Inferray: fast in-memory RDF inference

Julien Subercaze, Christophe Gravier
Jules Chevalier, Frederique Laforest
Laboratoire Hubert Curien, UMR CNRS 5516
Université Jean Monnet
25 rue docteur Rémy Annino
F-42000, Saint-Etienne, France

ABSTRACT

The advent of semantic data on the Web requires efficient reasoning systems to infer RDF and OWL data. The linked nature and the huge volume of data entail efficiency and scalability challenges when designing productive inference systems. This paper presents Inferray, an implementation of RDFS, ρdf, and RDFS-Plus inference with improved performance over existing solutions. The main features of Inferray are 1) a storage layout based on vertical partitioning that guarantees sequential access and efficient sort-merge join inference; 2) efficient sorting of pairs of 64-bit integers using ad-hoc optimizations on MSD radix and a custom counting sort; 3) a dedicated temporary storage to perform efficient graph closure computation. Our measurements on synthetic and real-world datasets show improvements over competitors on RDFS-Plus, and up to several orders of magnitude for transitivity closure.

1. INTRODUCTION

The Resource Description Framework (RDF) is a graph-based data model accepted as the W3C standard in February 1999, outlining the core data model for Semantic Web applications. At a high-level, the Semantic Web can be conceptualized as an extension of the current Web to enable the creation, sharing and intelligent re-use of machine-readable content on the Web. The adoption of RDF has broadened on the Web in recent years and the number of triples in the Linked Open Data (LOD) cloud has topped 60 billions.

The elemental constituents of the RDF data model are RDF terms that are used in reference to resources. The set of RDF terms is broken down into three disjoint subsets: URIs (or IRIs), literals and blank nodes. These RDF terms constitute the core of the Semantic Web named as RDF triple, which is a 3-tuple of RDF terms, and is commonly denoted by $\langle s, p, o\rangle$ for $\langle subject, property, object\rangle$. Each RDF triple represents an atomic "fact" or "claim", where the labeled *property* (must be a URI), also called a predicate, describes the relationship between the *subject* (either a URI or a blank node) and the *object* (a URI, blank node or literal). It is common practice to conceptualize RDF datasets as directed labeled graphs, where subjects and objects represent labeled vertices and properties describe directed, labeled edges. For instance, the following triple entails that humans belong to a larger group called mammal:

```
<human, rdfs:subClassOf, mammal>
```

There has been a growing interest in the database community to design efficient Semantic Web databases known as triple stores. The aim of triple stores is to handle potentially large volumes of RDF data and to support SPARQL, a mature, feature-rich query language for RDF content. Several novel triple store design principles have been suggested in the literature, such as vertical partitioning [1], horizontal partitioning [13], direct primary hashing [6], multiple indexing [33], compactness of data [34], and efficient query implementation [18, 19].

An inference layer coupled with a triple store provides an interesting feature over aggregated Web data. Ontology Based Data Access (OBDA) systems, usually support inference [24]. Loosely speaking, inference utilizes the formal underlying semantics of the terms in the data to enable the derivation of new knowledge. It is used to assert equalities between equivalent resources, flag inconsistencies where conflicting data are provided, execute mappings between different data models concerned with the same domain. For instance, utilizing the following triple with the previous one,

```
<mammal, rdfs:subClassOf, animal>
```

and considering that the property `rdfs:subClassOf` is transitive, we can infer the following new triple:

```
<human, rdfs:subClassOf, animal>
```

Inference is specified within a family of recommendations by the W3C, ranging from simple entailment regime like RDFS to more complex ones like OWL Full [29]. However, reasoning over RDF data is an ambitious goal with many inherent difficulties, especially the time complexity of entailment. This complexity ranges from P to NEXPTIME [29], and the OWL 2 Full language [30] is even proven undecidable.

While these theoretical results may seem discouraging for database practitioners, it is important to note that special

Proceedings of the VLDB Endowment, Vol. 9, No. 6

efforts have been made to specify *working* logics for real-world systems like simplified RDFS [31], ρdf, and RDFS-Plus. RDFS is the original specification. But in practice, systems usually perform incomplete RDFS reasoning and consider only rules whose antecedents are made of two-way joins [11]. This is fairly common in practice, as single-antecedent rules derive triples that do not convey interesting knowledge, but satisfy the logician. Hence, inference engines such as Jena [16] and OWLIM [4] propose using lightweight flavors of RDFS. ρdf is another common meaningful subset of RDFS, designed to contain the *essential* semantics of RDFS while reducing the number of rules. RDFS-Plus [2] was conceived to provide a framework that allows the merging of datasets and the discovery of triples of practical interest.

In general, inference schemes are inherently divided into two main approaches, backward-chaining and forward-chaining. Backward-chaining performs inference at query time, when the set of inferred triples is limited to the triple patterns defined in the query (rewriting/extending queries and posing them against the original data). Forward-chaining exhaustively generates all the triples from asserted ones, i.e., makes explicit the implicit data, through the application of certain rules. This process is called materialization, since inferred triples are explicitly written into the triple store. The query-specific backward-chaining techniques adversely affect query response times due to live inference. It is well suited to frequently changing data since forward-chaining requires full materialization after deletion. The materialization approach offers two particular benefits: off-line or pre-runtime execution of inference and consumer-independent data access, i.e., inferred data can be consumed as explicit data without integrating the inference engine with the runtime query engine. Herein, we focus on forward-chaining inference (materialization), but acknowledge the trade-off between both techniques. In the rest of the paper, unless otherwise stated, inference means forward-chaining inference.

Most of the research on forward-chaining has focused on decidability and fragment selection. The theory of processing per se has generally been relegated to secondary status. Regardless of the algorithm used for inference, execution involves numerous memory accesses. Excessive memory accesses in the presence of limited memory bandwidth results in CPU stalling – as the CPU waits for the data required while it is being fetched from the main memory. In contrast, a predictive memory access pattern guides the prefetcher to retrieve the data correctly in advance, thus enabling a better usage of memory bandwidth.

1.1 Contribution and outline

In this paper, we revisit the problem of in-memory inference, with a focus on sequential memory access and sorting algorithms. Our system Inferray, a portmanteau of inference and array, uses widely dynamic arrays of 64-bit integers. We demonstrate that the performance of sequential memory access can be exploited to design and implement an efficient forward-chaining inference system. RDFS-Plus.

The scientific contributions of the paper are three-fold:

- We address the transitivity closure issue that undermines performance of iterative rule-based inference systems. We validate that it is worth paying the penalty of translating data into Nuutila's algorithm data layout for a massive speedup and improved scalability.
- We show that using a fixed-length encoding in a vertical partitioning approach with sorted tables on both $\langle s, o\rangle$ and $\langle o, s\rangle$ enables very efficient inference and duplicate removal. As a consequence, most of the system performance relies on an efficient sort of the property tables made up of key-value pairs.
- Using a dense numbering scheme, we lower the entropy of the values outputted by the dictionary encoding. We leverage this low entropy by designing two fast sorting algorithms that outperforms state-of-the-art generic sorting algorithms on our use case.

We compare Inferray to alternative RDFS, ρdf, and RDFS-Plus inference systems and show that Inferray outperforms its competitors in scalability for transitive closure and performance on the most demanding ruleset. Inferray is available as a stand-alone reasoner, and is also compliant with Jena's interface. Inferray source code and experimental data are distributed under the Apache 2 License.

2. RELATED WORK & ANALYSIS

Traditional reasoning research literature has mostly focused on issues relating to expressivity, tractability, soundness, and completeness, particularly in closed domains with evaluation demonstrated over ontologies. However, inference over practical triple stores has an orthogonal set of requirements: scalability and execution time. This leads to two different sets of triple stores (including open source and commercial systems) where one offers an inference layer on top of the query layer (under the term Storage And Inference Layer (SAIL)), while the other primarily focuses on simple SPARQL querying support. Sesame [7], Jena TDB [16], RDFox [17] and OWLIM [4] belong to the first set, inherently supporting in-memory and native inference, while Hexastore [33], TripleBit [34] and RDF-3X [18] belongs to the second category of triple stores. The literature also points to some experimental techniques on large scale distributed reasoning [27] with no SPARQL support.

In terms of lightweight subsets of RDFS and OWL, approaches for forward-chaining are classified into two categories based on their underlying algorithmic approaches (Description Logics are outside the scope of this paper): the RETE Algorithm [10] and iterative rules systems. The RETE algorithm works on pattern matching principles – where rule sets are represented as a DAG and are matched to facts that are represented as relational data tuples. The current version of the algorithm is not public due to intellectual property constraints. However, several systems have implemented their own flavor of RETE [16]. Furthermore, RETE has recently been ported to GPU architectures [21].

Iterative rule systems offer a simpler design. Rules are iteratively applied to the data until a stopping criterion is matched. These systems usually operate on a fixed-point iteration principle – i.e., the inference process terminates when an iteration (application of all rules) derives no triples. Due to its simplicity, iterative rules application is widely used in distributed inference systems [27, 32], on highly parallel architectures such as CRAY XMT [11] and in Sesame and RDFox. Similarly, the OWLIM reasoners family uses a custom rule entailment process with a fixed-point inference mechanism.

2.1 Duplicate elimination

Duplicate elimination introduces a significant performance bottleneck in inference engines. Duplicates are triples that are generated by two different valid resolution paths, i.e., they are either already present in the system (as an output of other rules, or as an output of the same rule but with different input triples) or they are derived multiple times due to concurrency issues in distributed/parallel settings. Thus far, in an iterative system, it is important to cleanup duplicates immediately – this prevents a combinatorial explosion that would make the process intractable.

Cleaning up duplicates has been thoroughly discussed in the literature, especially when data are partially replicated during inference: on peers [27, 32] and on memory chunks using shuffle addressing [11]. Weaver and Haendler [32] acknowledge the issue, yet they offer no hint for duplicates elimination. In WebPIE [27], much of the design of the algorithm is focused on the duplicate elimination strategy. Even after introducing a series of optimizations to avoid duplicate generation, WebPIE spends a significant time cleaning up duplicates. WebPIE allows the measurement of time spent on duplicate elimination, since this is a synchronization barrier in the system. On the Lehigh University Benchmark (LUBM) [12], when applying OWL/1 rules, the system spends 15.7 minutes out of 26 on cleaning duplicates. Sesame checks the existence of each inferred triple to avoid duplicates, however it is not possible to measure the time spent eliminating duplicates, nor is this possible for Jena or OWLIM. In [11], Goodman et al. take advantage of the Cray XMT hardware architecture to implement large-scale reasoning. The Cray XMT is designed for massively parallel computing with non-uniform memory accesses, with primary application on large-scale graph processing. The memory architecture of the Cray XMT allows for checking the existence of a triple in a global hashtable before its insertion, without suffering the performance penalty that would occur on an x86 architecture.

2.2 Memory Access Patterns

The two primary data structures used by forward-chaining inference systems include graphs and dynamic collections of triples. Jena encompasses triples and rules as a graph, as an input for the RETE algorithm. Sesame also uses a graph model for its iterative rule-based algorithm. The main structure of its RDF graph is a linked list of statements. Thus, it offers good indexing to iterate over a list of triples. However, in this setting, the lack of multiple indices seriously hinders performance. These two previous reasoners are too slow and are usually not included in benchmarks. RDFox, a much faster reasoner, uses a tailor-made data structure that supports parallel hash-joins in a mostly lock-free manner. This structure enables scalable parallelism at the cost of random memory accesses. As for WebPIE, its in-memory model is a collection of objects representing triples. The disk/memory management is solely left to Hadoop and no optimization is provided. Finally, nothing can be commented on OWLIM design, as no details have been disclosed by its authors.

Accessing data from a graph structure requires random memory accesses. Many random memory accesses incur a high rate of cache misses. On sufficiently large datasets it also causes high rates of page faults and *Translation Lookaside Buffer* (TLB) load misses, thus, limiting the available memory bandwidth and forcing main memory accesses.

In the aforementioned work by Goodman, the random memory accesses issue no longer holds due to the architecture of the Cray XMT. Performance reported by the authors, using 512 processors and 4TB of main memory, indicates a rate of 13.2 million triples/sec. on a subset of RDFS, 10,000 times faster than WebPIE. However, the price tag of such a supercomputer, i.e., several million dollars, does not fit every wallet. On a more reasonable budget, recent experiments of the RETE algorithm on high-end GPUs [21] report a throughput of 2.7 million triples/sec. for ρdf and 1.4 million triples/sec. for RDFS. However, dedicated hardware is still required compared to the usual commodity servers.

Herein, we focus on mainstream architectures and our reasoner efficiently runs on any commodity machine. To overcome the memory bandwidth bottleneck, we focus our efforts on designing a data structure and algorithms that favor sequential access. As previously stated, there are two ways of designing an inference algorithm, namely, the RETE algorithm or the fixed-point inference that iteratively applies rules. The RETE algorithm requires a graph data structure that performs massive random memory accesses. Due to the very nature of the RETE algorithm, we believe it would be difficult to design a cache-friendly data structure that replaces the traditional graph. Fundamentally, one cannot foresee which triples will be matched by the pattern-matching stage of the algorithm. Hence, we do not pursue this approach and direct our research towards fixed-point inference. We demonstrate a rule-based inference system designed around sequential memory access that exhibits previously unmatched performance.

3. INFERRAY: OVERVIEW

We design Inferray with the objective of enhancing inference performance through an approach that both minimizes duplicate generation and leverages memory bandwidth using sequential memory accesses.

The global inference process is described in Algorithm 1. Inferray performs two steps to complete the inference process. In a first step, transitive closures are computed using a different data layout than for the second step (Line 2). The transitive closure cannot be performed efficiently using iterative rules application since duplicate generation rapidly degrades performance. To tackle this issue, we use the very efficient algorithm from Nuutila. This closure is applied on schema data for RDFS (`subClass` and `subProperty`), and is also applied to every transitive property as well as for `sameAs` for RDFS-Plus. Details on transitive closure are given in Section 4.1.

In a second step, Inferray applies rules iteratively, until a fixed-point is reached (condition in Line 4). To derive new triples using rules, Inferray takes two inputs: existing triples and newly-inferred triples, except for the very first iteration, where existing and newly-inferred storages are identical (see Line 3). For the rule-based inference stage, we observe that triple inference is identical to the process of performing joins and inserts. Due to the selectivity of the rules for all the fragments supported by Inferray, the vertical partitioning approach is well suited to perform inference efficiently using sort-merge joins. This structure along with the inference mechanism are described in Section 4.2.

Inferred triples are stored in the structure `inferred`. They may contain triples that are already in the `main` structure: the duplicates. Duplicates are consequently removed from

`inferred`, then these new triples are added in `main` and `new` to continue the process. Section 4.3 presents this process.

The rule sets supported by Inferray are of various complexities. RDFS and ρdf require only two-way joins that are efficiently performed. However, several rules of RDFS-Plus contain multi-way joins that may hinder performance. We detail in Section 4.4 how we efficiently handle these rules.

Algorithm 1: Inferray: main loop

```
Input: Set of triples triples, triples stores:
       main, inferred, new
Output: Set of triples with inferred triples
1 main ← triples;
2 transitivityClosures();
3 new ← main;
4 while new ≠ ∅ do
5 |   inferred ← infer(main, new);
6 |   new ← inferred \ main;
7 |   main ← main ∪ new;
8 end
```

To perform sort-merge joins efficiently, property tables that contain pairs of $\langle subject, object \rangle$ ($\langle s, o \rangle$ in the rest of the paper) must be sorted on $\langle s, o \rangle$ and possibly on $\langle o, s \rangle$. Inferray must also maintain sorted and duplicate-free property tables after each iteration of rules. The design choices and sorting algorithms are presented in Section 5.

4. INFERENCE MECHANISMS

4.1 Transitivity Closure

For the case of transitivity closure computation, performance is hindered by the large number of duplicates generated at each iteration [27]. By definition, subproperties do not inherit the transitivity property, i.e., we cannot infer that a subproperty of a transitive property is transitive. This implies that any statement of the form *p is transitive* must hold prior to the inference process. This allows us to handle transitivity closure before processing the fixed-point rule-based inference.

In practice, transitivity closure computation differs for RDFS and RDFS-Plus. For RDFS, the closure is computed for both `subClassOf` and `subPropertyOf`. For RDFS-Plus the closure is extended to every property declared as transitive. RDFS-Plus also includes `owl:sameAS` symmetric property on individuals. To compute the transitivity closure on the symmetric property, we first add, for each triple, its symmetric value and then we apply the standard closure.

Computing the transitivity closure in a graph is a well-studied problem in graph theory. It still remains a difficult problem for large scale data [8]. Fortunately, for the case of RDF data, transitivity closure is always applied to the schema data, the `Tbox`, which is always of a very reasonable size compared to the set of instances, the `Abox`. Instances (`Abox`) are guaranteed to match the closure of their respective classes through the `CAX-SCO` rule.

The schema graph does not present any guarantee regarding the presence or absence of cycles. The fastest known algorithm to compute transitivity closure in a digraph, cyclic or acyclic, is Nuutila's algorithm [20]. The algorithm first performs a strong component detection. A quotient graph is then computed, each component c is linked to another c' if there is an edge from some node in c to some node in c'. The quotient graph is then topologically sorted in reverse order. The reachable set of each node in the quotient graph is computed as the union of the reachable sets in its outgoing edges. This relation used to compute the quotient graph ensures that the transitivity closure of the original graph can be deduced from the transitivity closure of the quotient graph. The closed original graph is achieved by mapping back strong components from the quotient graph to their original nodes. All steps, other than unioning reachable sets, are linear time operations.

Cotton's implementation[1] of Nuutila's algorithm is oriented towards high-performance using compact data layout. The strong component detection is implemented with bit vectors to ensure compactness and a better cache behavior. The computationally intensive step of unioning reachable sets is optimized by storing reachable sets as sets of intervals. This structure is compact and is likely to be smaller than the expected quadratic size. Branchless implementation of merging ensures high performance.

This method performs well when node numbers are dense. This interval representation provides a compact representation and exhibits a low memory footprint. Therefore, for our implementation, sparsity of the graph is reduced by first splitting it into connected components and later renumbering the nodes to ensure high density. For this purpose, we use the standard `UNION-FIND` algorithm, then translate the nodes' ID to keep a dense numbering, finally applying Nuutila's algorithm. The closure of each connected component is thus far added to the list of $\langle s, o \rangle$ in the main triple store.

4.2 Sort-Merge-Join Inference

Each inference rule consists of a `head` element and a `body` element. The `body` indicates the condition(s) that must be validated to assert new triples. For instance the rule `CAX-SCO` (Table 5, #3) that is shared between the three aforementioned rulesets, states that if class c_1 is subclass of c_2, then every resource of type c_1 is also of type c_2.

In practice, most of the patterns are selective on the property value. Thus, we use vertical partitioning [1] to store the triples in the memory in order to harness the advantages of selectivity. The principle of vertical partitioning is to store a list of triples $\langle s, p, o \rangle$ into n two-column tables where n is the number of unique properties. Field testing has shown great performance for query answering as well as scalability. Experiments from Abadi et al. were also reproduced and validated by an independent study [25].

In Inferray, the triple store layout is based on vertical partitioning with column tables. Triples are encoded using a dictionary. This is common practice among triple stores (Triple Matrix, RDF3-X) to reduce the space occupied by triples and to enhance performance by performing arithmetic comparisons instead of strings comparisons. Each part of a triple, i.e., subject, properties, or object, is encoded to a fixed-length numerical value of 64 bits. Due to the large volume of data (the LOD cloud is already over several hundred billions triples) 32 bits would not be sufficient. From a database point of view, applying rules is similar to selecting triples matching a pattern and inserting new ones according to a given pattern[2]. The alternative consists in using a hash-join strategy, like RDFox. However, while applying

[1] https://code.google.com/p/stixar-graphlib/

[2] See `http://topbraid.org/spin/rdfsplus.html` for a SPARQL version of RDFS-Plus

rules, sorted property tables are reused in many join operations, thus mutualizing sorting time. As shown in [3], sorting represents more than half the time required in a sort-merge join. Moreover, sorted property tables are later reused to optimize both duplicate elimination and merging (Lines 6 and 7, Algorithm 1). Due to the mutualization of the sorting time, sort-merge join becomes more efficient than hash-join.

Traditional vertical partitioning approaches enforce sorting property tables on $\langle s, o\rangle$ alone. However, in order to perform sort-merge joins, some rules require property tables to be also sorted on $\langle o, s\rangle$. Consequently, property tables are stored in dynamic arrays sorted on $\langle s, o\rangle$, along with a cached version sorted on $\langle o, s\rangle$. The cached $\langle o, s\rangle$ sorted index is computed lazily upon need. This cache may be cleared at runtime if memory is exhausted.

Figure 1 shows a concrete example of inference, where the previously described rule CAX-SCO is applied to sample data. Triples for the property `subClassOf` are given in the first vertical list, sorted on $\langle s, o\rangle$, starting from the left. The triples contained in this column are the explicit ones in our running example (`<human, rdfs:subClassOf, mammal>` and `<mammal, rdfs:subClassOf, animal>`), and are encoded with the respective values $[2, 1, 3]$ and $[2, 1, 4]$. The triples read as follows: property value (1) then from top to bottom, subject and object – $[2, 3]$ for the first triple and $[2, 4]$ for the second. These triples are listed in the column of index 1, sorted on $\langle s, o\rangle$, according to the provided dictionary encoding (top right of Figure 1). The vertical partitioning scheme is preserved: properties are sorted, and for each, the set of $(subject, object)$ pairs are subject-sorted. We also introduce two instances for the sake of illustration – `:Bart` and `:Lisa` of `type human`. We illustrate how the system infers that `:Bart` and `:Lisa` are both of `type mammal` and `animal`.

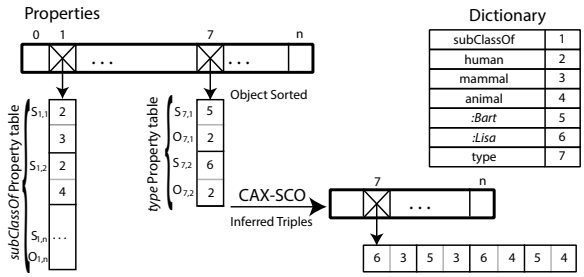

Figure 1: Sort-Merge-Join Inference: `CAX-SCO`.

This example presents the principle of Sort-Merge-Join Inference on a single triple store. However, in the practice, applying rules to a single triplestore leads to the derivation of numerous duplicates at each iteration. To avoid duplicates, new triples from the previous iteration are used in combination with the main triple store.

While the described procedure is specific to the rules similar to `CAX-SCO`, other rules do, however, follow a similar or simpler principle to infer new triples. We precisely describe the different classes of rules in Section 4.4 and analyze their respective complexities.

Our system applies all the rules in bulk to triples to result in a set of property tables containing inferred triples and possibly some duplicates. To perform the next iteration, i.e. the application of rules with the newly inferred triples, we first sort on $\langle s, o\rangle$ and remove duplicates from the inferred

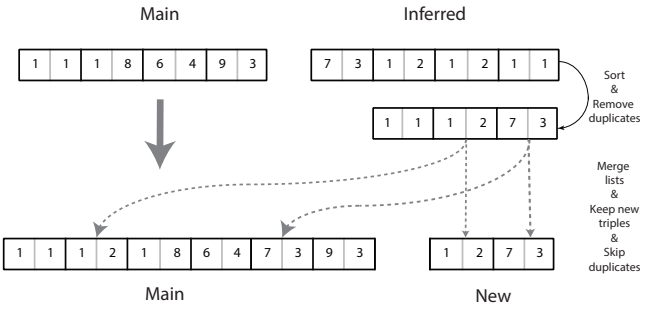

Figure 2: Updating property tables after an iteration. For the next iteration, *New* will serve as input for rule application

triples. Second, we add inferred triples that were not already present in the main triple store (Line 7, Algorithm 1) and finally, we sort the property tables to which new triples have been added. In the case of receiving new triples in a property table, the possibly existing $\langle o, s\rangle$ sorted cache is invalidated.

4.3 Merging Data

Fig. 2 shows the update process that takes place at each iteration, after all rules have been fired. This process takes place for every property when new triples have been inferred.

In the first step, inferred triples are sorted and duplicates are removed. In the second step, `main` and `inferred` triples are merged into the updated version of `main`; inferred triples that do not exist in `main` are copied in `new`. Duplicate removal takes place in these two steps: during the first step by removing the inferred duplicates; and during the second one by skipping triples in `inferred` that are already in `main`. The time complexity of the whole process is linear as both lists are sorted. On the one hand, each rule is executed on a dedicated thread (Line 5 of Algorithm 1) and holds its own inferred property table to avoid potential contention. On the other hand, data merging (Lines 6 and 7) deals with independent property tables, this process is also easily parallelized.

4.4 Rule Classes

We pigeonholed the rules into classes, according to their selection and insertion patterns. Under the hood, these classes are effectively used in Inferray to implement rules with similar patterns – the list of rules with their classes is given in Table 5. RDFS and ρdf contain rules with a maximum of two triple patterns in the *body* part, i.e. the select query. Rules with a single triple pattern in the body are trivial and are not detailed here.

For two-way joins on triple patterns $\langle s_1, p_1, o_1\rangle$ we introduced the following classes.

Alpha. Join for α-rules is performed on either subject or object for the two triples. Since Inferray stores both $\langle s, o\rangle$ and $\langle o, s\rangle$ sorted property tables, this sort-merge join is efficiently performed, as described for `CAX-SCO` in Fig. 1.

Beta. β-rules perform a self-join on the property table associated with the fixed property in both triple patterns. Join is performed on the subject for one pattern and on the object for the other. As Inferray stores both $\langle s, o\rangle$ and $\langle o, s\rangle$-sorted tables, the join is a standard sort-merge join as for

the α-rules, with the potential overhead of computing the $\langle o, s\rangle$-sorted table if not already present.

Gamma. For the γ-rules, one triple pattern has a fixed property and two variables $\langle s, o\rangle$, also being properties. The join is performed on the property of the second triple pattern. Consequently, this requires to iterate over several property tables for potential matches. Fortunately, in practice, the number of properties is much smaller compared to classes and instances.

Delta. For the δ-rules, the property table of the second antecedent is copied (may be reversed) into the resulting property table.

Same-as. The four same-as rules generate a significant number of triples. Choosing the base table for joining is obvious – since the second triple patterns select the entire database. Inferray handles the four rules with a single loop, iterating over the same-as property table. `EQ-REP-P` is the simplest case; the property table associated with the subject is copied and appended to the property table associated to the object. `EQ-REP-O` and `EQ-REP-S` follow the same plan as for γ-rules. `EQ-SYM` falls in the trivial case of the single triple pattern rule.

Theta. This class embeds all transitive rules. The transitivity closure for these rules is computed before the fixed-point iteration using the process described in Section 4.1.

RDFS-Plus has three rules with three triple patterns in the antecedent (`PRP-TRP`, `PRP-FP` and `PRP-IFP`). `PRP-TRP` is the generalization of the transitivity closure from `SCM-SCO` and `SCM-SPO` to all transitive properties. `PRP-TRP` is handled in a similar fashion, viz., by computing the closure as described in Section 4.1. `PRP-FP` and `PRP-IFP` are identical (except for the first property), the system iterates on all functional and inverse-functional properties, and performs self-joins on each property table. For `PRP-FP`, sorted property tables on $\langle s, o\rangle$ and $\langle o, s\rangle$ allow linear-time self-joins. The total complexity is $O(k \cdot n)$ where k is the number of functional and inverse functional properties and n is the largest number of items in a property table.

5. SORTING KEY/VALUE PAIRS

The cornerstone of Inferray is the sorting step that takes place at two different times in the process. First, when inferred triples are merged into `main` and `new`. Second, when a property table is required to be sorted on $\langle o, s\rangle$ for inference purpose. These two operations differ slightly, as the first one requires potential duplicates to be removed, whereas property tables are duplicate-free for the second operation.

To sort these lists of pairs of integers efficiently, we use a dense numbering technique to reduce the entropy. The low entropy allows us to design efficient algorithms that outperform state-of-the-art generic solutions on our use case.

5.1 Dense Numbering

A key observation is that inference does not produce new subjects, properties or objects – only new combinations of existing s, p, o are created in the form of triples. From a graph point of view, inferencing corresponds to increasing the density of the graph.

In property tables, except for `Same-as`, subjects are of the same nature (property or not) as are objects (either property or non-property). For instance, the property table for `rdfs:domain` contains properties as subjects and non-properties as objects. Subject/Object pairs are sorted lexicographically. We aim at keeping numbers dense for properties on one side, and for non-properties on the other.

Numbering of properties must start at zero for the array of property tables (Figure 1). The number of properties is negligible compared to the number of non-properties. Dense numbering is performed at loading time. Each triple is read from the file system, dictionary encoding and dense numbering happen simultaneously. However, we do not know in advance the number of properties and resources in the dataset, hence it is not possible to perform dense numbering starting at zero for properties. To circumvent this issue and to avoid a full data scan, we split our numbering space, i.e., $[0; 2^{64}]$, at 2^{32} – considering that in practice the number of properties is smaller than the number of resources. We assign the properties numbers in decreasing order and resources in an increasing order. For instance, the first property will be assigned 2^{32}, the second $2^{32} - 1$, and the first resource will be assigned $2^{32} + 1$, thereby, keeping dense numbering for both cases. To access the array of property tables, we perform a simple index translation.

5.2 Counting sort

Counting sort is an integer sorting algorithm that shows excellent performance for small values ranges, in other words, when the entropy is low.

Sorting pairs with counting sort is not a well-studied problem, therefore we devise an adaptation of the algorithm to fulfill our requirement (Algorithm 2). The underlying idea of our counting sort for pairs is to keep the histogram principle for sorting the subjects (assuming we sort on $\langle s, o\rangle$) – while being able to sort in linear time the objects associated with each subject value (i.e., keys of the histogram).

For this purpose, we store the objects in a single array. By using their positions in the unsorted array, combined with histogram values, we are able to sort subarrays corresponding to subjects. Finally, we build the sorted array by combining the histogram and the object sorted arrays.

The algorithm is given in Algorithm 2. For the sake of explanation, a trace of Algorithm 2 on a sample array is presented in Figure 3. The algorithm starts by computing the traditional histogram on the subjects' values and keeps a copy of it (Lines 1 and 2). The starting position of each subject in the final array is computed by scanning the array and summing the consecutive values (Line 3).

The second step (Lines 4 to 10) builds an array in which object values are placed according to the future position of their associated subjects in the final sorted array. These object values are unsorted. In Figure 3, the subject with value 4 appears twice (first and last pairs) with object values 1 and 4. After this step, the object value 4 is in position 2 in the *objects* array, before object value 1. Subarrays of *objects* are associated with each histogram value. For the subject s_i, the subarray starts at the position given by $startingPositions[s_i]$ and its length is equal to the number of subjects in the array, i.e. $histogram[s_i]$. Objects are added in the subarray starting from the end. The relative position in the subarray is given by $histogram[s_i]$. Once an object for s_i is added, this value is decreased. Hence, the absolute position of the object value in the *objects* array is given by the sum of $startingPosition[s_i]$ and $histogram[s_i]$.

Algorithm 2: Counting sort algorithm for pair of integers in a single array, also removing duplicate pairs.

Data: an array of integer *pairs*, representing a serie of pairs $p_i = (s_i, o_i)$. *width* is the range of the array. Subjects s are on even indices, Objects o are on odd indices.
Result: A sorted array of unique pairs of key-value, sorted by key then value

```
1  histogram = histogram of subjects;
2  histogramCopy = copyArray(histogram);
   // Compute starting position of objects
3  int[] start = cumulative(histogram);
   // Put objects in unsorted subarrays
4  int[] objects = int[pairs.length/2]
5  for i = 0; i < pairs.length; i += 2 do
6  |  int position = start[pairs[i] − min];
7  |  int remaining = histogram[pairs[i]-min];
8  |  histogram[pairs[i] − min]−−;
9  |  objects[position + remaining − 1] = pairs[i + 1];
10 end
   // Sort objects for each subject
11 for i = 0; i < start.length; i++ do
12 |  sortFromTo(objects, start[i], start[i + 1]);
13 end
   // Build the resulting array
14 int j = 0, l = 0, previousObject = 0;
15 for i = 0; i < histogramCopy.length; i++ do
16 |  int val = histogramCopy[i];
17 |  int subject = min + i
18 |  for k = 0; k < val; k++ do
19 |  |  int object = objects[l++]
   |  |  // Avoid duplicates
20 |  |  if k == 0 || object != previousObject then
21 |  |  |  pairs[j++] = subject;
22 |  |  |  pairs[j++] = object;
23 |  |  end
24 |  |  previousObject = object;
25 |  end
26 end
27 trimArrayToSize(pairs, j);
```

In the third step (Lines 11 to 13), the subarrays are sorted using the traditional counting sort algorithm. Start and end indices are given by scanning *startingPosition*.

The resulting array is reconstructed by scanning the copy of the histogram (the original contains only 0 after the second step) (Lines 14 to 26). Removal of duplicates and possible trim are also implemented at this step.

5.3 MSDA Radix

Radix sort is one of the oldest sorting algorithms. A radix sort successively examines D-bit digits (the radix) of the K-bit keys. There are two strategies to implement radix sort depending on the order in which digits are processed: Least Significant Digit (LSD) or Most Significant Digit (MSD). MSD groups blocks by digits and recursively processes these blocks on the next digits. While LSD needs to examine all the data, MSD is, in fact, sublinear in most practical cases. To sort the property tables, we sort pairs of key-value of 64 bits each. For this case, radix remains a better choice than merge sort [23]. A standard radix sort will group blocks on the first digit examined, and recursively process blocks until keys are sorted. In the case of equality between keys, radix sort is called to sort on the values.

In the worst case, 128 bits are examined. However, in our case, we can take advantage of the dense numbering – the whole range of 64 bits is unlikely to be completely

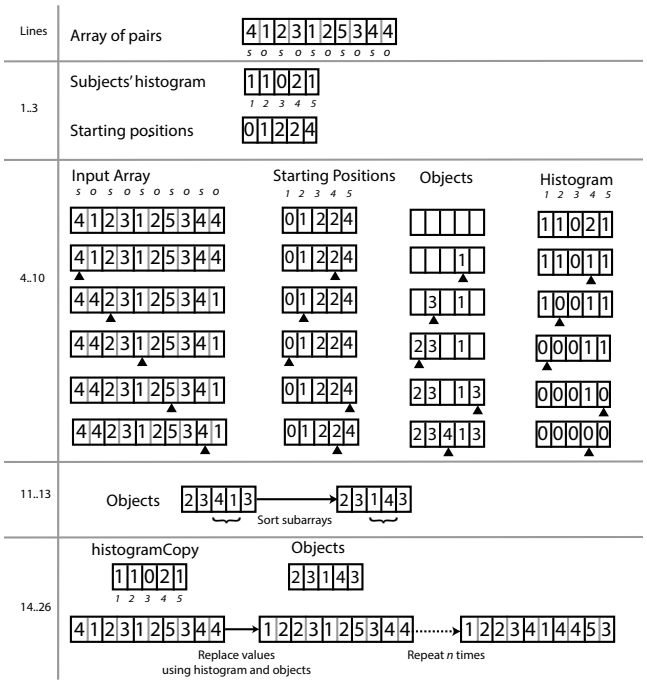

Figure 3: Trace of Algorithm 2 on a sample array. Steps are identified by the line numbers of the algorithm.

Range (entropy)	Algorithm	Size 500K	1M	5M	10M	25M	50M
500K	Counting	27.7	**35.7**	**64.1**	**70.4**	**42.0**	**56.0**
(18.9)	MSDA Radix	**31.2**	31.2	33.3	23.4	20.3	22.4
1M	Counting	20.8	27.7	**49.0**	**64.9**	**66.8**	**42.1**
(19.9)	MSDA Radix	**27.7**	**29.4**	33.7	32.8	19.1	18.4
5M	Counting	6.2	10.8	25.2	**33.1**	**48.0**	**63.2**
22.6	MSDA Radix	20.8	22.7	**29.4**	30.6	33.4	33.6
10M	Counting	3.3	6.0	18.5	24.6	**36.3**	**47.7**
(23.26)	MSDA Radix	**47.6**	20.8	**27.7**	**29.4**	30.4	33.1
25M	Counting	1.4	2.7	9.9	16.1	23.5	**31.8**
(24.58)	MSDA Radix	**41.6**	**33.3**	**19.5**	21.9	**24.0**	24.5
50M	Counting	0.7	1.3	5.7	10.1	17.5	**23.6**
(25.58)	MSDA Radix	**35.7**	**35.7**	17.6	19.3	**22.8**	23.2
Generic	*Radix*$_{128}$	21	20.4	21.8	22.4	22.7	22.7
	Merge$_{128}$	27.5	27	25	21.8	20	18.75
	Mergesort	22.7	20.8	18.0	17.8	16.2	15.6
	Quicksort	8.6	7.9	5.9	5.6	5.2	5.1

Table 1: Performance in millions of pairs second for counting, MSD radix adaptive for ranges and sizes from 500K to 50M. Standard algorithms are given without entropy. Operating ranges are underlined. Results in italic are adapted from [23].

in use for every property table and all numbers are in a window around 2^{32}. Therefore, the significant bit for radix sorting may start in the i-th position, where $i \geq D$. It is consequently useless to sort the leading $i - 1$ bits. To determine i, Inferray computes the number of leading zeros of the range divided by the size of the radix. For a range of 10 million with an 8-bit radix, significant values start at

the sixth byte out of eight. For small or dense datasets this optimized MSD radix sort saves multiple recursive calls. We denote this version as MSDA (A for adaptive) radix in what follows.

5.4 Operating Ranges

Intuitively, counting and radix sort algorithms are expected to reach their respective peaks under different conditions. The range of the data is the primary bottleneck for counting sort. For a very large range where data is sparsely distributed, the algorithm will perform poorly. Meanwhile, radix will efficiently cluster data by blocks and the sublinear effect will be higher. On the contrary, we expect counting sort to outperform radix for small data ranges.

We conduct experiments to fix the boundaries of optimal operating ranges for both algorithms. We considered ranges (r) and sizes (n) that are common for Linked Open Data datasets, from 500K to 50M. We compare our counting sort and MSDA radix with state-of-the-art algorithms from [23]. The algorithms noted $Radix_{128}$ and $Merge_{128}$ are SIMD optimized implementations of Radix and Mergesort, running on CPU. Since Inferray sorts pairs of 64-bit integers, we compare our approach against 128-bit integer sorting algorithms.

The results of our experiments are shown in Table 1. Our algorithms are at least on par with state-of-the art algorithms for 83.3% of the considered cases, and offer a three times improvement in the best case. In the worst case (5M elements, 25.58 bits entropy), our approach reaches 70% of state-of-the-art throughput.

The main reason behind this performance is that by design, both our algorithms (Counting and MSDA) takes advantage of the low entropy induced by the dense numbering. At the opposite, generic sorting algorithms should not be impacted by entropy variation. As reported in [23], both $Radix_{128}$ and $Merge_{128}$ are barely impacted by entropy variation. A second reason, is that these algorithms are highly performant on smaller keys (i.e. 32 bits). Current width of the SIMD registers prevent them to perform efficiently on 128-bit integers.

For small ranges (500K and 1M), counting sort outperforms largely radix algorithms when the size of the collection outgrows the range of the data. For $r = 1M$ and $n = 25M$, counting offers a 3.5 factor of improvement. These values mean a very dense graph, in which radix algorithms must perform a high number of recursions. For very sparse graphs, i.e., large r and small n, counting sort suffers a performance penalty from scanning large sparse arrays. Radix sort at the opposite end offers better performance when data is sparse. Grouping becomes efficient from the first examined digits and thus strongly limits recursive calls. As a rule of thumb, counting outperforms MSD radix when the size of the collection is greater than its range. When the range is greater than the number of elements, the adaptive MSD radix consistently outperforms the standard implementation. The values used by Inferray for sorting property tables are the ones underlined in Table 1.

6. BENCHMARK

In this section, we evaluate our Java implementation of Inferray against state-of-the-art systems supporting in-memory inference. Our testbed is an Intel Xeon E3 1246v3 processor with 8MB of L3 cache. The system is equipped with 32GB of main memory; a 256Go PCI Express SSD. The system runs a 64-bit Linux 3.13.0 kernel with Oracle's JDK 7u67.

Competitors. We run the benchmark on major solutions from both the commercial and open-source worlds. As a commercial world representative, we use OWLIM-SE, the lite version being limited in its memory usage [15]. OWLIM is widely used in both academic benchmarks and commercial solutions. OWLIM is also well known for powering the knowledge base of the BBC's FIFA WorldCup 2010 website [14], and for its performance [15]. As open-source solutions we use RDFox [17] and WebPIE [27]. RDFox allows fast in-memory, parallel, datalog reasoning. Unlike our sort-merge-join inference approach, RDFox proposes a mostly lock-free data structure that allows parallel hash-based joins. WebPIE, unlike its backward-chaining counterpart QueryPIE [28], is a forward-chaining reasoner, based on Hadoop. All competitors but WebPIE – which cannot by design, are configured to use in-memory models.

Data. As there is no well-established benchmark for RDFS and RDFS-Plus inference, we built our own benchmark with widely used real-world ontologies as well as generated ontologies usually found in SPARQL benchmarks. As stated in [9], it is necessary to benchmark on both generated and real-world data to obtain significant results. Regarding real-world ontologies, we use the taxonomy of Yago [26] that contains a large set of properties. This dataset challenges the vertical partitioning approach, due to the large number of generated tables. Transitive closure is challenged by the large number of `subClassOf` and `subPropertyOf` statements. We use the Wikipedia Ontology [22], automatically derived from Wikipedia categories, which contains a large set of classes and a large schema. As generated ontologies, we use 8 LUBM [12] datasets ranging from 1 million to 100 million triples. Only RDFS-Plus is expressive enough to derive many triples on LUBM. We use BSBM [5] to generate ontologies that support RDFS. We generated datasets from 1 million triples up to 50 million. We implemented a transitive closure dataset generator that generates chains of *subclassOf* for a given length. We use 10 of these datasets, with chains from 100 nodes to 25,000. All datasets are accessible online.

Measures. All experiments are conducted with a timeout of 15 minutes. Experiments are executed twice to warm up caches and are then repeated five times, average values over the five runs are reported. We report inference time for Inferray , RDFox and WepPIE. For OWLIM, we use the same methodology as [17] – we measure execution times for materialisation and for import, we then substract the two times. CPU and software counters for cache, TLB and page faults are measured with `perf` software. This tool comes with the `repeat` option for warming up the file system caches.

Rulesets. We ran the benchmarks using four rulesets, ρdf, RDFS, RDFS default and RDFS-Plus. OWLIM-SE and RDFox have built-in rulesets and offer to use custom ones. We use their built-ins for RDFS and we implemented custom rule sets for the remaining ones. WebPIE supports RDFS and direct type hierarchy inferencing, but does not offer support for custom rule sets.

6.1 Transitivity Closure

For the first experiment, we compute the transitivity closure for chains of triples of various length. For an input chain of length n, exactly $\frac{n^2-n}{2}$ triples are inferred. For In-

Type	Dataset	Fragment	Reasoners			
			Inferray	OWLIM	RDFox	Webpie
Synthetic	BSBM_1M	ρdf	193	1,1157	**56**	N/A
		RDFS-default	254	1,750	**63**	N/A
		RDFS-Full	403	786	**85**	66,300
	BSBM_5M	ρdf	341	7,522	**282**	N/A
		RDFS-default	**394**	3,424	405	N/A
		RDFS-Full	863	5,503	**510**	89.,400
	BSBM_10M	ρdf	**580**	16,856	589	N/A
		RDFS-default	**702**	18,283	763	N/A
		RDFS-Full	2,026	19,930	**990**	110,700
	BSBM_25M	ρdf	**1,220**	22,279	1,298	N/A
		RDFS-default	2,756	22,425	**1,859**	N/A
		RDFS-Full	5,082	22,443	**2,355**	296,300
	BSBM_50M	ρdf	**3,049**	–	3,908	N/A
		RDFS-default	**2,865**	–	4,898	N/A
		RDFS-Full	6,445	–	**4,566**	–
Real-World	Wikipedia	ρdf	275	573	**120**	N/A
		RDFS-default	250	719	**161**	N/A
		RDFS-Full	652	791	**160**	63,700
	Yago	ρdf	3,124	21,925	**1,575**	N/A
		RDFS-default	3,200	22,639	**1,659**	N/A
		RDFS-Full	3,824	28,438	**2,251**	1.4e6
	Wordnet	ρdf	536	6,182	**313**	N/A
		RDFS-default	444	2,503	**288**	N/A
		RDFS-Full	1,398	7,248	**396**	67,340

Table 2: Experiments on RDFS flavors. Execution times is in miliseconds. Each value is the average over five runs.

Type	Dataset	Fragment	Reasoners		
			Inferray	OWLIM	RDFox
Synthetic	LUBM_1M	RDFS-Plus	**19**	1,324	69
	LUBM_5M	RDFS-Plus	**114**	3,907	322
	LUBM_10M	RDFS-Plus	**540**	6,175	855
	LUBM_25M	RDFS-Plus	**1,092**	12,493	1,920
	LUBM_50M	RDFS-Plus	**1,984**	31,187	4,077
	LUBM_75M	RDFS-Plus	**2,047**	48,233	6,939
	LUBM_100M	RDFS-Plus	**2,514**	72,098	10,613
Real-world	Wikipedia	RDFS-Plus	**310**	3,033	342
	Yago Taxonomy	RDFS-Plus	**3,085**	29,747	3,204
	Wordnet	RDFS-Plus	**232**	5,692	860

Table 3: Experiments on RDFS-Plus. Execution times is in miliseconds. Each value is the average over five runs.

	Length of `subclassOf` chain						
System	100	500	1000	2500	5000	10000	25000
Inferray	22	**53**	**165**	**612**	**1253**	**3275**	**14712**
OWLIM	45	1952	21200	361231	–	–	–
RDFox	**8**	303	3034	87358	597286	–	–

Table 4: Performance for the transitivity closure. Execution time in milliseconds

ferray, we measure regular runs, data are translated into the vertical partitioning structure, rules are started, even if no further triples are inferred. Results are reported in Table 4.

First of all, Inferray scales well beyond its competitors. While OWLIM stops for a chain of 2,500 nodes, i.e., to generate around three millions triples, RDFox manages to reach the generation of 50M triples for a chain of 5,000 nodes. Inferray scales up to the generation of 313M triples in 14.7 seconds, reaching a throughput of 21.3M triples per second. For chains over 25,000 nodes, Inferray stops due to lack of memory ($\geq 16GB$). Second, Inferray is much faster than other systems. For a chain of 2,500 nodes, Inferray is 142 times faster than RDFox and 590 times faster than OWLIM. Cotton's implementation exhibits very good memory behavior. While OWLIM struggles with high page faults and TLB misses rates, Inferray remains memory-friendly, as depicted in Figure 4. RDFox is faster than Inferray on the smallest dataset, thanks to a lower TLB misses rate. For largest datasets, RDFox suffers from growing cache misses and TLB misses rates. Meanwhile, its page faults rates demain mostly on par with Inferray.

The introduction of the temporary storage for transitivity closure along with the compact and efficient implementation of Nuutila's algorithm offers a major improvement in both performance and scalability.

6.2 RDFS Fragments

RDFS, RDFS-default and ρdf have very similar levels of expressiveness. RDFS in its full version derives many more triples than its default version (Section 3). However, this derivation does not entail a higher complexity since all *useless* triples can be inferred in linear-time in a post-processing step. For this latter fragment, RDFox consistently outperforms OWLIM and Inferray on both real-world and synthetic datasets. The hash-based structure of RDFox offers a better handling of triples generated by RDFS rules 8, 12, 13, 6 and 10 from Table 5. Inferray is largely impacted by the duplicate elimination step caused by these triples. Regarding scalability, both Inferray and RDFox handle materialization up to 50M, while OWLIM timeouts. On RDFS-default and ρdf, performance are more balanced between Inferray and RDFox. RDFox remains faster on small dataset (BSBM1M and real-world datasets), while Inferray performance improves with the size of the dataset. The memory behaviour, not reported due to lack of space, shows that the hash-based data structure from RDFox does not suffer much from random memory access. Its L1 cache miss rate is slightly over 1.2%, while Inferray exhibit a constant rate of 3.4 %. We believe this behaviour is due to the nature of the rules applied for RDFS. These rules are applied on small set of properties, and properties are never variable in the rules. Therefore, the linked-list mechanism from RDFox that connects triples with similar $\langle s, p\rangle$ and similar $\langle o, p\rangle$ offers an interesting solution that matches well with the nature of RDFS rules.

6.3 RDFS-Plus

Table 3 present results for RDFS-Plus. RDFS-Plus is the most demanding ruleset in our benchmark. It contains more rules than the previous ones, including three-way joins and rules with property as a variable. For this benchmark, OWLIM is much slower than Inferray and RDFox, by at least a factor 7. Inferray consistently outperforms RDFox, by a factor 2 except on the Wikipedia Ontology where both reasoners offer similar performance. Inferray scales linearly with the size of the dataset on the synthetic LUBM dataset. RDFox offers a linear scalability up to LUBM50M. Performance degrades slightly afterwards. The main difference with the RDFS benchmark, is the complexity of the rules. In this case, Inferray offers better performance than its com-

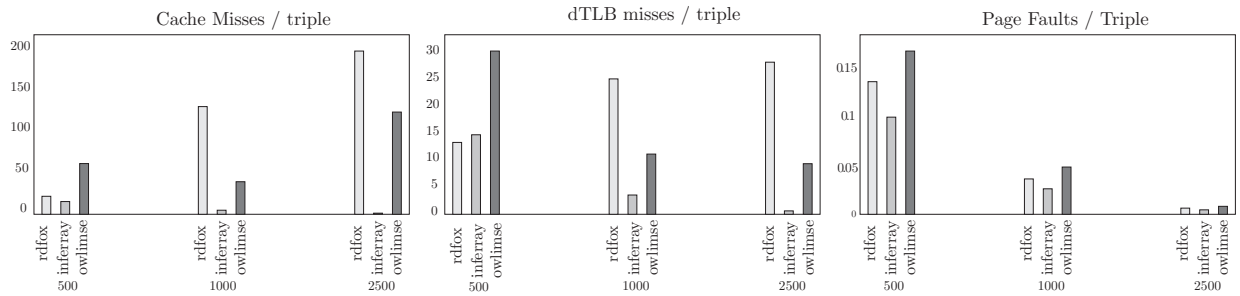

Figure 4: Cache misses, TLB misses and Page Faults per triple inferred for the transitivity closure benchmark.

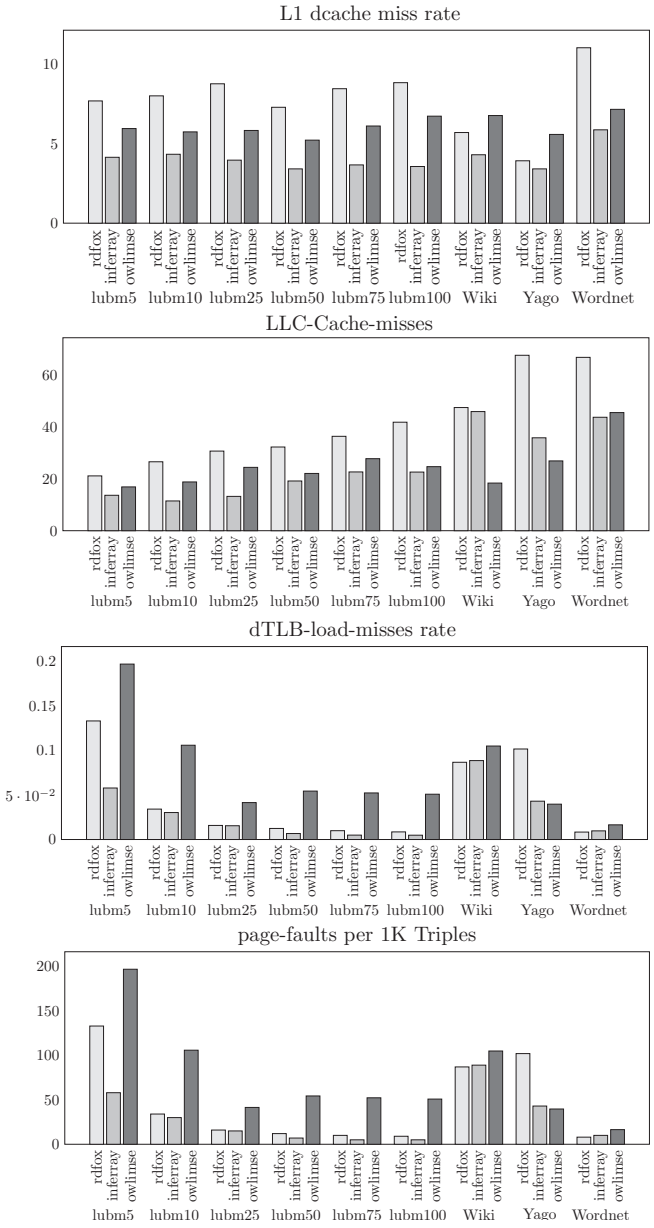

Figure 5: Cache misses (L1d and LLC), TLB misses and Page Faults per triple inferred for the RDFS-Full benchmark.

petitors when rules become more complex. RDFox fails to scale for more complex fragments due to its triple store data structure: the more complex the fragment, the more non-uniform memory accesses are made in its two-dimensional array through pointers. Wikipedia and Yago benchmarks present similar results for Inferray and RDFox because they do not rely much on `owl:` constructs. The difference is more noticeable on the rest of the datasets, where Inferray largely outperforms its competitors. Vertical partitioning is challenged by multi-way joins but it still offers superior performance compared to the hash-based structure of RDFox. In this case, the RETE approach from OWLIM seems to have difficulties handling a large ruleset while iterative approaches from RDFox and Inferray suffer less penalty.

6.4 Memory accesses

Figure 5 presents the rates of cache misses (L1 data and Last Level), TLB misses, and page faults per triple inferred for the RDFS-Plus benchmark. Regarding main memory access pattern (TLB and page faults), the iterative approaches from RDFox and Inferray largely outperform the RETE algorithm from OWLIM. We reported in Section 6.2 the following L1 cache miss rates for RDFox and Inferray: respectively 1.2 % and 3.4 %. Interestingly, Inferray cache behaviour does not vary with the ruleset, whereas RDFox cache performance degrades on RDFS-Plus, reaching a miss rate of 11 % on Wordnet. The size of the dataset, as expected by design, does not impact Inferray's cache behaviour.

7. CONCLUSION

In this paper, we present algorithms and data structures that drive Inferray, a new forward-chaining reasoner. The main technical challenges are to perform efficient transitive closure, to foster predictable memory access patterns for mechanical sympathy, and to efficiently eliminate duplicates. The first challenge is addressed by a transitive closure computation performed before the inference algorithm. The second challenge is tackled by a sort-merge-join inference algorithm over sorted arrays of fixed-length integers that favors sequential memory access. Duplicate elimination is handled by scanning arrays after efficient sorting, using efficient sorting algorithms for low entropy. Our exhaustive experiments demonstrate that leveraging solutions for these issues lead to an implemented system that outperforms existing forward-chaining reasoners on complex fragments. This paper comes with an open-source implementation of Inferray, which will be of the utmost practical interest to working ontologists.

Acknowledgement

The authors would like to thank the reviewers for their helpful comments. The authors would also like to thank Satish Nadathur for his help with sorting algorithms, Jacopo Urbani for his help with WebPIE and QueryPIE and the authors of RDFox for their help in configuring their system.

8. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *PVLDB*, pages 411–422, 2007.

[2] D. Allemang and J. Hendler. *Semantic web for the working ontologist: effective modeling in RDFS and OWL*. Elsevier, 2011.

[3] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.

[4] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov. OWLIM: A family of scalable semantic repositories. *Semantic Web*, pages 33–42, 2011.

[5] C. Bizer and A. Schultz. The Berlin SPARQL benchmark. *IJSWIS*, 2009.

[6] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient RDF store over a relational database. In *SIGMOD*, pages 121–132, 2013.

[7] J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying RDf and RDF schema. In *ISWC*, pages 54–68. 2002.

[8] S. Dar and R. Ramakrishnan. A performance study of transitive closure algorithms. In *SIGMOD Record*, pages 454–465, 1994.

[9] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and Oranges: A Comparison of RDF Benchmarks and Real RDF Datasets. In *SIGMOD*, pages 145–156, 2011.

[10] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, pages 17–37, 1982.

[11] E. L. Goodman and D. Mizell. Scalable in-memory RDFS closure on billions of triples. In *SSWS*, pages 17–31, 2010.

[12] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, pages 158–182, 2005.

[13] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In *SIGMOD*, pages 289–300, 2014.

[14] A. Kiryakov, B. Bishop, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov. The features of BigOWLIM that enabled the BBC's World Cup website. In *Workshop on Semantic Data Management*, 2010.

[15] L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, and S. Liu. *Towards a complete OWL ontology benchmark*.

[16] B. McBride. Jena: Implementing the RDF Model and Syntax Specification. In *SemWeb*, 2001.

[17] B. Motik, Y. Nenov, R. Piro, I. Horrocks, and D. Olteanu. Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In *Proc. AAAI*, pages 129–137, 2014.

[18] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, pages 647–659, 2008.

[19] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, pages 627–640, 2009.

[20] E. Nuutila. *Efficient transitive closure computation in large digraphs*. PhD thesis, Helsinki University, 1995.

[21] M. Peters, C. Brink, S. Sachweh, and A. Zündorf. Scaling Parallel Rule-Based Reasoning. In *The Semantic Web: Trends and Challenges*, pages 270–285. 2014.

[22] S. P. Ponzetto and M. Strube. Deriving a large scale taxonomy from Wikipedia. In *AAAI*, pages 1440–1445, 2007.

[23] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*, pages 351–362, 2010.

[24] J. F. Sequeda, M. Arenas, and D. P. Miranker. OBDA: Query Rewriting or Materialization? In Practice, Both! In *ISWC 2014*, pages 535–551. 2014.

[25] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *PVLDB*, pages 1553–1563, 2008.

[26] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, 2007.

[27] J. Urbani, S. Kotoulas, J. Maassen, F. Van Harmelen, and H. Bal. OWL reasoning with WebPIE: calculating the closure of 100 billion triples. In *ESWC*, pages 213–227. 2009.

[28] J. Urbani, F. Van Harmelen, S. Schlobach, and H. Bal. QueryPIE: Backward reasoning for OWL Horst over very large knowledge bases. *The Semantic Web–ISWC 2011*, pages 730–745, 2011.

[29] W3C. OWL 1.1 Tractable Fragments. `http://www.w3.org/Submission/owl11-tractable/`, Dec. 2006.

[30] W3C. OWL 2 Language Primer. `http://www.w3.org/TR/owl2-primer/`, Dec. 2012.

[31] W3C. RDF Semantics. `http://www.w3.org/TR/rdf-mt/`, 2014.

[32] J. Weaver and J. A. Hendler. Parallel materialization of the finite rdfs closure for hundreds of millions of triples. In *ISWC*, pages 682–697. 2009.

[33] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, pages 1008–1019, 2008.

[34] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. TripleBit: a fast and compact system for large scale RDF data. *PVLDB*, pages 517–528, 2013.

#	RDFS	ρdf	RDFS+	Name	Body	Head	Class
1	○	○	●	**CAX-EQC1**	c_1 owl:equivalentClass c_2 x rdf:type c_1	x rdf:type c_2	α
2	○	○	●	**CAX-EQC2**	c_1 owl:equivalentClass c_2 x rdf:type c_2	x rdf:type c_1	α
3	●	●	●	**CAX-SCO**	c_1 rdfs:subClassOf c_2 x rdf:type c_1	x rdf:type c_2	α
4	○	○	●	**EQ-REP-O**	o_1 owl:sameAs o_2 s p o_1	s p o_2	–
5	○	○	●	**EQ-REP-P**	p_1 owl:sameAs p_2 s p_1 o	s p_2 o	–
6	○	○	●	**EQ-REP-S**	s_1 owl:sameAs s_2 s_1 p o	s_2 p o	–
7	○	○	●	**EQ-SYM**	x owl:sameAs y	y owl:sameAs x	–
8	○	○	●	**EQ-TRANS**	x owl:sameAs y y owl:sameAs z	x owl:sameAs z	θ
9	●	●	●	**PRP-DOM**	p rdfs:domain c x p y	x rdf:type c	γ
10	○	○	●	**PRP-EQP1**	p_1 owl:equivalentProperty p_2 x p_1 y	x p_2 y	δ
11	○	○	●	**PRP-EQP2**	p_1 owl:equivalentProperty p_2 x p_2 y	x p_1 y	δ
12	○	○	●	**PRP-FP**	p rdf:type owl:FunctionalProperty x p y_1 x p y_2 with ($y1 \neq y2$)	y_1 owl:sameAs y_2	–
13	○	○	●	**PRP-IFP**	p rdf:type owl:InverseFunctionalProperty x_1 p y x_2 p y with ($x1 \neq x2$)	x_1 owl:sameAs x_2	–
14	○	○	●	**PRP-INV1**	p_1 owl:inverseOf p_2 x p_1 y	y p_2 x	δ
15	○	○	●	**PRP-INV2**	p_1 owl:inverseOf p_2 x p_2 y	y p_1 x	δ
16	●	●	●	**PRP-RNG**	p rdfs:range c x p y	y rdf:type c	γ
17	●	●	●	**PRP-SPO1**	p_1 rdfs:subPropertyOf p_2 x p_1 y	x p_2 y	γ
18	○	○	●	**PRP-SYMP**	p rdf:type owl:SymetricProperty x p y	y p x	γ
19	○	○	●	**PRP-TRP**	p rdf:type owl:TransitiveProperty x p y y p z	x p z	θ^*
20	●	○	●	**SCM-DOM1**	p rdfs:domain c_1 c_1 rdfs:subClassOf c_2	p rdfs:domain c_2	α
21	●	●	●	**SCM-DOM2**	p_2 rdfs:domain c p_1 rdfs:subPropertyOf $p2$	p_1 rdfs:domain c	α
22	○	○	●	**SCM-EQC1**	c_1 owl:equivalentClass c_2	c_1 rdfs:subClassOf c_2 c_2 rdfs:subClassOf c_1	–
23	○	○	●	**SCM-EQP2**	c_1 rdfs:subClassOf c_2 c_2 rdfs:subClassOf c_1	c_1 owl:equivalentProperty c_2	β
24	○	○	●	**SCM-EQP1**	p_1 owl:equivalentProperty p_2	p_1 rdfs:subPropertyOf p_2 p_2 rdfs:subPropertyOf p_1	–
25	○	○	●	**SCM-EQP2**	p_1 rdfs:subPropertyOf p_2 p_2 rdfs:subPropertyOf p_1	p_1 owl:equivalentProperty p_2	β
26	●	○	●	**SCM-RNG1**	p rdfs:range c_1 c_1 rdfs:subClassOf c_2	p rdfs:range c_2	α
27	●	●	●	**SCM-RNG2**	p_2 rdfs:range c p_1 rdfs:subPropertyOf p_2	p_1 rdfs:range c	α
28	●	●	●	**SCM-SCO**	c_1 rdfs:subClassOf c_2 c_2 rdfs:subClassOf c_3	c_1 rdfs:subClassOf c_3	θ
29	●	●	●	**SCM-SPO**	p_1 rdfs:subPropertyOf p_2 p_2 rdfs:subPropertyOf p_3	p_1 rdfs:subPropertyOf p_3	θ
30	○	○	◐	**SCM-CLS**	c rdf:type owl:Class	c rdfs:subClassOf c c owl:equivalentClass c c rdfs:subClassOf owl:Thing owl:Nothing rdfs:subClassOf c	–
31	○	○	◐	**SCM-DP**	p rdf:type owl:DatatypeProperty	p rdfs:subPropertyOf p p owl:equivalentProperty p	–
32	○	○	◐	**SCM-OP**	p rdf:type owl:ObjectProperty	p rdfs:subPropertyOf p p owl:equivalentProperty p	–
33	◐	◐	◐	**RDFS4**	x p y y rdf:type rdfs:Ressource	x rdf:type rdfs:Ressource	–
34	◐	○	○	**RDFS8**	x rdf:type rdfs:Class	x rdf:type rdfs:Ressource	–
35	◐	○	○	**RDFS12**	x rdf:type rdfs:ContainerMembershipProperty	x rdfs:subPropertyOf rdfs:member	–
36	◐	○	○	**RDFS13**	x rdf:type rdfs:Datatype	x rdfs:subClassOf rdfs:Literal	–
37	◐	○	○	**RDFS6**	x rdf:type rdf:Property	x rdf:subPropertyOf x	–
38	◐	○	○	**RDFS10**	x rdf:type rdfs:Class	x rdfs:subClassOf x	–

Table 5: Composition of the rulesets supported by Inferray. Half circle denotes rules that do not produce meaningful triples and are used only in *full* versions of rulesets.