Exploration of Knowledge Graphs via Online Aggregation

Oren Kalinsky
*Amazon**
Israel
orenkalinsky@gmail.com

Aidan Hogan
DCC, Universidad de Chile & IMFD
Chile
ahogan@dcc.uchile.cl

Oren Mishali
Technion, Israel Institute of Technology
Israel
omishali@cs.technion.ac.il

Yoav Etsion
Technion, Israel Institute of Technology
Israel
yetsion@technion.ac.il

Benny Kimelfeld
Technion, Israel Institute of Technology
Israel
bennyk@cs.technion.ac.il

***Abstract*—Exploration systems over large-scale RDF knowledge graphs often rely on aggregate count queries to indicate how many results the user can expect for the possible next steps of exploration. Such systems thus encounter a challenging computational problem: evaluating aggregate count queries efficiently enough to allow for interactive exploration. Given that precise results are not always necessary, a promising alternative is to apply online aggregation, where initially imprecise results converge towards more precise results over time. However, state-of-the-art online aggregation algorithms, such as Wander Join, fail to provide accurate results due to frequent rejected paths that slow convergence. We thus devise an algorithm for online aggregation that specializes in exploration queries on knowledge graphs; our proposal leverages the low dimension of RDF graphs, and the low selectivity of exploration queries, by augmenting random walks with exact partial computations using a worst-case optimal join algorithm. This approach reduces the number of rejected paths encountered while retaining a fast sample time. In an experimental study with random interactions exploring two large-scale knowledge graphs, our algorithm shows a clear reduction in error over time versus Wander Join.**

I. Introduction

A variety of prominent knowledge graphs have emerged in recent years, including open knowledge graphs such as DBpedia [1], Freebase [2], LinkedGeoData [3], Wikidata [4], and YAGO [5]. Several companies have also announced the creation of proprietary knowledge graphs to power a variety of Web applications, including eBay, Facebook, Google, and Microsoft [6], among others.

Due to their scale and diversity, a major challenge faced when considering a knowledge graph is to understand what content it holds: what sorts of entities it describes, what sorts of relations are represented, the extent of the coverage of particular domains, etc. Prominent knowledge graphs, such as DBpedia [1], Freebase [2] and Wikidata [4], contain in the order of tens of millions of nodes and billions of edges represented using thousands of classes and properties, spanning innumerable different domains. A variety of approaches [7]–[12] have been proposed to help users explore large-scale knowledge graphs by summarizing their content in terms of the relationships between entities, the most common types, and so forth. Such systems are enabled through *exploratory queries* that provide counts of distinct elements matching certain criteria in the knowledge graph, further indicating how many results can be expected for the next exploration steps [7]–[13].

Given the volume and diversity of prominent knowledge graphs, the number of (intermediate) results that can be generated by exploratory queries, and the goal of supporting interactive exploration, a major challenge faced is that of performance. The general trend in the aforementioned exploration systems is towards computing the aggregation operations for exploration either offline [13], or by relying on off-the-shelf query engines [14]–[16]. However, computing aggregations offline leads to space limitations, where for diverse knowledge graphs (with tens of thousands of properties, classes, etc. that can be combined), typically only a subset of relevant results can be materialized [13], [17], [18]. Conversely, in preliminary experiments with Virtuoso [19] – a state-of-the-art [20]–[22] SPARQL query engine – we found, for example, that computing the distribution of properties over all nodes in DBpedia online (i.e., computing how many distinct nodes have each property defined) takes over 5 minutes, which precludes the possibility of interactive exploration.

To face the critical performance problem, we investigate two orthogonal approaches. First, we explore the deployment of a query engine from the recent breed of *Worst-Case Optimal* Join (WCOJ) algorithms [23]–[27] in order to avoid an explosion of intermediate results when processing multiway joins over large graphs. It was shown that these algorithms are not only theoretically better than traditional approaches, they are also empirically superior on graph query patterns joining relations with low dimension. Specifically, we select the Cached Trie Join algorithm [27] that offers superior performance results on path counting queries, which

* Work was done prior to joining Amazon.

matches the nature of our exploration queries. Second, given that WCOJ algorithms can still take tens of seconds to run and precise counts are not always required for visualization, we explore *online aggregation* algorithms [28] that trade precision for performance, computing approximate counts at a fraction of the cost observed even in the WCOJ setting; for these purposes, we select the Wander-Join algorithm [29]. In essence, Wander Join applies a random walk between database tuples that (jointly) match the join query, and upon termination, updates an estimator of the aggregate function.

Ultimately we conclude that these two approaches are complementary [30] and offer an algorithm that combines online aggregation with exact computation. The general idea is to apply the random walk of Wander Join, and at each step, consider replacing the remaining walk with a precise computation of the space of possible suffixes, this time using Cached Trie Join. This consideration is done via an estimate of selectivity. Using this approach substantially reduces the rejection rate of the random walks on highly selective patterns. Moreover, the estimator needs to be updated accordingly, and we prove that it remains unbiased. We further extend our algorithm to estimate counts in the presence of the distinct operator, which is crucial to our exploration use case. We call the resulting algorithm Audit Join, and prove that it provides unbiased estimators of counts with and without the distinct operator. In experiments that evaluate randomly-generated exploration queries over two knowledge graphs – specifically DBpedia [31] and LinkedGeoData [3] – our algorithm dramatically reduces error with respect to computation time versus Wander Join.

Our contribution is summarized as follows:

1) We evaluate three main alternatives for evaluating exploration queries: an off-the-shelf SPARQL engine (Virtuoso), a WCOJ algorithm (Cached Trie Join), and an online aggregation algorithm (Wander Join). This evaluation reveals the critical limitations of all alternatives for the use-case of exploring large-scale knowledge graphs.
2) We devise Audit Join – a specialized online-aggregation algorithm for exploring knowledge graphs – and prove that it produces unbiased estimates of counts.
3) We describe an experimental study of performance over random explorations, showing the benefits of Audit Join over the aforementioned alternatives.

II. Related Work

Next, we discuss related work on exploration tools for knowledge graphs, and some relevant approaches to query evaluation.

Exploration Tools. Various approaches have been proposed in recent years for exploring and visualizing graph data [32].

Faceted Browsing: In *faceted browsers*, users incrementally add restrictions – called *facets* – to refine the current results [33]. Often facets are annotated with counts to indicate the results returned if selected (and to remove facets leading to empty results). Some faceted systems focus on smaller graphs from specific domains, such as mSpace (multimedia) [34], BrowseRDF (crime) [35], /facet (art) [36], Ontogator (art) [37], ReVeaLD (biomedicine) [38] and Hippalus (zoology) [39]. Other faceted systems have been proposed for multi-domain knowledge graphs, which contain thousands of classes and properties and potentially billions of triples. Among these systems, we can mention Neofonie [40], Rhizomer [17], SemFacet [41], Semplore [42] and Sparklis [43] for exploring DBpedia; Broccoli [18] and Parallax [44] for exploring Freebase; and GraFa for exploring Wikidata [13]. Of these systems, many do not present runtime performance evaluation [17], [40], [44], [45], delegate query processing to a general-purpose query engine [17], [43], [44], [46], apply a manual selection of useful facets or a subset of data [40], [41], and/or otherwise rely on a materialization approach to precompute and cache aggregated meta-data (such as counts) [13], [17], [18].

Graph Profiling: Other works on *graph profiling* focus on summarizing the content of a large knowledge graph for the user [7]. Some systems provide a *graph summary* or *quotient graph* [47], which groups nodes into super-nodes, between which the most important relations are then summarized. The partition of nodes may be based on bisimulations [48], formal concept analysis [49], [50], semantic types [51]–[54], etc. Other works generate statistical summaries of large graphs, in terms of the most popular classes, properties, etc., offering interactive visualizations [8]–[12]. These typically either apply offline aggregations or use general-purpose query engines.

Query Engines. We provide an overview of approaches for querying knowledge graphs as relevant to this work.

SPARQL Engines: SPARQL [55] is the standard for querying RDF graphs, and is used by public query services over the DBpedia, LinkedGeoData, and Wikidata knowledge graphs. While several query engines support SPARQL (e.g., [14]–[16]), we adopt Virtuoso [56] as a representative of this strategy given its competitiveness in various benchmarks (e.g. [20], [22]) and the fact that it optimizes for aggregate queries by applying vectorized execution on columns represented as compressed vectors of values [56].

Worst-Case-Optimal Joins: Worst-case optimal join algorithms (e.g., [23], [25]–[27]) evaluate join queries with a runtime guarantee that meets the Atserias–Grohe–Marx (AGM) bound [57]: a worst-case tight bound for the size of the output. Such algorithms are not only theoretically better than traditional approaches, they have been empirically shown to offer better performance for evaluating graph query patterns [26], [27], [58]. In this paper, we will adopt Cached Trie Join [27] (CTJ): a state-of-the-art worst-case optimal join algorithm that offers better performance on aggregate queries compared to other approaches, including dynamic programming algorithms with superior theoretical bounds.

Online Aggregation: Algorithms for online aggregation provide approximate results for aggregate queries that converge

Fig. 1: System architecture where G denotes the (RDF) knowledge graph, Q denotes an exploration query, $Q(G)$ denotes the results of Q over G; dashed and solid arrows denote offline and online interactions respectively

over time to the exact result. This provides a tradeoff between the accuracy and run-time, where users will immediately receive an initial estimate for the query, which will improve as they wait. Since the concept was coined by Hellerstein et al. [28] works have proposed support for additional operators and better statistical guarantees [59], as well as distributed and parallel support [60]–[62]. While the solution was originally for a single table, Haas et al. [63] developed Ripple Join, an online aggregation algorithm that supports joins. More recently, Li et al. [29] have introduced the Wander Join algorithm for online aggregation over join results based on random walks. Wander Join has also been used as an unbiased sampling method for approximate query answering [64] and join-size estimation [30], [65]. Section IV describes the Wander Join algorithm in more detail with examples.

Novelty. The runtime performance of aggregation queries is a critical and often limiting factor for interactive exploration of knowledge graphs. We empirically show that existing approaches are insufficient for large knowledge graphs: exact computation is too slow and existing online aggregation algorithms are slow to converge due to rejecting a high number of "dead-end" random walks. We thus design a novel online aggregation strategy, called Audit Join, that switches from random walks to a worst-case optimal join algorithm when encountering low-selectivity joins that cause high rejection rates. We further prove that this strategy provides an unbiased estimator of counts with and without a distinct operator, where, to the best of our knowledge, no existing online aggregation algorithm offers unbiased estimators in the distinct case.

III. Exploration Use-Case

Our proposed Audit Join algorithm can be applied in a broad range of use-cases that require fast and accurate approximations for aggregate count queries over knowledge graphs. For a concrete motivation, we now summarize the exploration system that prompted the effort on Audit Join and serves as our main use-case. A more detailed description and examples of the user interface and exploration model can be found in our demonstration paper of the system [12] (that used Virtuoso as its query engine).

Our system offers online exploration of large-scale knowledge graphs and is implemented as a web application that communicates with a specialized query engine, as illustrated in Figure 1. In practice, the system can use any query engine that supports aggregate queries over graph patterns (specifically, count-distinct over joins); however, our engine aims to provide interactive exploration with subsecond performance, including in cases where millions of elements need to be counted. We discuss query engines in Section IV. The user experience is visual: no SPARQL knowledge is required from the user. In principle, the user should have only a basic understanding of what classes and properties are.

Data. Our system supports the exploration of an RDF graph, which is a set of RDF triples. Each such triple consists of a *subject*, a *predicate*, and an *object*. An RDF graph can be viewed as a directed edge-labeled graph where each triple encodes an edge. The subjects and predicates come from a collection of *Unique Resource Identifiers* (URIs) and the object is either a URI or a *literal* (e.g., a number). In the remainder of this section, we assume a fixed underlying RDF graph. We refer to terms used in the predicate position of a triple (e.g., birthPlace) as *properties*. Nodes in this RDF graph may be instances of *classes* (e.g. Person, Movie, etc.) where these classes may be further organized into a *subclass hierarchy* (e.g., defining Movie to be a subclass of Work). A URI u is said to be *of class* c if the RDF graph contains the triple $(u, \text{rdf:type}, c)$. One could also define membership based on the *transitive closure* on subclasses; the choice between the two is orthogonal to our model.

Functionality. Before describing our system in more detail, we illustrate its functionality with an exploration example for DBpedia, which results in the view shown in Figure 2.

Example III.1. Suppose that the user is interested in philosophers, and in particular, they wish to learn about people who have influenced philosophers. Starting from the top-level class owl:Thing, the user is presented with a chart displaying its direct subclasses, with the height of each bar in the chart indicating the number of instances of each subclass. From this chart, the user clicks on Agent to now view a chart with its direct subclasses, subsequently selecting Person and then Philosopher. The user can then switch to a property view to see a chart representing either the incoming properties or outgoing properties on instances of Philosopher. Selecting the bar for influencedBy from the outgoing properties opens a new chart, showing the different class instances connected to philosophers via the influencedBy property. Clicking the Person type in this chart and selecting the outgoing property view then reveals the bar chart shown in Figure 2, showing outgoing properties on instances of type Person that have influenced philosophers; this allows the user to further explore *only* people who have influenced philosophers and not the entire set of Person instances. □

We now define more formally the system's model of visual exploration. This model is based on bar charts constructed in an iterative and interactive manner. We have three kinds of bars. A *class bar* represents URIs of a common class (e.g., Person). An *outgoing-property bar*, or *out-property bar* for short, represents URIs that are the subject (source) of a common associated outgoing property (e.g., subjects of locatedIn triples). Analogously, an *incoming-property bar*, or

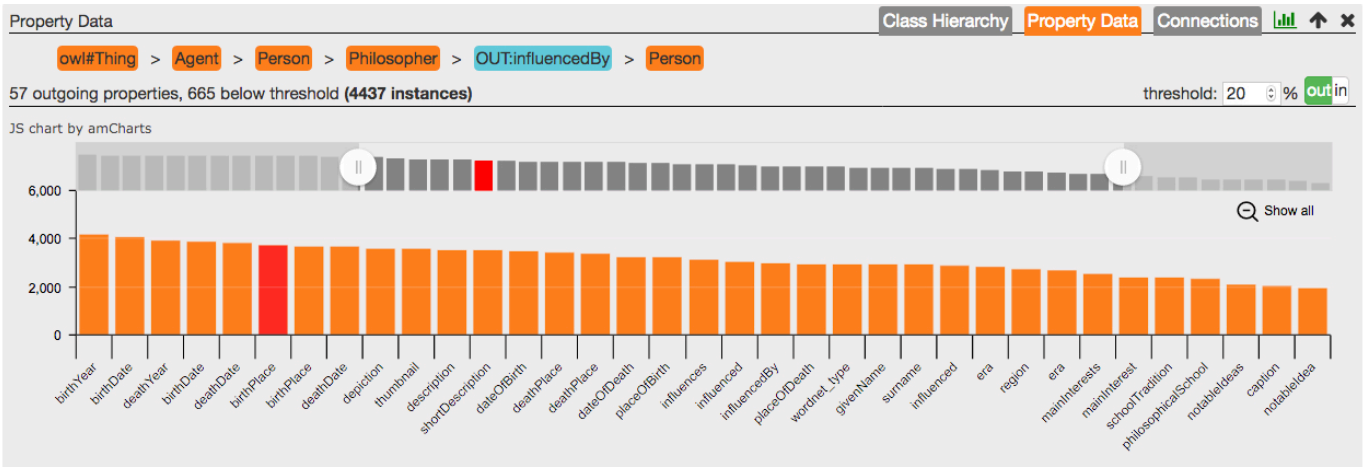


Fig. 2: An exploration pane over DBpedia showing outgoing properties about *persons* who have influenced *philosophers*.

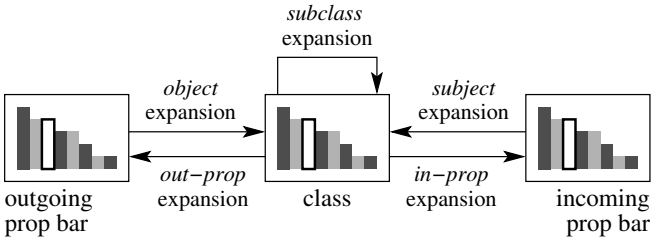


Fig. 3: State transitions in the exploration model

in-property bar, represents URIs that are the object (target) of a common associated incoming property (e.g., objects of locatedIn triples).

The *category* of a bar is the corresponding class or property, depending on the kind of the bar. A *bar chart* (or simply *chart*) is a mapping from categories to bars. Figure 2 presents an instance of an outgoing-property bar chart in our system, which we will further refer to later. In this example, the category of the bar marked in red is the *birthPlace* property and the height of the bar corresponds to the number of distinct instances with this property.

Bar expansion The user can navigate using *bar expansions* by clicking on the bars. A bar expansion is a function that transforms the clicked bar into a new chart, with each successive chart being the expansion of a bar from the previous chart. We define five types of bar expansions that offer natural exploration steps. These expansions lead to a transition system between chart types, as seen in Figure 3.

In *subclass expansion*, we do the following. Given a class bar with a category (class) c, this expansion displays all categories that are direct subclasses of c, that is, all c' such that the graph contains $(c', \text{rdfs:subClassOf}, c)$. In the new chart, each bar has the category c' and it consists of all the nodes of the clicked bar (of category c) with the class c'.

Given a class bar of category c, *out-property expansion* displays all categories that are *outgoing properties* of c, that is, all p such that the graph contains a triple (s, p, o) for some s in the bar. The new chart has an out-property bar with category p for each relevant property p, and the nodes of the resulting bar are the URIs of the clicked bar that have the property p. See Figure 2 for an example. Similarly, *in-property expansion* is analogous to the out-property expansion, except that the bar of p in the new chart is an *in-property* bar with the category p, and it consists of all URIs that have p as an incoming property; that is, all o from the clicked bar such that (s, p, o) is a graph triple for some s.

Object expansion is enabled only for out-property bars. Recall that, in this case, the category is a property p. The categories of the new chart are the classes of the *objects* that are connected to the nodes of the clicked bar through the property p; that is, these are the classes c such that for some (s, p, o) it is the case that s is in the clicked bar and o is of class c. Hence, the new chart contains a class bar with the category c that consists of all such o. Analogously, *subject expansion* considers the subjects of incoming properties instead of the objects of outgoing properties. In particular, when expanding the bar of p, each bar of category c in the new chart consists of the subjects s of class c such that the graph has the triple (s, p, o) where o belongs to the clicked bar.

IV. Query Engine and Algorithms

To support interactive exploration in our use-case, the query engine should answer join queries with grouped, distinct counts in less than a second. Though such queries form a natural fragment for exploring and profiling knowledge graphs [7], [32], [33] – where joins represent the user's exploration path, with counts used to summarize results – we did not find any query engine that could evaluate such queries with the sub-second response times required to enable interactive exploration of large knowledge graphs with hundreds of millions or billions of edges.

More specifically, in initial experiments with the Virtuoso system, such queries would sometimes take minutes to complete on knowledge graphs such as DBpedia. On the other

```
SELECT α COUNT(DISTINCT β) WHERE {
    a_1 b_1 c_1.
    ...
    a_n b_n c_n.
} GROUP BY α
```

Fig. 4: The general form of an exploration query

hand, algorithms that implement worst-case-optimal joins have recently been shown to be capable of orders-of-magnitude speedup on path counting queries compared to traditional join approaches [66], [67] (including join algorithms with linear complexity [27]), and hence, offer a promising alternative. Still, in experiments with Cached Trie Join [27] – a state-of-the-art representative of these join algorithms – queries that require large join results on multi-domain knowledge graphs (e.g., DBpedia) may still take tens of seconds to run.

In order to achieve acceptable performance, we turn to *online aggregation*, relaxing the expectation of exact counts to instead aim for a fast but approximate initial response whose error reduces over time [28]. Such a compromise is well justified in the context of our exploration use-case, which can suffer some loss of precision without impacting the user experience. We thus investigate Wander Join [29], which is designed for aggregate queries over the grouped results of join queries; this algorithm has been demonstrated to offer much better convergence compared to traditional online aggregation approaches in experiments over TPC-H [29]. However, Wander Join has two limitations for our use-case:

1) rejected paths slow convergence of the estimation, *and*
2) it does not support (i.e., provide an unbiased estimator) for the count-*distinct* operator.

The main contribution of this paper is thus to propose a novel online-aggregation algorithm, Audit Join, which addresses these limitations of existing algorithms. We adopt the approach of online exploration algorithms that combine random walks with exact computations for the problem of uniform random sampling [30]. First, in cases where a high number of rejected paths are deemed likely to occur, Audit Join defers to partial exact computations using Cached Trie Join. Second, Audit Join incorporates a novel estimator for counts under the distinct operator that we prove to be unbiased.

This section discusses the various algorithms we investigate to improve query performance in our interactive exploration setting, starting with preliminaries on query translation, then discussing Cached Trie Join and Wander Join, before proposing Audit Join.

A. Query Translation and Structure

In Section III, we defined our exploration model. The five operations of subclass, in-property, out-property, object and subject expansions are translated to SPARQL queries. These SPARQL queries produce the information required to generate the next bar chart by first executing a multiway join that encodes the expansions thus far, then a grouping on the URIs of the next chart, and finally a distinct count on the focus set of the next chart. Given the structure of exploration steps, cyclic queries cannot occur.

The general form of these SPARQL queries is illustrated by the query template in Figure 4. Here, each pattern of the form $a_i\ b_i\ c_i$ refers to a *triple pattern*, where each term a_i, b_i and c_i (where $1 \leq i \leq n$) is either a variable (e.g., `?s`) or a constant (e.g., `<Person>`). A variable may appear in at most two triple patterns. Finally, α denotes a variable that will be assigned the URIs of the next bar chart (either some b_i, or some c_i where b_i = `rdf:type`), while β returns the focus set of the next bar chart (either some a_i or c_i). As an example, the exploration *birthplaces of persons* is translated to the SPARQL query shown in Figure 5. Aside from our exploration system, such queries form the basis of a variety of systems for searching and exploring knowledge graphs [7], [32], [33].

Remark. In practice, patterns with the "`rdf:type`" property are joined with the transitive closure of subclasses. For example, in the pattern "`?s rdf:type <Person>`" of Figure 5, `?s` will also be mapped to instances of (possibly indirect) subclasses of `<Person>`. We materialize this subclass closure and view it as a raw relation; instances, on the other hand, are typed per the original data and joined with the subclass closure at runtime. For simplicity, we leave the subclass closure (and other forms of inference) implicit in the presentation of the queries since it is orthogonal to the model. □

In what follows, we denote by G_i the subset of triples of the knowledge graph G that match the triple pattern (a_i, b_i, c_i), where a triple (a, b, c) *matches* (a_i, b_i, c_i) if the two agree on the constants (i.e., if a_i is a constant then $a = a_i$, and so on).

B. Aggregation via Cached Trie Join

The exact evaluation we incorporate in our approach is based on the *Cached Trie Join* algorithm (CTJ) [27]. This algorithm incorporates caching of intermediate join results on top of the *LeapFrog Trie Join* algorithm (LFTJ) [23]—a backtracking join algorithm that traverses over trie indexes. In our context, we maintain six trie indexes over G, each corresponding to an ordering of the three attributes (s, p and o). The trie index has a root, and under the root a layer with the values of the first attribute, and then a layer with the values of the second attribute, and then the third attribute. Each triple (s, p, o) corresponds to a unique root-to-leaf path of the trie. For example, if the order is (p, o, s), then the first layer corresponds to the predicates, the second to the objects, and the third to the subjects; in this case, a path root $\rightarrow b \rightarrow c \rightarrow a$

```
SELECT ?c COUNT(DISTINCT ?o) WHERE {
   ?s <birthPlace> ?o.
   ?s rdf:type <Person>.
   ?o rdf:type ?c.
} GROUP BY ?c
```

Fig. 5: An instance of an exploration query

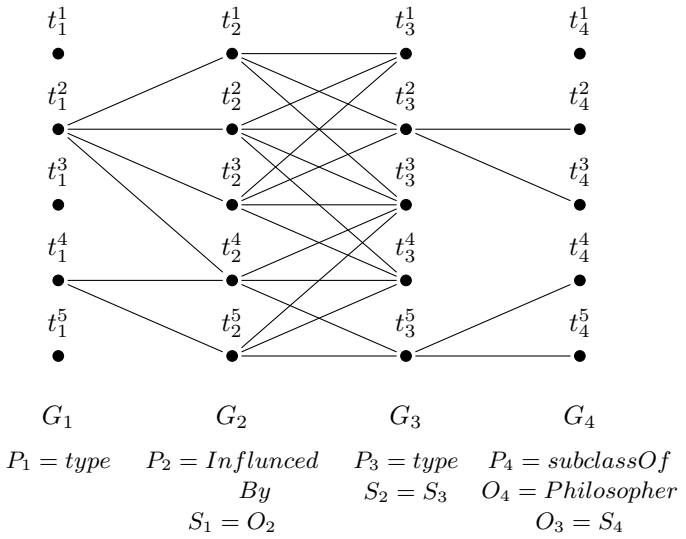


Fig. 6: A join graph example of Object Expansion of the `InfluencedBy` property and subjects of type `Philosopher` from the DBpedia knowledge graph.

represents the triple (a, b, c) of G. In our implementation, B-tree like indexes are used, similar to the indexes commonly used in SPARQL query engines.

LFTJ assumes a predetermined order over the variables, say $x_1, \ldots, x_m$. We access the tuples of G_i using a trie T_i with an order consistent with the predetermined order. For example, if the triple pattern is `?q <birthPlace> ?r`, and `?r` precedes `?q` in the predefined order, then T_i will be the trie for (o, s, p), (o, p, s), or (p, o, s). LFTJ uses a backtracking algorithm that walks over the T_i and looks for assignments for $x_1, \ldots, x_m$. It starts by finding the first matching value v_1 for x_1. Then, the tries T_i that contain x_1 restrict their search to the subtree under $x_1 = v_1$. Next, it looks for the first match v_2 of the next variable x_2, and all relevant tries restrict to the subtree under $x_2 = v_2$. The algorithm continues to remaining variables, until a match is found for all variables, or no match is found for the next variable. Once a match is found, or the algorithm gets stuck, it backtracks to the next value of the previously scanned x_i. Grouping and counting are applied in the natural way.

While LFTJ guarantees worst-case optimality, it frequently re-computes the same intermediate joins, since it does not materialize any of the intermediate results [27]. To effectively reuse the partial answers, CTJ augments LFTJ with a cache structure guided by a tree decomposition of the query, guaranteeing the correctness of the algorithm. In the use-case of this paper, the tree decomposition is easily determined by the path formed by the query. CTJ uses different caching schemes to cache partial count results that are later reused. Empirically, CTJ can achieve orders of magnitude speedup over LFTJ and other known join algorithms for graph queries on relations with a low dimension [27].

Example IV.1. To illustrate the effectiveness of CTJ versus LFTJ, we continue our running example from Example III.1. Specifically, we select the object expansion that returns the objects (and their corresponding types) that were `InfluencedBy Philosophers`. Figure 6 depicts an example join graph of the query. There, each column in the figure is a graph G_i, and each node t_i^j is a tuple of G_i. An edge exists between two tuples if they agree on their join attributes. For instance, G_2 ($P_2 = InfluencedBy$) holds all the subjects that were influenced by the objects and the join with $G_1 (S_1 = O_2)$ returns the types of all the influenced objects. We assume that LFTJ uses the same predetermined join order as in the figure from left to right. LFTJ will backtrack over the join tree, recalculating the number of results under t_2^4, t_3^2 and t_3^5 multiple times since they have multiple incoming paths (e.g., $(t_1^2, t_2^4), (t_1^4, t_2^4)$). Nevertheless, CTJ will encounter these nodes one time each and store the number of results in its cache until it encounters them in later steps.

C. Wander Join

Wander Join is an online aggregation algorithm that is designed for aggregates over joins [66], [67]. Since Wander Join does not support the distinct operator, we ignore the operator in this section. We also postpone discussion of grouping for later in the section.

Given the query of Figure 4, Wander Join samples query answers via independent random walks over the G_i, in contrast to the full pre-order traversal of CTJ. It estimates counts using the Horvitz–Thompson estimator [68], where each random walk γ produces an estimator $C_{\mathsf{wj}}(\gamma)$ (described next), and the final estimator is the average of $C_{\mathsf{wj}}(\gamma)$ over all random walks γ. The walk γ is constructed as follows. First select a random tuple t_1 uniformly from G_1. Next select a tuple t_2 that is consistent with t_1; that is, t_1 and t_2 agree on their common attributes; the choice is again uniform among all consistent t_i. Continue in this way, selecting a random tuple t_i from G_i in the i^{th} step such that t_i is consistent with t_{i-1}. If, at any point, no matching t_i exists, the random walk γ terminates, the path is rejected, and $C_{\mathsf{wj}}(\gamma) = 0$. Otherwise, let d_i be the number of ways of selecting t_i, for $i = 1, \ldots, n$. The probability of γ is $\prod_{i=1}^{n} 1/d_i$. The estimator $C_{\mathsf{wj}}(\gamma)$ is defined by

$$C_{\mathsf{wj}}(\gamma) := \prod_{i=1}^{n} d_i = \frac{1}{\Pr(\gamma)}\,.$$

The estimator $C_{\mathsf{wj}}(\gamma)$ is unbiased; consequently, the final estimator (i.e., the average) is also unbiased. To see why C_{wj} is unbiased, denote by $\mathbf{\Gamma}$ the set of all successful (full) paths from G_1 to G_n. Then the sought count is $|\mathbf{\Gamma}|$. Indeed,

$$\mathbb{E}[C_{\mathsf{wj}}] = \sum_{\gamma \in \mathbf{\Gamma}} \Pr(\gamma) \cdot C_{\mathsf{wj}}(\gamma) = \sum_{\gamma \in \mathbf{\Gamma}} \frac{\Pr(\gamma)}{\Pr(\gamma)} = |\mathbf{\Gamma}|\,.$$

Example IV.2. To demonstrate Wander Join, we use our running example of `InfluencedBy Philosophers` depicted in Figure 6. The random walk is from left to right. Choosing the random path $\gamma_1 = (t_1^2, t_2^2, t_3^2, t_4^2)$ yields the estimate $C_{\mathsf{wj}}(\gamma_1) = 5 \cdot 4 \cdot 4 \cdot 2 = 160$ objects (and their

corresponding types) that were influenced by philosophers. For $\gamma_2 = (t_1^4, t_2^5, t_3^5, t_4^5)$ we will get $C_{\mathsf{wj}}(\gamma_2) = 5 \cdot 2 \cdot 3 \cdot 2 = 60$. Finally, partial paths, such as (t_1^2, t_2^2, t_3^3), will yield the estimate zero. The final estimator is the average over all estimates (including the zero estimates). □

Wander Join adapts to *grouping* similarly to Ripple Join [63]: maintaining separate estimators for each group, using the random walk γ to only update the separator of the group to which γ belongs.

Wander Join can further offer a confidence interval together with the estimator [59]. For example, taking a confidence level of 0.95:

$$\Pr(|C_{\mathsf{wj}}(\gamma) - |\mathbf{\Gamma}|| \leq \epsilon) \geq 0.95.$$

The confidence level quantifies the probability that $|C_{\mathsf{wj}}(\gamma) - |\mathbf{\Gamma}||$ is in the (half-width) confidence interval, denoted by ϵ. The confidence interval (CI) should shrink together with the estimator.

D. Audit Join

We first describe Audit Join without the distinct operator. We also ignore grouping, since Audit Join is adapted to grouping similarly to Wander Join and Ripple Join. Hence, our goal is again to estimate $|\mathbf{\Gamma}|$, where $\mathbf{\Gamma}$ is the set of all full random walks γ from G_1 to G_n.

Base Algorithm. For a prefix $\delta = (t_1, \ldots, t_\ell)$ of a random walk, denote by $\mathbf{\Gamma}_\delta$ the set of full paths γ with the prefix δ. At each step of the random walk, we make a rough estimation of the complexity of computing the precise $|\mathbf{\Gamma}_\delta|$; we describe this estimation later in the section. If the estimate is low, we actually compute $|\mathbf{\Gamma}_\delta|$ using CTJ (as described in Section IV-B), and then our estimate is

$$C_{\mathsf{aj}}(\delta) := |\mathbf{\Gamma}_\delta| \times \prod_{i=1}^{\ell} d_i = \frac{|\mathbf{\Gamma}_\delta|}{\Pr(\delta)}\,.$$

Otherwise, we proceed exactly as in the case of Wander Join. In particular, if we cannot continue in the random walk, or reach a full path, then we use $C_{\mathsf{aj}}(\delta) :- C_{\mathsf{wj}}(\delta)$.

Example IV.3. We illustrate Audit Join (without distinct) by continuing our example over Figure 6. Suppose that after the random walk $\delta = (t_1^2, t_2^2)$, we choose to run an exact evaluation. Then $|\mathbf{\Gamma}_\delta| = 2$, since there are two full paths (ending at t_4^2 and t_4^3) that begin with δ. The estimate is then $C_{\mathsf{aj}}(\delta) = |\mathbf{\Gamma}_\delta|/\Pr(\delta) = 2 \cdot (5 \cdot 4) = 40$. □

The following proposition shows that C_{aj} is unbiased by straightforwardly adapting the argument for Wander Join.

Proposition IV.1. C_{aj} is an unbiased estimator of $|\mathbf{\Gamma}|$.

Proof. Let $\mathbf{\Delta}$ be the set of all paths δ where Audit Join decides to terminate the path and produce an estimate. This can be because δ is a full path, because it cannot proceed, or because we decide to compute the exact $|\mathbf{\Gamma}_\delta|$. The reader can verify that, no matter which of the three is the case, Audit Join produces the same estimator, namely $|\mathbf{\Gamma}_\delta|/\Pr(\delta)$. In particular, $|\mathbf{\Gamma}_\delta| = 1$ in the first case and $|\mathbf{\Gamma}_\delta| = 0$ in the second. We have the following.

$$\mathbb{E}[C_{\mathsf{aj}}] = \sum_{\delta \in \mathbf{\Delta}} \Pr(\delta) \cdot \frac{|\mathbf{\Gamma}_\delta|}{\Pr(\delta)} = \sum_{\delta \in \mathbf{\Delta}} |\mathbf{\Gamma}_\delta| = |\mathbf{\Gamma}|$$

Therefore, C_{aj} is unbiased, as claimed. □

Note that Audit Join automatically leverages the caching of CTJ, potentially avoiding re-computation when building the same prefix δ in later random walks. The reuse of the cached-prefix can dramatically increase the number of successful samples as shown in our experiments in Section V.

Distinct. We now extend our estimator to support the distinct operator. Recall the query of Figure 4. Our goal is to count the distinct values taken by β. For a full path γ, we denote by $\beta(\gamma)$ the value to which γ assigns β. Let $V = \{\beta(\gamma) \mid \gamma \in \mathbf{\Gamma}\}$. Our goal is to estimate $|V|$. For $b \in V$, we denote by $\Pr(b)$ the probability that the random walk reaches a full path γ with $\beta(\gamma) = b$; that is, $\Pr(b)$ is the sum of the probabilities of all $\gamma \in \mathbf{\Gamma}$ that assign b to β. Similarly, we denote by $\Pr(\delta, b)$ the probability that the random walk starts with δ and reaches a full path γ with $\beta(\gamma) = b$; that is, $\Pr(\delta, b)$ is the sum of the probabilities of all $\gamma \in \mathbf{\Gamma}$ such that δ is a prefix of γ and γ assigns b to β. We then combine these probabilities into the following estimator for distinct.

$$C_{\mathsf{aj}}^{\mathsf{d}}(\delta) := \sum_{b \in V} \frac{\Pr(\delta, b)}{\Pr(\delta) \cdot \Pr(b)} \tag{1}$$

Example IV.4. To demonstrate Audit Join with count distinct, we again use our running example over Figure 6. For this example, suppose that β occurs in G_3, and moreover, that each tuple t_3^i holds a unique value for β (while many join tuples may include t_3). Suppose that the random walk produces $\delta = (t_1^2, t_2^2)$, and that Audit Join decides to run an exact evaluation at this point. There are two full paths that extend δ, both through (t_2^3). We denote by b the value of β for (t_2^3). From the previous example we get that $\Pr(\delta) = \frac{1}{20}$. There are three paths leading to t_2^3, and by summing their probabilities we get $\Pr(b) = \frac{1}{5 \cdot 4 \cdot 3} + \frac{2}{5 \cdot 4 \cdot 4} = \frac{1}{24}$. The last probability of our estimator is $\Pr(\delta, b) = \Pr(b \mid \delta) \cdot \Pr(\delta) = \frac{1}{4} \cdot \frac{1}{20} = \frac{1}{80}$, Hence, our estimator yields the following.

$$C_{\mathsf{aj}}^{\mathsf{d}}(\delta) = \frac{\Pr(\delta, b)}{\Pr(\delta) \cdot \Pr(b)} = \frac{20 * 24}{80}$$

The estimate is, therefore, six: $C_{\mathsf{aj}}^{\mathsf{d}}(\delta) = 6$. □

In our implementation, the probability $\Pr(b)$ is computed online, after sampling the partial random path δ, by using CTJ to materialize all paths leading to the sampled $b = \beta(\delta)$, summing up their probabilities, and caching the results. With respect to Example IV.4, this is necessary to determine that there are three paths leading to t_2^3 and to compute their probabilities, which are then summed to compute $\Pr(b)$. Clearly this can be an expensive join query. Nevertheless, our experiments show that the online $\Pr(b)$ computation is very often tractable

AuditJoin$(G_1, \ldots, G_n, \alpha, \beta)$

```
1: N := 0
2: A := the set of possible assignments for α
3: B := the set of possible assignments for β
4: repeat
5:     F_1 := G_1
6:     δ := ϵ
7:     for i = 1, ..., n do
8:         N := N + 1
9:         select t ∈ F_i randomly and uniformly
10:        δ := (δ, t)
11:        if i = n or tipping point is reached then
12:            for all a ∈ A do
13:                C_a := C_a + Σ_{b∈B} Pr(a,b,δ) / (Pr(a,b)·Pr(δ))
14:            end for
15:            continue                          ▷ Go to line 5
16:        end if
17:        F_{i+1} := G_{i+1} ⋉ t
18:        if F_{i+1} = ∅ then
19:            continue                          ▷ Go to line 5
20:        end if
21:    end for
22: until time limit is reached
23: for all a ∈ A do
24:    estimate count-distinct for a as C_a/N
25: end for
```

Fig. 7: Audit Join pseudo code

after setting $\beta = b$, taking about 2.5 microseconds on average and up to 20 milliseconds in rare cases.

Next, we show that the estimator is unbiased.

Proposition IV.2. $C^{\mathsf{d}}_{\mathsf{aj}}$ is an unbiased estimator of $|V|$.

Proof. Following a similar reasoning as in the proof of Proposition IV.1, we can treat all cases of the estimator in a uniform way, that is, according to Equation (1). In the following analysis, we identify value $b \in V$ with the event that the random walk is complete and, moreover, assigns b to β. Hence, we have the following.

$$\begin{aligned}
\mathbb{E}[C^{\mathsf{d}}_{\mathsf{aj}}] &= \sum_{\delta \in \mathbf{\Delta}} \Pr(\delta) \cdot \sum_{b \in V} \frac{\Pr(\delta, b)}{\Pr(\delta) \cdot \Pr(b)} \\
&= \sum_{\delta \in \mathbf{\Delta}} \Pr(\delta) \cdot \frac{1}{\Pr(\delta)} \times \sum_{b \in V} \Pr(\delta \mid b) \\
&= \sum_{b \in V} \sum_{\delta \in \mathbf{\Delta}} \Pr(\delta \mid b) \\
&= \sum_{b \in V} 1 = |V|
\end{aligned}$$

Therefore, $C^{\mathsf{d}}_{\mathsf{aj}}$ is unbiased, as claimed. □

Audit Join can then use the same formulas as Wander Join [59] to provide a confidence interval using the Audit Join estimator.

Tipping Point. To decide when to use partial exact computations, we use the same simple technique for join-size estimation as used by PostgreSQL [69]. In the case of two triple patterns (a_1, b_1, c_1) and (a_2, b_2, c_2) joining on say $c_1 = c_2$, the size is estimated as the product between the number of triples matched by (a_1, b_1, c_1) and (a_2, b_2, c_2), divided by the maximum number of distinct terms of c_1 or c_2. For more than two patterns, we compose the estimates in the straightforward manner. If the estimate is lower than a predefined threshold, Audit Join switches to exact computation. In this case we say that the *tipping point* is reached. Though simple, this technique allows Audit Join to consistently achieve considerable improvements, as shown in the experimental section. Investigating more sophisticated estimates (e.g., [70]) is left for future research.

Summary. We summarize the Audit Join algorithm in Figure 7. The code is similar to the process described in Section IV-D, except that it incorporates grouping. The sets A and B are projections over the attributes α and β, respectively (Figure 4). The estimates are accumulated in C_a for every group a. The probability $\Pr(a, b, \delta)$ corresponds to the event that the random walk starts with δ and includes the group a and the counted value b. Hence, it is the sum of the probabilities of all such random walks. Similarly, $\Pr(a, b)$ is the probability that the random walk includes the group a and the counted value b. Note that in line 17, the left semi-join $G_{i+1} \ltimes t$ consists of all the tuples of G_{i+1} that can be matched with t. Finally, on line 24, the number of distinct results for each group a is computed by dividing C_a by the number of random walks N.

Limitations. Our focus is on supporting exploration queries of the form illustrated in Figure 4, where a variable appears in at most two triple patterns (and constants can appear anywhere). Like WJ, the AJ algorithm is based on random walks and could utilize similar methods to support online aggregation for cyclic queries, while following CTJ's caching policies. The AJ algorithm does not, however, currently support other forms of query operators, such as negation (anti-joins), optionals (left-joins), etc.; or other forms of aggregation, such as sum, average, etc. Extending the AJ algorithm to support such queries is an interesting direction for future work.

V. Experimental Study

Our experimental study compares the performance of four query engine strategies – Virtuoso, Cached Trie Join (CTJ), Wander Join (WJ) and Audit Join (AJ) – for answering a variety of queries in the context of our use-case exploration system. Though Virtuoso and CTJ both compute exact results, and thus are somewhat incomparable with WJ and AJ, we include their results primarily to motivate the need for online aggregation in the context of knowledge graph exploration. The focus of our experiments is rather to compare WJ and AJ, which are both online aggregation algorithms.

A. Configuration

We take Virtuoso as an off-the-shelf query engine and implement the other three strategies in C++. We implement WJ and CTJ as described in their corresponding papers and then implement AJ on top of these algorithms. We now discuss the configuration and implementation of each strategy.

We use Virtuoso v. 07.20.3217 [56] and configure it to use all available memory for caching its indexes. Specifically, we maintain the index orders (s, p, o), (o, p, s), (p, s, o), and (p, o, s); these orders are sufficient to support our exploration queries. Virtuoso runs the subclass closure for expansions using property paths on the original graph; the query planner shows that this closure is executed first and takes a few milliseconds for both knowledge bases.

Our CTJ implementation uses LFTJ [23] as the trie join algorithm. Since WJ does not support transitivity, the subclass closure is computed offline and materialized in the graph (instances are indexed with only their explicit types per the original data). The trie indexes are implemented using sorted arrays (*std::vector*) such that each search is done in $O(\log(n))$. Similarly to Virtuoso, we implement CTJ with four of the six possible index orders: (s, p, o), (o, p, s), (p, s, o), and (p, o, s). The CTJ cache structure uses an array of hashtables (*std::unordered_map*[]).

In WJ, the graph is saved in an unsorted array (*std::vector*). Analogously to CTJ, the subclass closure is performed offline and materialized. The algorithm uses hashtable indexes (*std::unordered_map*) over the array that enable sampling in $O(1)$ time. In the case of distinct – as there is no formal support for this operator in WJ – we augment it with the technique proposed by Haas et al. in Ripple Join [28], [63] for performing online aggregation in the distinct case: this technique stores the set of samples seen thus far and rejects new samples that have already been seen.

AJ is implemented on top of WJ and CTJ; it likewise assumes that the subclass closure has been materialized. Our implementation uses a hybrid hashtable/trie data-structure where the hashtable indexes point to a sorted array, allowing $O(1)$-time sampling for WJ and $O(log(n))$-time search for CTJ. Analogously to CTJ, AJ maintains four index orders and the same caching structure. Thus, all systems use a similar amount of memory which fits into the memory of our server. Specifically, the indexes for all systems use 72GB and 194GB of memory for the DBpedia and LinkedGeoData KGs (described below), respectively. We note that that much larger KGs may not fit into memory, and leave the investigation of index compression and paging techniques for future work. Unlike WJ, AJ uses its own unbiased estimator for distinct (per Section IV).

B. Methodology

Data. Our data include two large-scale knowledge graphs: DBpedia [1] and LinkedGeoData [3]; details of these two graphs are presented in Table I. DBpedia contains multi-domain data extracted from Wikimedia projects such as Wikipedia; we take the English version of DBpedia v.3.6, containing more than 400 million RDF triples, 370 thousand classes and 61 thousand properties. LinkedGeoData specializes in spatial data extracted from OpenStreetMap; we take the November 2015 version, containing more than 1.2 billion RDF triples, one thousand classes, and 33 thousand properties. In the case of LinkedGeoData, since no root class is defined in the original data, we explicitly add a class that is the parent of all classes previously without a parent in the class hierarchy.

TABLE I: Dataset information

Dataset	Version	Size	Triples	Classes	Props
DBpedia	3.6	4.9 GB	432M	370,082	61,944
LGD	2015-11	14.0 GB	1,217M	1,147	33,355

Queries. Our queries consist of randomly generated exploration paths that imitate users applying incremental expansions. More specifically, our generator starts with the root class of a graph. At each step, the generator uniformly selects one of the expansion operations, which is translated to a SPARQL query of the form shown in Figure 4. Next, one of the groups (aka. bar) from the answer is randomly sampled; we apply a weighted sampling according to the size of the group to increase focus on large groups (otherwise since the majority of groups are small, the explorations would fixate on small groups). The generator continues for four steps or until it gets an empty result. Queries with empty results are ignored and not considered part of the path. We ran this generator 25 times for each graph resulting in a total of 50 distinct non-empty queries. The error is computed as the absolute difference between the exact count and estimated count divided by the exact result; consequently, the reported absolute mean error (MAE) is the average error over all groups in the result. For each query, we define the selectivity of a query similarly to Wander Join [66]:

$$1 - \frac{\text{join size including filters}}{\text{join size without filters}}.$$

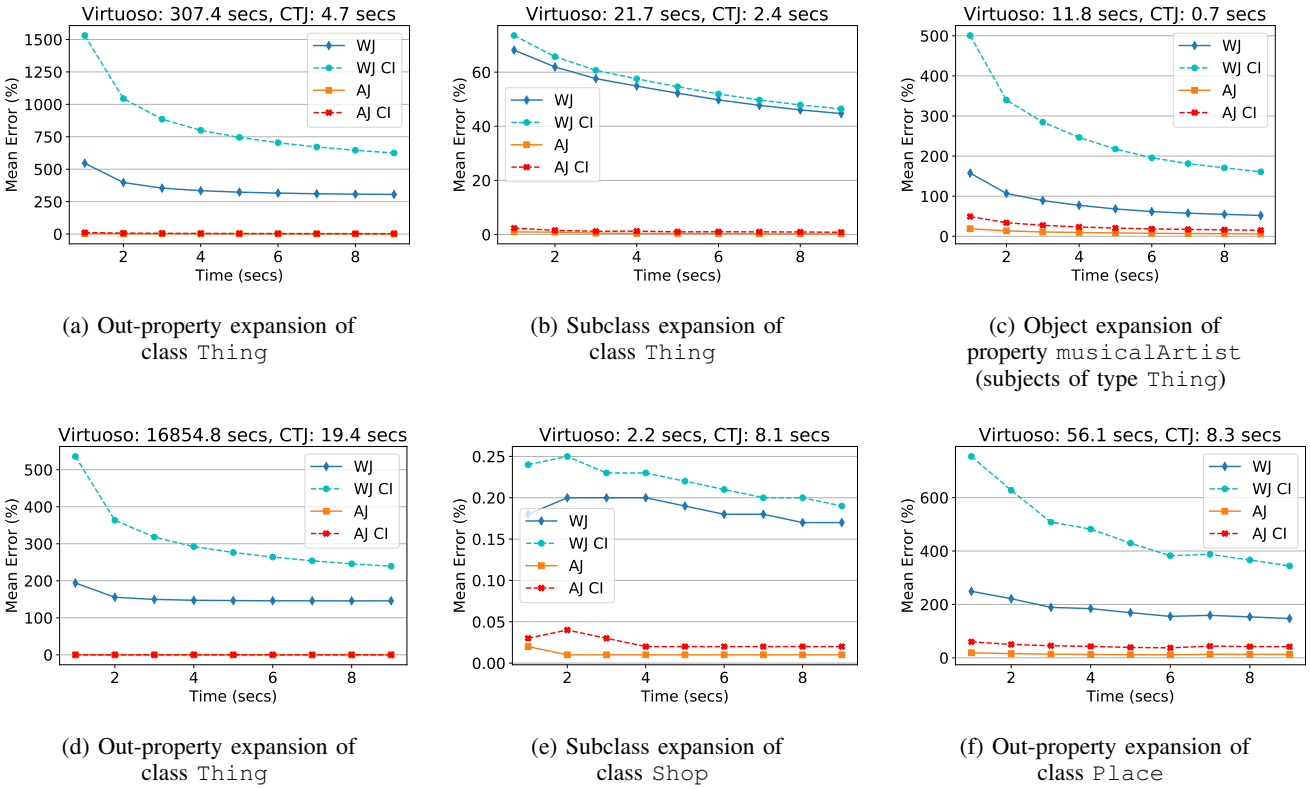
In our setting, each filter sets a variable in a query to a constant. Since each group has its own estimator, we compute the selectivity of each group separately and average all groups.

Machine and Testing Protocol. Our server has four 2.1 GHz Xeon E5-2683 v4 processors, 500 GB of DDR4 DRAM, and runs Ubuntu 16.04.4 Linux. Each experiment was performed three times, and the average runtime and mean absolute error (MAE) are reported. MAE will simply be referred to as mean error. We run each online aggregation algorithm for nine seconds and report the estimate after each second. For each query, we tested different join orders of WJ and selected the one with the best MAE.

C. Results

We first present results on a selection of six queries to illustrate different behaviour in all four compared approaches. We then compare the error observed over time for the two online aggregation approaches over all queries, contrasting different exploration depths, the two different datasets, and queries with and without distinct.

Selected queries with distinct. Figure 8 presents the results for a selection of six queries (with distinct) that illustrate a variety of behaviours in the compared approaches. Each graph



(a) Out-property expansion of class `Thing`

(b) Subclass expansion of class `Thing`

(c) Object expansion of property `musicalArtist` (subjects of type `Thing`)

(d) Out-property expansion of class `Thing`

(e) Subclass expansion of class `Shop`

(f) Out-property expansion of class `Place`

Fig. 8: The MAE per second of the estimators (WJ and AJ) and 0.95 confidence intervals (WJ CI and AJ CI), as well as exact runtimes (Virtuoso and CTJ) for a selection of exploration queries: from DBpedia (top) and from LinkedGeoData (bottom)

presents the mean error over time, in seconds, for a specific exploration query. The times for the exact engines, specifically Virtuoso and CTJ, are indicated above the graph. The top row presents results over DBpedia, while the bottom row presents results over LinkedGeoData. The WJ CI and AJ CI approaches assume a 0.95 confidence interval.

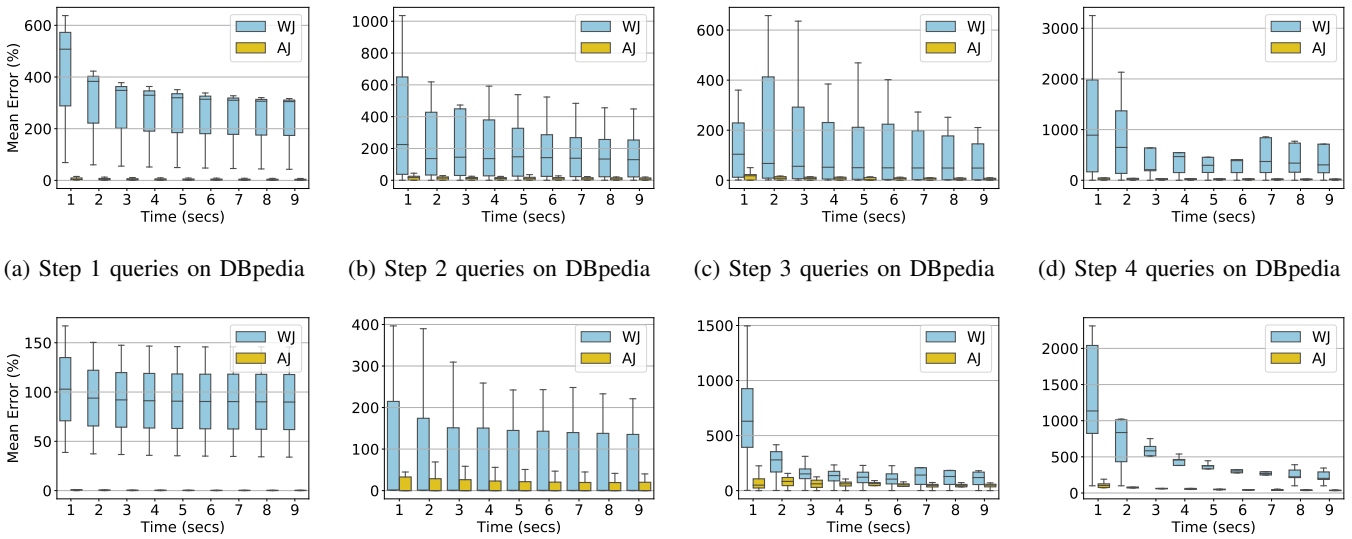
Virtuoso and CTJ take more than a second to answer most queries and take notably longer for the larger dataset: LinkedGeoData (with three times more edges). Focusing on Virtuoso, we see that while some queries run in the order of seconds, others run in minutes or even hours: the out-property expansion of `Thing` (the root class) on LinkedGeoData runs for more than four hours (see Figure 8d). On the other hand, CTJ generally offers a major performance improvement over Virtuoso: though slower in one case (see Figure 8e), in the worst cases, CTJ returns results in tens of seconds; for example, the query that took Virtuoso over 4 hours takes CTJ around 20 seconds. Still, even with the considerable performance improvements offered by CTJ, runtimes in the order of tens of seconds would hurt the interactivity and usability of our system.

When comparing the mean errors of WJ and AJ over time in Figure 8, it is clear that the accuracy and convergence of AJ are considerably higher than WJ for the selected queries. These improvements are due to the two extended features of AJ: the reduction of rejection rates using CTJ, and the addition of an unbiased estimator for the distinct case (we shall test without distinct in later experiments). We now discuss some individual cases in more detail.

Looking first at DBpedia, for the out-property expansion of `Thing` (Figure 8a), the mean error of WJ is 519% after one second and 303% after nine seconds; the corresponding errors for AJ are 7.5% and 3.7%, respectively—almost two orders of magnitude improvement compared to WJ. The drastic improvement is mainly because AJ uses partial exact aggregations that more accurately estimate the large number of groups in the query (more than 52,000); these groups have a high per-group selectivity averaging very close to 1. Consequently, while the sample rejection rate of WJ is 86% for this query, AJ rejection rate is only 6% (14× lower).

In the object expansion of `musicalArtist` (Figure 8c), which originates from the subclass expansion of `Thing` (Figure 8b), WJ has a mean error of 163% and 53% after one and nine seconds, respectively. While still better than WJ, the accuracy of AJ drops, starting at a mean error of 24% and reaching 9% (on the other hand, given the high selectivity of the full join query, namely 0.996, CTJ returns exact results in less than a second).

Moving to the second row of plots in Figure 8, while the larger scale of LinkedGeoData notably affects Virtuoso and CTJ, the results for online aggregation remain similar to DBpedia. Figure 8d shows that WJ's estimation of the out-property expansion on `Thing` results in a mean error of 194% after one second, which slowly drops to 145% after



(a) Step 1 queries on DBpedia (b) Step 2 queries on DBpedia (c) Step 3 queries on DBpedia (d) Step 4 queries on DBpedia

(e) Step 1 queries on LinkedGeoData (f) Step 2 queries on LinkedGeoData (g) Step 3 queries on LinkedGeoData (h) Step 4 queries on LinkedGeoData

Fig. 9: MAE over time of all queries with a varying number of exploration steps (WJ left; AJ right)

nine seconds; for the same query, AJ estimates the results with a mean error close to 0% from the first second. The selectivity of the out-property expansion query is 0.88, while the average per-group selectivity is 0.9998. For this query, AJ reduces the rejection rate by 3.2× and increases the number of successful samples by 2.6× due to the reuse of the CTJ caches for recurrent samples.

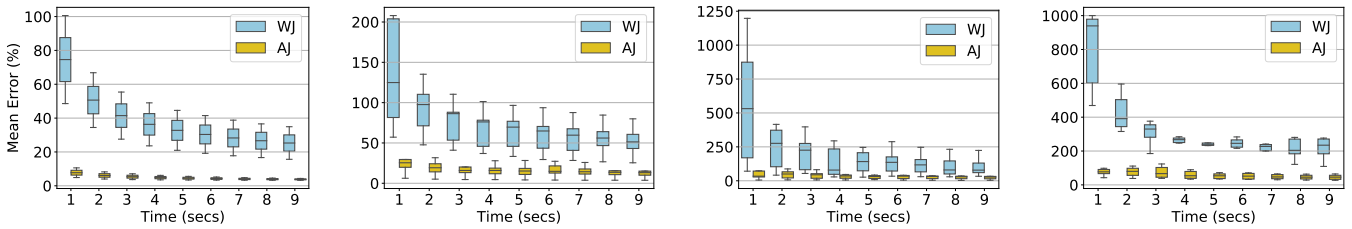
The subclass expansion on `Shop` (Figure 8e) is quickly well-estimated by both WJ and AJ, resulting in a mean error of 0.17% and 0.04% after one second, respectively. This is mostly due to a small number of groups with a low per-group selectivity. Though estimates worsen in the third query for an out-property expansion of `Place` (Figure 8f), AJ still outperforms WJ in this case.

All queries with distinct. For each exploration depth and knowledge graph, Figure 9 presents the range of mean errors for all estimates under the distinct operator as box plots (specifically, Tukey plots), displaying the interquartile range of error (the box), the median error (the inner line), and the most extreme value within 1.5× the interquartile range (the whiskers). These plots show the variation of mean error across all queries over time, where we see that AJ is consistently better than WJ over all randomly generated queries: the median of mean errors for WJ in some cases reaches over 1,000% after one second and 300% after nine seconds (see Figure 9d); on the other hand, the same median errors for AJ are at worst 104% after one second, and 50% after nine seconds.

An interesting result can be found when comparing the graphs for a varying numbers of exploration steps. Taking LinkedGeoData, for example, the accuracy of WJ drops when comparing the first step (Figure 9e) with subsequent steps (Figures 9f–9h resp.). A similar trend can be seen for DBpedia (though less clear moving from step one to two). We attribute this trend to *(1)* an increasing number of rejections: later steps tend to become more specific, adding more selective joins; *and (2)* increasing duplicates: as the query becomes larger, more variables are projected away. Both issues are specifically addressed by the AJ algorithm. With respect to point (1), Figure 11 presents the rejection rate for each query sorted by the rejection rate. It shows that the rejection rate of AJ is much lower than that of WJ. For example, AJ offers a rejection rate of less than 25% for 28 queries, while WJ for only 9 queries.

Our results show that the average sample time of both algorithms is comparable, at about 2.5 microseconds: while the AJ estimator is sometimes slower to compute (reaching a maximum sample time of 20 milliseconds), the use of caching offsets AJ's slower estimations, resulting in comparable sample times with WJ.

All queries without distinct. Though our exploration system requires the distinct operator for counts, we also perform experiments in the non-distinct case to understand the relative impact of the unbiased distinct estimator and the partial exact computations on reducing error in AJ. Figure 10 presents Tukey plots of mean error over time for all queries without distinct, and both datasets, separated by the number of exploration steps. Overall we conclude that: *(1)* the error observed for WJ drops from the distinct case to the non-distinct case; *(2)* the errors generally increase in AJ versus the distinct case; *and (3)* AJ continues to significantly outperform WJ, though by a lesser margin when considering the non-distinct case. We surmise that though AJ no longer benefits from the unbiased distinct estimator, it still outperforms WJ in the non-distinct case due to the partial exact computations; hence the benefits



(a) Step 1 queries (b) Step 2 queries (c) Step 3 queries (d) Step 4 queries

Fig. 10: MAE over time in seconds for queries without the distinct operator (WJ left; AJ right)

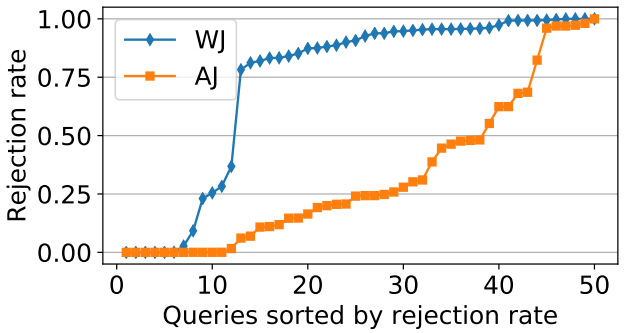


Fig. 11: Rejection rate of AJ and WJ on each query

of AJ are not only due to the unbiased distinct estimator.

Summary. With respect to the traditional approach of computing exact results, exploration queries take in the order of hours for Virtuoso, and in the order of tens of seconds for CTJ, with both approaches slowing down for the larger LinkedGeoData graph when compared with DBpedia. Computing exact counts (with either approach) is not compatible with our goal of interactive runtimes while exploring large-scale knowledge graphs.

With respect to comparing online aggregation methods (WJ vs. AJ), we find that AJ significantly reduces error (by orders of magnitude) versus WJ over the same time period, and in most cases can quickly converge to estimates with less than 1% mean error. We also find that the relative benefit of AJ and WJ improves as more exploration steps are added. Through experiments on non-distinct queries, we find that AJ continues to significantly reduce error versus WJ due to its inclusion of CTJ for partial exact computations.

Overall we find that online aggregation is a viable alternative for interactively exploring large-scale knowledge graphs, for which our proposed AJ algorithm – using partial exact computations and an unbiased distinct estimator – significantly reduces error.

VI. Conclusions

We presented an efficient implementation of a system for exploring large knowledge graphs. The system utilizes algorithms that specialize in the exploration queries of our system. Our queries are based on a formal framework for the visual exploration of knowledge graphs, where an exploration step consists of the transformation of the bar of one bar chart into the next bar chart. We investigated the implementation of this framework using various query engines, including a SPARQL engine, a recent in-memory join algorithm, and online aggregation. Finally, as our main contribution, we devised and analyzed Audit Join: a specialized online-aggregation algorithm that combines the random walks of Wander Join with exact computation, and extends its estimator to accommodate the count aggregations under the distinct operator. Our experiments show that the runtimes of both methods for performing exact counts are too slow for supporting interactive exploration over large-scale knowledge graphs. Focusing thereafter on online aggregation, we find that when compared with Wander Join, Audit Join significantly reduces error with respect to time in all experiments, often by orders of magnitude, including both distinct and non-distinct cases. This we believe that Audit Join is a promising approach for powering the exploration of large-scale knowledge graphs.

Looking to the future, we see two principal lines of research.

The first line relates to further improving AuditJoin. For one, we wish to arrive at a more general understanding of how partial exact computation complements online aggregation in order to devise a more principled way to combine both; our results, along with those of Zhao et al. [30], show this to be a very promising approach in general, and more work can be done to refine this idea. We will also explore the application of our approach to general join queries, beyond the acyclic queries produced by our system, as well as support for other types of query operators and aggregation functions.

The second line of research relates to exploring and further developing use-cases for AuditJoin. We wish to explore directions for improving the usability of our exploration system based on AuditJoin. Some envisaged extensions include: allowing users to explore and contrast multiple knowledge graphs simultaneously, adding support for incremental indexing on updates, extending filtering capabilities, and adding support for further semantics beyond subclass closure. We are also interested in exploring other use-cases for AuditJoin in the context of scenarios requiring efficient cardinality estimations over large-scale knowledge graphs.

REFERENCES

[1] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann, "DBpedia - a crystallization point for the web of data," *Web Semantics*, vol. 7, no. 3, pp. 154–165, Sep. 2009.

[2] K. D. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2008, pp. 1247–1250.

[3] S. Auer, J. Lehmann, and S. Hellmann, "Linkedgeodata: Adding a spatial dimension to the web of data," in *The Semantic Web - ISWC 2009*, A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta, and K. Thirunarayan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 731–746.

[4] D. Vrandecic and M. Krötzsch, "Wikidata: a free collaborative knowledgebase," *Commun. ACM*, 2014.

[5] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum, "YAGO2: A spatially and temporally enhanced knowledge base from wikipedia," *Artif. Intell.*, vol. 194, pp. 28–61, 2013.

[6] N. F. Noy, Y. Gao, A. Jain, A. Narayanan, A. Patterson, and J. Taylor, "Industry-scale knowledge graphs: lessons and challenges," *Commun. ACM*, vol. 62, no. 8, pp. 36–43, 2019.

[7] M. B. Ellefi, Z. Bellahsene, J. Breslin, E. Demidova, S. Dietze, J. Szymanski, and K. Todorov, "RDF Dataset Profiling – a Survey of Features, Methods, Vocabularies and Applications," *Semantic Web*, 2018.

[8] S. Auer, J. Demter, M. Martin, and J. Lehmann, "LODStats - An Extensible Framework for High-Performance Dataset Analytics," in *Knowledge Engineering and Knowledge Management (EKAW)*, 2012.

[9] Z. Abedjan, T. Grütze, A. Jentzsch, and F. Naumann, "Profiling and mining RDF data with ProLOD++," in *International Conference on Data Engineering (ICDE)*, 2014.

[10] N. Bikakis, G. Papastefanatos, M. Skourla, and T. Sellis, "A hierarchical aggregation framework for efficient multilevel visual exploration and analysis," *Semantic Web*, vol. 8, no. 1, pp. 139–179, 2017.

[11] R. A. A. Principe, B. Spahiu, M. Palmonari, A. Rula, F. D. Paoli, and A. Maurino, "ABSTAT 1.0: Compute, Manage and Share Semantic Profiles of RDF Knowledge Graphs," in *European Semantic Web Conference (ESWC)*, 2018, pp. 170–175.

[12] T. Yahav, O. Kalinsky, O. Mishali, and B. Kimelfeld, "eLinda: Explorer for Linked Data," in *Extending Database Technology (EDBT)*, 2018.

[13] J. Moreno-Vega and A. Hogan, "GraFa: Scalable Faceted Browsing for RDF Graphs," in *International Semantic Web Conference (ISWC)*, 2018.

[14] T. Neumann and G. Weikum, "Rdf-3x: A risc-style engine for rdf," *Proc. VLDB Endow.*, pp. 647–659, Aug. 2008.

[15] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov, "Owlim: A family of scalable semantic repositories," *Semant. web*, vol. 2, no. 1, pp. 33–42, Jan. 2011.

[16] B. B. Thompson, M. Personick, and M. Cutcher, "The bigdata® RDF graph database," in *Linked Data Management*. CRC Press, 2014, pp. 193–237.

[17] J. M. Brunetti, R. G. González, and S. Auer, "From overview to facets and pivoting for interactive exploration of Semantic Web data," *IJSWIS*, vol. 9, no. 1, pp. 1–20, 2013.

[18] H. Bast and B. Buchhold, "An index for efficient semantic full-text search," in *Conference on Information and Knowledge Management (CIKM)*, 2013.

[19] O. Erling, "Virtuoso, a Hybrid RDBMS/Graph Column Store," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 3–8, 2012.

[20] BSBM, "Berlin sparql benchmark," http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/.

[21] C. Bizer and A. Schultz, "The berlin SPARQL benchmark," *Int. J. Semantic Web Inf. Syst.*, vol. 5, no. 2, pp. 1–24, 2009.

[22] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, "Diversified stress testing of RDF data management systems," in *International Semantic Web Conference (ISWC)*, 2014, pp. 197–212.

[23] T. L. Veldhuizen, "Triejoin: A simple, worst-case optimal join algorithm," in *ICDT*, 2014, pp. 96–106.

[24] M. Abo Khamis, H. Q. Ngo, and A. Rudra, "Faq: Questions asked frequently," in *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS '16. New York, NY, USA: ACM, 2016, pp. 13–28.

[25] H. Q. Ngo, D. T. Nguyen, C. Re, and A. Rudra, "Beyond worst-case analysis for joins with minesweeper," in *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '14. New York, NY, USA: ACM, 2014, pp. 234–245.

[26] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, "Emptyheaded: A relational engine for graph processing," *ACM Trans. Database Syst.*, 2017.

[27] O. Kalinsky, Y. Etsion, and B. Kimelfeld, "Flexible caching in trie joins," in *EDBT*, 2017, pp. 282–293.

[28] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online aggregation," in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '97. New York, NY, USA: ACM, 1997, pp. 171–182.

[29] F. Li, B. Wu, K. Yi, and Z. Zhao, "Wander join: Online aggregation via random walks," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016*, 2016.

[30] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi, "Random sampling over joins revisited," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. ACM, 2018, pp. 1525–1539.

[31] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer, "DBpedia – A large-scale, multilingual knowledge base extracted from Wikipedia," *Semantic Web*, vol. 6, no. 2, pp. 167–195, 2015.

[32] A.-S. Dadzie and M. Rowe, "Approaches to visualising linked data: A survey," *Semantic Web*, vol. 2, no. 2, pp. 89–124, 2011.

[33] Y. Tzitzikas, N. Manolis, and P. Papadakos, "Faceted exploration of RDF/S datasets: a survey," *J. Intell. Inf. Syst.*, vol. 48, no. 2, pp. 329–364, 2017.

[34] m. schraefel, M. Wilson, A. Russell, and D. A. Smith, "mSpace: improving information access to multimedia domains with multimodal exploratory search," *Commun. ACM*, vol. 49, no. 4, pp. 47–49, 2006.

[35] E. Oren, R. Delbru, and S. Decker, "Extending faceted navigation for RDF data," in *International Semantic Web Conference (ISWC)*, 2006, pp. 559–572.

[36] M. Hildebrand, J. van Ossenbruggen, and L. Hardman, "/facet: A browser for heterogeneous Semantic Web repositories," in *International Semantic Web Conference (ISWC)*, 2006, pp. 272–285.

[37] E. Mäkelä, E. Hyvönen, and S. Saarela, "Ontogator - A semantic view-based search engine service for web applications," in *International Semantic Web Conference (ISWC)*, 2006, pp. 847–860.

[38] M. R. Kamdar, D. Zeginis, A. Hasnain, S. Decker, and H. F. Deus, "Reveald: A user-driven domain-specific interactive search platform for biomedical research," *Journal of Biomedical Informatics*, vol. 47, pp. 112–130, 2014.

[39] Y. Tzitzikas, N. Bailly, P. Papadakos, N. Minadakis, and G. Nikitakis, "Using preference-enriched faceted search for species identification," *IJMSO*, vol. 11, no. 3, pp. 165–179, 2016.

[40] R. Hahn, C. Bizer, C. Sahnwaldt, C. Herta, S. Robinson, M. Bürgle, H. Düwiger, and U. Scheel, "Faceted Wikipedia Search," in *Business Information Systems (BIS)*, 2010, pp. 1–11.

[41] M. Arenas, B. C. Grau, E. Kharlamov, S. Marciuska, and D. Zheleznyakov, "Faceted search over RDF-based knowledge graphs," *J. Web Sem.*, vol. 37-38, pp. 55–74, 2016.

[42] H. Wang, Q. Liu, T. Penin, L. Fu, L. Zhang, T. Tran, Y. Yu, and Y. Pan, "Semplore: A scalable IR approach to search the web of data," *J. Web Sem.*, vol. 7, no. 3, pp. 177–188, 2009.

[43] S. Ferré, "Expressive and scalable query-based faceted search over SPARQL endpoints," in *International Semantic Web Conference (ISWC)*, 2014, pp. 438–453.

[44] D. F. Huynh and D. R. Karger, "Parallax and companion: Set-based browsing for the data web," Technical Report, http://davidhuynh.net/media/papers/2009/www2009-parallax.pdf.

[45] A. Wagner, G. Ladwig, and T. Tran, "Browsing-oriented semantic faceted search," in *Database and Expert Systems Applications (DEXA)*, 2011.

[46] P. Heim, T. Ertl, and J. Ziegler, "Facet graphs: Complex semantic querying made easy," in *The Semantic Web (ESWC)*, 2010, pp. 288–302.

[47] Š. Čebirić, F. Goasdoué, and I. Manolescu, "Query-oriented summarization of RDF graphs," *PVLDB*, vol. 8, no. 12, pp. 2012–2015, 2015.

[48] M. P. Consens, V. Fionda, S. Khatchadourian, and G. Pirrò, "S+EPPs: Construct and Explore Bisimulation Summaries, plus Optimize Navigational Queries; all on Existing SPARQL Systems," *PVLDB*, 2015.

[49] M. Kirchberg, E. Leonardi, Y. S. Tan, S. Link, R. K. L. Ko, and B. Lee, "Formal Concept Discovery in Semantic Web Data," in *International Conference on Formal Concept Analysis (ICFCA)*. Springer, 2012, pp. 164–179.

[50] M. Rouane-Hacene, M. Huchard, A. Napoli, and P. Valtchev, "Relational Concept Analysis: mining concept lattices from multi-relational data," *Ann. Math. Artif. Intell.*, vol. 67, no. 1, pp. 81–108, 2013.

[51] S. Kinsella, U. Bojars, A. Harth, J. G. Breslin, and S. Decker, "An interactive map of semantic web ontology usage," in *International Conference on Information Visualisation*, 2008, pp. 179–184.

[52] S. Campinas, T. Perry, D. Ceccarelli, R. Delbru, and G. Tummarello, "Introducing RDF graph summary with application to assisted SPARQL formulation," in *Database and Expert Systems Applications Workshop (DEXA)*. IEEE Computer Society, 2012, pp. 261–266.

[53] M. Dudás, V. Svátek, and J. Mynarz, "Dataset summary visualization with lodsight," in *Extended Semantic Web Conference (ESWC) – Demo*. Springer, 2015, pp. 36–40.

[54] F. Florenzano, D. Parra, J. L. Reutter, and F. Venegas, "A Visual Aide for Understanding Endpoint Data," in *International Workshop on Visualization and Interaction for Ontologies and Linked Data (VOILA@ISWC)*. CEUR-WS.org, 2016, pp. 102–113.

[55] "Sparql," https://www.w3.org/TR/sparql11-overview/.

[56] "Openlink Virtuoso," https://virtuoso.openlinksw.com/.

[57] A. Atserias, M. Grohe, and D. Marx, "Size bounds and query plans for relational joins," in *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science*, ser. FOCS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 739–748.

[58] D. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra, "Join processing for graph patterns: An old dog with new tricks," in *Proceedings of the GRADES'15*. New York, NY, USA: ACM, 2015, pp. 2:1–2:8.

[59] P. J. Haas, "Large-sample and deterministic confidence intervals for online aggregation," in *Proceedings of the Ninth International Conference on Scientific and Statistical Database Management*, ser. SSDBM '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 51–63.

[60] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie, "Online aggregation for large mapreduce jobs," *PVLDB*, 2011.

[61] C. Qin and F. Rusu, "Parallel online aggregation in action," in *Conference on Scientific and Statistical Database Management, SSDBM '13, Baltimore, MD, USA, July 29 - 31, 2013*, 2013, pp. 46:1–46:4.

[62] ——, "PF-OLA: a high-performance framework for parallel online aggregation," *Distributed and Parallel Databases*, vol. 32, no. 3, pp. 337–375, 2014.

[63] P. J. Haas and J. M. Hellerstein, "Ripple joins for online aggregation," in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '99. ACM, 1999, pp. 287–298.

[64] A. Galakatos, A. Crotty, E. Zgraggen, C. Binnig, and T. Kraska, "Revisiting reuse for approximate query processing," *PVLDB*, vol. 10, no. 10, pp. 1142–1153, 2017.

[65] Y. Chen and K. Yi, "Two-level sampling for join size estimation," in *SIGMOD Conference*. ACM, 2017, pp. 759–774.

[66] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case optimal join algorithms," *J. ACM*, vol. 65, no. 3, pp. 16:1–16:40, 2018.

[67] D. T. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra, "Join processing for graph patterns: An old dog with new tricks," in *GRADES@SIGMOD/PODS*. ACM, 2015, pp. 2:1–2:8.

[68] D. G. Horvitz and D. J. Thompson, "A generalization of sampling without replacement from a finite universe," *Journal of the American statistical Association*, vol. 47, no. 260, pp. 663–685, 1952.

[69] PostgreSQL, "How the planner uses statistics," https://www.postgresql.org/docs/current/static/planner-stats-details.html.

[70] D. Vengerov, A. C. Menck, M. Zaït, and S. Chakkappen, "Join size estimation subject to filter conditions," *PVLDB*, vol. 8, no. 12, pp. 1530–1541, 2015.