,9**	**0,5**	**16,2**	**9,5**	**5,9**	**15,4**	**5,8**	**0,8**	0,33	**30,6**	**35,5**	**10,5**
Scale Factor 3													
Virtuoso	T/O	T/O	T/O	1465	T/O	T/O	T/O	63,5	2,8	3,1	T/O	T/O	68,5
SemiJoin	46,3	**5,4**	**2,2**	330,7	303,4	344,1	**20,2**	14,2	250,7	**0,6**	**45,4**	**105,3**	39,8
PartialAgg	**18,4**	18,8	8,3	**29,5**	**13,2**	**8,2**	23,2	**8,6**	**1,1**	0,7	217,4	606	**33,9**
MedJoin	T/O	T/O	T/O	T/O	205,7	39,5	1312	44,8	2	2,4	T/O	T/O	305,3
CoDA	**18,4**	**5,4**	**2,2**	**29,5**	**13,2**	**8,2**	**20,2**	**8,6**	**1,1**	**0,6**	**45,4**	**105,3**	**33,9**
Scale Factor 4													
Virtuoso	T/O	T/O	T/O	T/O	T/O	T/O	T/O	86,9	4,7	4,7	T/O	T/O	118,4
SemiJoin	64,2	**6.9**	**2,4**	368,5	430,3	455,4	**23,7**	14,5	275,6	**0,7**	**54,2**	**116,2**	73,5
PartialAgg	**33,9**	27,6	9,8	**146,2**	**15,2**	**12,9**	27,2	**12,5**	**1,6**	0,8	980,8	1017	**68,3**
MedJoin	T/O	T/O	T/O	T/O	267,5	43,6	T/O	64,5	2,3	3,9	T/O	T/O	T/O
CoDA	**33,9**	**6.9**	**2,4**	**146,2**	**15,2**	**12,9**	**23,7**	**12,5**	**1,6**	**0,7**	**54,2**	**116,2**	**68,3**
Scale Factor 5													
Virtuoso	T/O	T/O	T/O	T/O	T/O	T/O	T/O	109,2	5,3	5,7	T/O	T/O	143,4
SemiJoin	77,7	**8,4**	**2,9**	453,4	460,3	503,6	**60,9**	15,8	352,9	**1,2**	**59,2**	**126,8**	123,6
PartialAgg	**37,7**	29,2	18,4	**249,5**	**19,8**	**14,9**	78,5	**14,4**	**2,2**	1,7	1565	1577	**105,1**
MedJoin	T/O	T/O	T/O	T/O	301,2	46,3	T/O	80,4	3,3	5,8	T/O	T/O	T/O
CoDA	**37,7**	**8,4**	**2,9**	**249,5**	**19,8**	**14,9**	**60,9**	**14,4**	**2,2**	**1,2**	**59,2**	**126,8**	**105,1**

Table 2: Benchmark Results For Scale Factor 1 to 5, in seconds

Table 2 shows the results for scale factors 1 to 5. CoDA clearly chooses the best strategy for all queries. For scale factor 1, the CoDA algorithm selected the SemiJoin strategy for queries with highly selective subqueries (where the number of intermediate subquery results are low) (Q1.1, Q1.2, Q1.3, Q3.1, Q3.4, and Q4.2), the MedJoin strategy for queries with high selectivity (Q2.3), and the PartialAgg strategy for the rest.

CoDA scales well with the increase in the number of triples as the results for scale factors 2 to 5 in Table 2 show. Due to the increased number of triples to process, the

strategy for Query 2.3 changes from MedJoin to PatrialAgg. CoDA also changed the strategies for queries 1.1 and 4.1 due to different estimations of C_C and C_P for various scale factors. In general, CoDA chooses the best strategy for all queries (the difference between the CoDA approach and the best approach for query Q3.4 in scale factor 2 is due to the overhead of optimization, which is only 14 ms).



(a) Query 2.2 (Low) (b) Query 3.2 (Low) (c) Query 2.3 (Low)

(d) Query 4.3 (High) (e) Query 3.3 (High) (f) Query 3.4 (High)

Fig. 2: Execution Times for Queries with Low and High Selectivity, One Endpoint

Figure 2 shows the execution times for several queries with high selectivity (Q4.3, Q3.3, Q3.4) and low selectivity (Q2.2, Q3.2, Q2.3) for different strategies and scale factors – due to timeouts in execution, some lines end earlier than others. MedJoin and native Virtuoso do not scale well and some queries time out while SemiJoin and PartialAgg return answers for all the queries. This can be explained by the internal logic behind the strategies. For example, Virtuoso sends SPARQL requests for every aggregated observation, while MedJoin needs to transfer much data to the mediator. Due to the result size restrictions (the maximum result set size for Virtuoso is 1,048,576), the system downloads all data in chunks but still times out. In contrast, SemiJoin and PartialAgg transfer only necessary data and are thus reducing the communication costs.

We also evaluated the influence of the number of endpoints. For this purpose, we chose an example query from our workload (Q4.3) that is complex enough to be rewritten into a query with up to three `SERVICE` endpoints and selective enough not to require all triples for the calculation (Fig. 3). Going up to three endpoints, only the PartialAgg strategy was able to answer the query. With data coming from two or three endpoints, the number of values that needs to be passed in the SemiJoin strategy increases and system performance quickly degrades (yellow lines in Fig. 3). With the partition of the dataset into more endpoints, MedJoin also needs to load much more data into the mediator site to answer the query and for the scale factors 3 to 5 this leads to timeouts (green lines in Fig. 3). The same reason (the need to send more requests to answer the query) leads to the timeout in the Virtuoso strategy (red lines) for queries with more than one `SERVICE` endpoint. Therefore, the obvious choice of the CoDA strategy is PartialAgg (blue lines) in these cases.

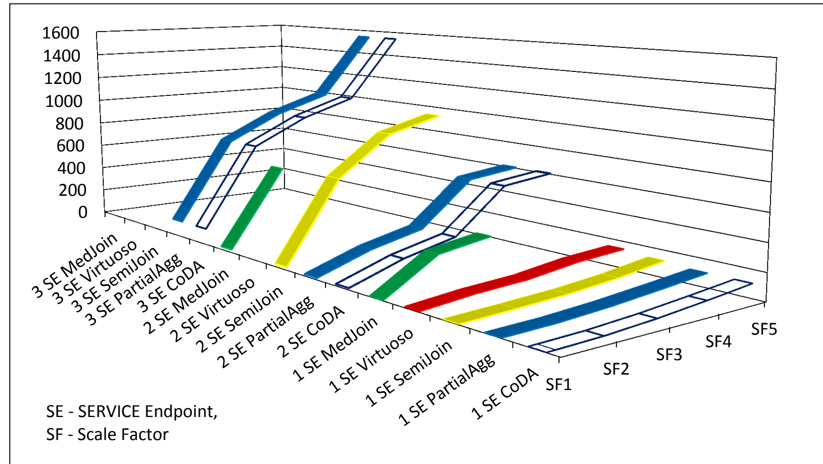


Fig. 3: Execution of Query 4.3 over Several Endpoints

In summary, the experimental results show that CoDA is able to select the best strategy and thus executes all queries for RDF data of all tested data sizes.

6 Conclusions and Future Work

Motivated by the increasing availability of RDF data over SPARQL endpoints, the new powerful aggregation functionality in SPARQL 1.1, and the desire to perform ad-hoc analytical queries, this paper investigated the problem of efficiently processing aggregate queries in a federation of SPARQL endpoints.

More precisely, the paper proposed the Mediator Join, SemiJoin, and Partial Aggregation query processing strategies for this scenario. The paper also proposed a cost model, and techniques for estimating constants and result sizes for triple patterns, joins, grouping and aggregation, and the combination of these with the processing strategies into the Cost-based Optimizer for Distributed Aggregate queries (CoDA) approach for aggregate SPARQL queries over endpoint federations. The comprehensive experimental evaluation, based on an RDF version of the widely used Star Schema Benchmark, showed that CoDA is efficient and scalable, able to pick the best query processing plan in different situations, and significantly outperforms current state-of-the art triple stores.

Interesting directions for future work include using more complex statistics with precomputed join result sizes and correlation information to better estimate cardinalities, optimizing the execution of more complex queries (e.g., with optional patterns or complex aggregation functions), and investigating the influence of ontological constraints and inference/reasoning in the context of federated aggregate SPARQL queries.

Acknowledgment This research is partially funded by the Erasmus Mundus Joint Doctorate in "Information Technologies for Business Intelligence – Doctoral College (IT4BI-DC)".

References

1. M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. In *ISWC'11*, 2011.

2. Z. Akar, T. G. Halaç, E. E. Ekinci, and O. Dikenelli. Querying the Web of Interlinked Datasets using VoID Descriptions. In *LDOW'12*, 2012.
3. K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. Describing linked datasets. In *LDOW'09*, 2009.
4. S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. DBpedia: A Nucleus for a Web of Open Data. In *ISWC'07*, 2007.
5. C. Basca and A. Bernstein. Avalanche: Putting the Spirit of the Web back into Semantic Web Querying. In *SSWS'10*, 2010.
6. T. Berners-Lee. Linked Data. W3C Design Issues. `http://www.w3.org/DesignIssues/LinkedData.html`, 2006.
7. C. Buil-Aranda, M. Arenas, and Ó. Corcho. Semantics and Optimization of the SPARQL 1.1 Federation Extension. In *ESWC'11*, pages 1–15, 2011.
8. C. Buil-Aranda, M. Arenas, O. Corcho, and A. Polleres. Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *Web Semantics*, 18(1):1–17, 2013.
9. C. Buil-Aranda, A. Hogan, J. Umbrich, and P-Y. Vandenbussche. SPARQL Web-Querying Infrastructure: Ready for Action? In *ISWC'13*, pages 277–293, 2013.
10. C. Buil-Aranda, A. Polleres, and J. Umbrich. Strategies for Executing Federated Queries in SPARQL 1.1. In *ISWC'14*, pages 390–405, 2014.
11. Transaction Processing Performance Council. TPC Benchmark H – Decision Support. `http://www.tpc.org/tpch`.
12. A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.
13. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book*. Pearson Education, second edition, 2009.
14. O. Görlitz and S. Staab. Federated Data Management and Query Optimization for Linked Open Data. In *New Directions in Web Data Management*, pages 109–137. Springer Berlin Heidelberg, 2011.
15. O. Görlitz and S. Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VoID Descriptions. In *COLD'11*, 2011.
16. J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *ICDE'96*, pages 152–159, 1996.
17. S. Hagedorn, K. Hose, K-U. Sattler, and J. Umbrich. Resource Planning for SPARQL Query Execution on Data Sharing Platforms. In *COLD'14*, 2014.
18. G. Ladwig and T. Tran. SIHJoin: Querying Remote and Local Linked Data. In *ESWC'11*, pages 139–153, 2011.
19. P. O'Neil, E. J. O'Neil, and X. Chen. The star schema benchmark (SSB). Technical report, UMass/Boston, June 2009.
20. A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *ISWC'11*, pages 601–616, 2011.
21. T. Urhan and M. J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.
22. M. Wick. GeoNames geographical database. `http://www.geonames.org`.
23. World Wide Web Consortium. Describing Linked Datasets with the VoID Vocabulary (W3C Interest Group Note 03 March 2011). `http://www.w3.org/TR/void/`.
24. World Wide Web Consortium. SPARQL 1.1 Overview (W3C Recommendation 21 March 2013). `http://www.w3.org/TR/sparql11-overview/`.
25. M. Wylot, J. Pont, M. Wisniewski, and P. Cudré-Mauroux. dipLODocus[RDF] - Short and Long-Tail RDF Analytics for Massive Webs of Data. In *ISWC'11*, pages 778–793, 2011.