Optimizing Enterprise-scale OWL 2 RL Reasoning in a Relational Database System

Vladimir Kolovski[1], Zhe Wu[2], George Eadon[1]
Oracle
[1] 1 Oracle Drive, Nashua, NH 03062 USA
[2] 400 Oracle Parkway, Redwood City, CA 94065 USA
{vladimir.kolovski, alan.wu, george.eadon}@oracle.com

Abstract. OWL 2 RL was standardized as a less expressive but scalable subset of OWL 2 that allows a forward-chaining implementation. However, building an enterprise-scale forward-chaining based inference engine that can 1) take advantage of modern multi-core computer architectures, and 2) efficiently update inference for additions remains a challenge. In this paper, we present an OWL 2 RL inference engine implemented inside the Oracle database system, using novel techniques for parallel processing that can readily scale on multi-core machines and clusters. Additionally, we have added support for efficient incremental maintenance of the inferred graph after triple additions. Finally, to handle the increasing number of owl:sameAs relationships present in Semantic Web datasets, we have provided a hybrid in-memory/disk based approach to efficiently compute compact equivalence closures. We have done extensive testing to evaluate these new techniques; the test results demonstrate that our inference engine is capable of performing efficient inference over ontologies with billions of triples using a modest hardware configuration.

1 Introduction

As part of the OWL 2 [9] standardization effort, three new, less expressive OWL subsets were proposed that have polynomial (or less) complexity and are suitable for efficient and scalable reasoning over large datasets [12]. These profiles are OWL 2 EL, based on the EL++ description logic [7], OWL 2 QL based on DL-Lite [5] and OWL 2 RL, which was designed with rule-based implementations in mind.

Since it is described as a collection of positive Datalog rules, OWL 2 RL can be theoretically implemented on top of semantic stores that already provide rule-based reasoning. One of these semantic inference engines is Oracle's Semantic Technologies offering [10], which has supported inference over scalable rule-based subsets of OWL since Oracle Database 11g Release 1. Oracle's inference engine pre-computes and materializes all inferences using forward chaining, and later uses the materialized graph for query answering[1].

There are several challenges in handling enterprise-scale OWL 2 RL reasoning:

- OWL 2 RL supports equivalence relations such as owl:sameAs or owl:equivalentClass. With the emergence of inter-connected Linked Data and its heavy use of owl:sameAs, it becomes increasingly difficult to fully materialize owl:sameAs closures. A naïve representation of the closure could

[1] Note that our focus is not on query time inference; therefore we have not incorporated techniques such as magic sets rewriting.

be $O(N^2)$ in the size of the original triple set; we have observed these owl:sameAs blowups using UniProt[2] and OpenCyc [24] data.

- New RDF data is being published at an increasing rate; efficiently reasoning through such updates becomes a bottleneck if the inference closure needs to be maintained. There exists previous work on optimizing Datalog reasoning through updates using semi-naïve evaluation (see [21] for a survey); however it has neither been applied nor evaluated in an OWL setting using large-scale datasets and complex rule sets.
- Since OWL 2 RL has more than 70 rules, performing RL inference on billion-triple sized datasets could take hours to finish. With the proliferation of multi-core and multi-CPU machines, an approach is needed that could efficiently parallelize OWL 2 RL inference so that it could readily scale by adding more processors to the inference engine.

In this paper, we present a new[2] version of the inference engine built inside Oracle Database that supports OWL 2 RL and addresses the above challenges. The main contributions are the following:

Compact Materialization of Equivalence Closures - We address the challenge of efficiently computing owl:sameAs equivalence closures on massive scales by providing a hybrid (memory and disk-based) algorithm for generating compressed closures and integrating it with the general forward chaining inference engine.

Incremental Maintenance of Inferred Closure – We have developed a technique to efficiently update the inferred graph after triple additions to the underlying data model. Our technique is based on semi-naïve evaluation, with additional optimizations such as lazy duplicate elimination and dynamic semi-naïve evaluation.

Parallel Inference - We have parallelized the rules engine by leveraging Oracle's support for parallel SQL execution [17], which scales well with modern multi-core and multi-CPU architectures. To this end, we developed novel rule optimization techniques specifically aiming at parallel execution of queries. We also developed a source table design to align the structure of the table that stores semantic models with the table that stores intermediate temporary data generated during inference. Finally, we developed optimizations to reduce the data storage footprint of inference to reduce memory and I/O consumption.

Note that no knowledge of Oracle internals is needed to apply the techniques presented in this paper. Thus, they should be applicable to any RDBMS-based OWL 2 RL implementation (except for parallel inference, which assumes that the underlying database has support for parallel query evaluation).

We evaluated the new features using datasets with billions of triples, including versions of the LUBM ontology benchmark, UniProt ontology and various other real world datasets. With the optimized handling of equivalence closures, inference over owl:sameAs-heavy datasets that was extremely time and space consuming in previous versions of Oracle can now be done in minutes. We also show that our incremental OWL 2 RL inference over graphs of 1 billion triples takes less than 30 seconds to up-

[2] The algorithms described in this paper along with full support for the OWL 2 RL/RDF entailment and validation rules are available in an Oracle Database 11g Release 2 patch and will be part of the next release.

date the inferred graph. Finally, in the empirical evaluation section, we demonstrate the advantages of using parallel inference; this allows us to perform inference faster on less powerful hardware than well-known triple store vendors [3].

2 Preliminaries

2.1. OWL 2 RL

OWL 2 RL is a profile of OWL 2 aiming at applications that require scalable reasoning, efficient query answering, and more expressiveness than RDF(S), without needing the full expressive power of OWL 2. The specification [12] provides a partial axiomatization of the OWL 2 RDF-Based Semantics in the form of first order implications, called the OWL 2 RL/RDF rules.

The OWL 2 RL/RDF rule set is a superset of the non-trivial RDF(S) rules [22]; total number of rules in the partial axiomatization of OWL 2 RL is 78, compared to the 14 rules defined for RDF(S). In addition to supporting all of the RDF(S) constructs (except for axiomatic triples which are omitted for performance reasons), OWL 2 RL also supports inverse and functional properties, keys, existential and value restrictions, and owl:intersectionOf, owl:unionOf to some extent. For a lack of space, we will not enumerate all of the OWL 2 RL rules; we refer the reader to the standard specification [12] for more information. Inference and query answering has polynomial data complexity for OWL 2 RL.

2.2. Oracle Semantic Technologies

Oracle Semantic Technologies [23] provides a semantic data management framework in Oracle Database that supports storing, querying, and inferencing of RDF/OWL data via either SQL or Java APIs. It allows users to create one or more semantic models to store an RDF dataset or OWL ontology. The built-in native inference engine allows inference on semantic models using OWL, SKOS, RDF(S), and user-defined rules. The semantic model (and/or *entailed semantic model*, that is, model data plus inferred data) is materialized and can be queried using either SPARQL query patterns embedded in SQL or standard SPARQL query interface in Java. Oracle also supports ontology-assisted querying over enterprise relational data and semantic indexing of documents.

Inference Engine: The semantic inference engine [10] in Oracle 11g Database is based on forward chaining. It compiles entailment rules directly to SQL and uses Oracle's native cost-based SQL optimizer to choose an efficient execution plan for each rule. Various optimizations were added to improve performance and scalability:

- Dependency Graph – We developed a dependency graph such that we only apply a rule in round *n* if in round *n-1* there have been new inferences for at least one of the predicates contained in the rule's body.
- Using a Partitioned, Un-indexed Table – A temporary table is used to materialize all inferences while applying the inference rules. This table is partitioned by predicate to allow efficient queries, but is not indexed, since inserting inferred triples in an indexed table significantly slows down total inference time.

– Optimized Transitive Closure Evaluation – this optimization is critical for predicates such as rdfs:subClassOf. Instead of using hierarchical queries natively provided by Oracle Database, we discovered that implementing semi-naïve evaluation [21] to compute transitive closure results in better performance.

The following notation is used throughout this paper to refer to various data structures maintained in the semantic store: *M* refers to a single semantic model, i.e., an RDF graph containing asserted instance and schema triples. I(M), or *I* for short, refers to the entailed OWL 2 RL graph for M which contains only the materialized inferred triples. *PTT* is the partitioned, un-indexed temporary table that stores inferred triples during inference. *D* and *DI* are related to incremental inference: D stores the triples added to M since the last inference call, and DI contains the triples inferred in the current inference round.

3 Optimized Equivalence Reasoning

For equivalence relations such as owl:sameAs, owl:equivalentProperty or owl:equivalentClass, fully materializing the equivalence closures can be problematic for large datasets. In general, given a connected RDF graph with N resources using only owl:sameAs relationship, there will be $O(N^2)$ inferred owl:sameAs triples. Note that the alternative of searching the RDF graph at query time to determine if two URIs are equivalent is not feasible because of the interactions among owl:sameAs inferences with other rules in OWL 2 RL. This will require a query rewriting approach, which given the large number of rules in OWL 2 RL will slow down queries.

Each group of owl:sameAs-connected resources represents a *clique*; when doing full materialization the cliques' sizes (number of owl:sameAs triples) can grow quite large. For instance, the Oracle 11g inference engine [10] exhausts disk space (500GB) before completing the owl:sameAs closure for the benchmark ontology UniProt 80M [2]. Note that this version of UniProt80 contains a clique of size 22,000+ individuals so that a full materialization generates more than 480 million triples.

Our approach to handling equivalence closures is based on partial materialization. Instead of materializing the cliques, we choose one resource (individual) from each clique as a representative and all of the inferences for that clique are consolidated using that representative. The idea behind this partial materialization has been explored in previous work [3, 6, 8]; our novel contribution is in developing a hybrid (memory and disk-based), scalable approach for building the owl:sameAs[3] cliques.

Following, we discuss how we solve the technical challenge of large scale clique building, that is: given an arbitrarily large input of owl:sameAs pairs, efficiently build a map $\rho : ID \rightarrow ID$ which will take an ID[4] of a subject, a property or an object as input and return the corresponding clique representative ID. Note while building ρ, we maintain an invariant that $x \geq \rho(x)$.

[3] For brevity, we will only be discussing owl:sameAs closures in the rest of this paper, but our approach is applicable to other equivalence relations such as (owl:equivalentClass).

[4] Note that in our internal storage structures, URIs and literals are mapped to number-based IDs for performance reasons.

3.1. Large Scale Clique Building

The main challenges in building owl:sameAs cliques are that 1) a pure memory based approach does not scale due to memory size limitations, and 2) a pure SQL based approach is not efficient because of the performance implications of many joins on input required to build ρ.

Our proposed solution uses a hybrid approach – we load batches of owl:sameAs assertions (where the batch size is a tunable parameter) from the input table, merge each batch in memory and then append the generated cliques to ρ, which is stored as a table. After all batches are processed, there may be owl:sameAs relationships across different cliques. To capture these cases, we again employ batch processing on the ρ table itself, merging where needed, until we reach a fixpoint.

The flow of the algorithm is as follows:

```
function build_cliques (I)
I :    input table containing owl:sameAs pairs
ρ :    empty map (resource_id -> clique_id)

1. Read batch B from I
      a. ρ_B = Merge(B); b. Append ρ_B to ρ
2. Repeat 1 until no batches left in I
3. Loop
      a. Select batch of merge candidates B from ρ
      b. ρ_B = Merge(B)
      c. Update ρ with ρ_B
4. Repeat 3 until no more merges possible in ρ
5. return ρ
```

`Merge` is done in memory using the Union-Find algorithm [8, 27]. Given an input of equivalence relations (i.e., owl:sameAs assertions), `Merge` builds a map of resources to clique representatives such that given a resource, retrieving its representative is done using only one lookup. The algorithm has time complexity of O(N log N) and polynomial space complexity, however using path compression [8] we achieved almost linear performance in our testing.

After steps 1 and 2, ρ is not fully merged since there may be inter-clique merges remaining. For instance, if one clique contains A owl:sameAs B and another contains A owl:sameAs C, then B and C should belong to the same clique and they will be selected as merge candidates. Additionally, if one clique contains A owl:sameAs B and another contains B owl:sameAs C, then A and C should be in the same clique and they will also be selected as merge candidates. In step 3c, ρ is updated with the merged in-memory map ρ_B. This is done using an OUTER JOIN where, for each key x in ρ_B, ρ(x) is replaced by ρ_B(x).

After ρ has been built, we update the asserted and inferred graph with the new information, replacing resources x with their clique representatives ρ(x).

Performance On a UniProt 80 million triple sample, the optimized owl:sameAs approach took 26 minutes to finish inference, producing 61 million consolidated

triples. More than 100,000 cliques were generated with an average membership size of 5.6; the largest owl:sameAs clique had 22,064 resources. The storage savings compared to a full materialization of the owl:sameAs closure are more than 95%. More evaluation results are shown in Section 6.

4 Parallel OWL Inference

An extensive performance evaluation of the previous version of Oracle's inference engine (11g Release 1) on a server class machine with solid-state disk based storage revealed that the inference process is CPU-bound in such a setup. Thus, the native OWL inference engine needs to be parallelized to fully leverage hardware configurations that have multiple CPUs (cores) and high I/O throughput.

We explored several schemes to parallelize the native OWL inference process. Simply applying Oracle SQL engine's parallel execution capability to each inference rule (which is translated to a SQL query) without any modification to the inference algorithm did not produce any performance benefits. In the following subsections, we propose several new inference optimization techniques that successfully leverage Oracle's parallel execution engine. We believe they are general enough to be applied to any database supporting parallel query executions.

4.1. Query Simplification for Efficient Parallel Inference

After comparing the performance difference of all the rules running in serial and parallel mode, we observed that rules with smaller number of patterns in the body tend to have bigger performance gains when run in parallel. This observation leads to a new optimization to *simplify complex, multi-pattern rules*. Next, we provide an example of how this rule simplification by break up technique is used to optimize the parallel execution of the OWL 2 RL rule CLS-SVF1 (listed below):

```
T(?x,owl:someValuesFrom,?y)
T(?x,owl:onProperty,?p)
T(?u,?p,?v)
T(?v, rdf:type, ?y)   ➔      T(?u, rdf:type, ?x).
```

The first two patterns `T(?x, owl:someValuesFrom, ?y)` and `T(?x, owl:onProperty, ?p)` are much more selective compared to the rest. Intuitively, execution of this rule can be divided into two parts, where one part focuses on the selective patterns, and the other part focuses on the rest. After the selective sub-query is executed, the variable bindings are then used to further constraint the rest of the patterns. Putting this idea into context, a query can be executed to find all bindings for ?x, ?y, and ?p that satisfy the first two patterns `T(?x, owl:someValuesFrom, ?y)` and `T(?x, owl:onProperty, ?p)`. For each binding tuple (x, y, p) coming from the query result set, we execute the following rule in parallel:

`T(?u, p, ?v). T(?v, rdf:type, y)` ➔ `T(?u, rdf:type, x)`

Executing CLS_SVF1 in parallel mode using the hybrid approach described above is five times faster than running this rule as a single SQL statement.

This idea of breaking up a rule in two parts can easily be generalized to complex rules containing selective and unselective patterns in the rule body[5]. The pseudo code of the algorithm is as follows:

```
function find_sel_patterns (I, R) returns C
I   :  Input RDF graph containing asserted data
R   :  Set of triple patterns belonging to an OWL 2
       RL rule body
C   :  Candidate selective subset that is returned,
       initially empty
1. Estimate average out- and in- degree for subjects
   and property nodes respectively for each property
   in I
2. Estimate selectivity for each property in I by sam-
   pling
3. For each subset S of R
    If est(S, I) < threshold[6] then
      If cardinality(S) > cardinality(C) then C := S
      Else if cardinality(S) == cardinality(C) and
              est(S, I) < est(C, I) then C := S
4. Return C
```

Note that all of the rules in OWL 2 RL have less than 10 triple patterns in the body, so the search space for selective subsets is fairly small.

In the pseudo-code above, `est(S, I)` estimates the selectivity (size of return set) of a set of triple patterns S against a triple dataset I. We use a simple, conservative estimation technique where we start with the property count estimates and then we iteratively multiply by the average in- or out- degrees (depending on the position of the join variables). These property count and average in/out degree estimates are done once, when the first time `find_sel_patterns` is executed. Note that more sophisticated SPARQL selectivity estimation methods like [26] could be used here.

The idea of query simplification also applies to those rules, including CLS-INT1 [12], with recursive/hierarchical structures that use rdf:list. Using CLS-INT1 as an example, instead of using a single complex SQL to find all ?y that satisfy `T(?y, rdf:type, ?C1)` … `T(?y, rdf:type, ?Cn)`, a series of simpler SQLs are used to first find all matches for the `T(?y, rdf:type, ?C1)` and then join this result set with the next pattern `T(?y, rdf:type, ?C2)`. This kind of operation is repeated until the last pattern `T(?y, rdf:type, ?Cn)` is processed. Apart from the query simplification, another benefit is that for ontologies containing tens of thousands or more owl:intersectionOf axioms, it is feasible to process all the axioms together in an iterative fashion. Details are omitted here due to space limitations.

[5] Note that we are not simply reordering the patterns; instead we find a selective subset of rule body patterns to be used as the driving query.

[6] Currently, we set the threshold for the selective triple pattern estimate to 1000. We do not use a larger number because we need to re-evaluate the second part of the rule for each binding produced by the selective part.

4.2. Compact Data Structures

An examination of the underlying table design shows a discrepancy between the table that holds the original semantic model(s) and the partitioned temporary table (PTT) that holds the intermediate inference results. Namely, PTT is partitioned using predicates while the semantic models are not.

To allow efficient parallel execution, we designed a single *source table* with the *same* structure as PTT; this source table contains all data of the original semantic model(s). Then, queries executed during inference only use this source table and PTT. This design change produced critical performance improvements for parallel inference. For example, rules that tend to generate many new inferred triples including RDFS2, RDFS3, RDFS9, RDFS11, RDFS7, PRP-INV1 [12], PRP-INV2 are running 30% ~ 60% faster when Oracle SQL engine's parallel query execution is turned on and the degree of parallelism (**DOP**[7]) is set to 4[8].

As an additional storage optimization, we use an 8-byte binary RAW type as a column type for the PTT and source table instead of a generic numeric type (NUMBER). RAW is an Oracle-specific native datatype which is returned as a hexadecimal string. This column type change saves more than 12% disk storage size using typical benchmark ontologies and this space saving directly translates into better inference performance.

As a final optimization, we also use *perfect reverse hashing*, based on the fact that the set of all generated resource IDs for even a large-scale ontology tends to be sparse (imagine 1 billion = 10^9 unique IDs scattered across a space consisting of 2^{64} which is roughly $1.8*10^{19}$ different values). Perfect reverse hashing provides additional storage savings by mapping the sparse ID values into a sequential set of values starting with 1. For example, assume the original data model has the following set of unique ID values: {10, 1009123, 834132227519661324, 76179824290317, 621011710366788}, where some of them require multiple bytes for storage. If we map them to this sequence {1, 2, 3, 4, 5}, then one byte for each ID is sufficient. In our algorithm, we get the set of unique integer IDs out from the semantic models, map them into a set of sequential integer values, which are then stored in a variable length data type. Then, the RDBMS determines the number of bytes needed for storage. Note that the more compact table structures provided by perfect reverse hashing will improve serial inference as well.

5 Incremental Inference

Incremental inference tackles the following problem: Given a model M with a materialized inference graph I, how can we efficiently update I after a new set of triples D is added to M?

Our algorithm for incremental inference is based on semi-naïve evaluation [21]. The goal is to avoid re-deriving existing facts in I after an update. The following example illustrates the basic idea using the rule:

---

[7] Degree of parallelism (DOP) is an Oracle setting that specifies the number of parallel processes that should be used to execute a SQL statement.

[8] On a PC with dual-core CPU, three 1TB disks and 8GB RAM running 64-bit Linux.

X rdf:type C1. C1 rdfs:subClassOf C2 => X rdf:type C2
p_1 p_2 h

In "naïve" inference, the patterns p1 and p2 are both selected from M UNION I. For shorthand, we use $p^{A,B}$ to indicate that pattern p selects from relation A UNION B. After adding D to M, we know that the join $p_1^{M,I} \times p_2^{M,I}$ was already evaluated. Joining $p_1^{M,I,D} \times p_2^{M,I,D}$ means mostly re-deriving the same inferences.

To avoid redundant derivation, at least one predicate should select from the new set of triples D. The semi-naïve rule evaluation is done in two steps:

1) $h \leftarrow p_1^{D} \times p_2^{M,I,D}$
2) $h \leftarrow p_1^{M,I,D} \times p_2^{D}$

Given the assumption that D is small relative to M and I, this divide and conquer approach has the potential for significant performance improvements. We implemented two custom optimizations on top of this well-known evaluation algorithm in order to further improve performance.

5.1. Lazy Duplicate Elimination

During inference, inferred triples are checked to see if they already exist in M, I or PTT before the triples are inserted in PTT. This check usually involves a hash join which essentially scans through the M, I, and PTT tables. Given a small size of D, we assume that the number of triples inferred will be relatively small compared to M and I, so we allow duplicates to accumulate by not removing them after firing each rule. Instead, we perform the join to remove duplicates only once, at the end of each inference round.

Lazy duplicate elimination will introduce duplicates in DI during an inference round. However, our results (see Table 1) indicate that the duplicate overhead is acceptable since we do not have to perform duplicate elimination after each rule.

Model name (#triples)	Yago (19.9 million)	WordNet (1.9 million)	LUBM8000 (1.06 billion)
#Duplicate/#Unique triples	83,583 / 17,180	123,123 / 23,410	20,944 / 2,453

Table 1 Duplicate Triples in Incremental Inference. The number of newly asserted triples (i.e., delta size) is 10,000.

5.2. Dynamic Semi-Naïve Evaluation

The semi-naïve evaluation technique described in this section can also be used when performing inference from scratch, by treating the inferred triples in each round as delta D.

However, we observed that using semi-naïve evaluation for each inference round (we refer to it as *static* semi-naïve evaluation) is not always the optimal choice. This is because in the initial inference rounds the number of inferred triples |DI| could be quite large compared to the size of the asserted model(s) |M|; in such cases, when splitting and evaluating each rule the same execution plan (usually consisting of hash joins) might be used and it might be slower than evaluating the rule in one step. On

the other hand, if |DI| is small enough, the SQL optimizer will select a different plan where a nested loop join with index is used instead of hash join, which could dramatically improve performance when the driving table |DI| << |M|.

Thus, we *selectively* use semi-naïve evaluation depending on the number of triples inferred in an inference round. At the end of each round *r*, we use the following heuristic formula to determine whether to use semi-naïve evaluation in round *r+1*:

$$\frac{|DI|}{|PTT| + \sum_i |M_i|} < t \quad \textbf{(1)}$$

where the threshold *t* is set to 0.1 by default. In other words, if the number of triples inferred is less than 10% of the overall triple count (including cumulative inferences and asserted models), then we use semi-naïve evaluation in the following round.
Below, we demonstrate the benefits of dynamic semi-naïve evaluation compared to naive evaluation and to "static" semi-naïve evaluation (running times in seconds).

Dataset	Model Size	#Inf. Rounds	Dynamic Semi-Naive	Naïve	Static Semi-Naive
LUBM1000	133M	3	**3,628**	4,497	4,996
Yago	19.9M	2	**981**	1,230	2,049
Wordnet	1.9M	6	**335**	427	605

Table 2 Evaluation results for dynamic semi-naïve evaluation.

6 Evaluation

This section presents the results of a performance study of the techniques presented in this paper using various real-world and synthetic semantic datasets.

6.1. Experimental Setup

We used 2 commodity PCs for our experiment; we refer to them as S1 and S2. Each runs Redhat Enterprise Linux v5 64 Bit (2.6.18-128). Each PC has Oracle Database 11.2 installed and three disks attached. We used Automatic Storage Management (ASM) to spread the I/O load across multiple disks.

	CPU	Memory	Disk
S1	Intel Core 2 Duo 2.13 GHz	6GB	750GB
S2	Intel Core 2 Quad 2.4 GHz	8GB	3TB

The databases were setup with a block size of 8k bytes. S1 had 2400M memory allocated to system global area (SGA), and 3200M to aggregated program global area (PGA) whereas S2 had allocated 3400M and 4400M to SGA and PGA.

In addition to the two commodity PC setups, we also use two server-class machines. S3 is a Sun 4150 server with dual quad core CPUs and Sun Storage 5100

Flash Array. S4 is a Sun Oracle Database Machine and Exadata Storage Server (Full Rack [9] with 8 nodes).

	CPU	Memory	Disk
S3	Intel Xeon CPU E5440 2.83GHz	32GB	1TB
S4[4]	Intel Xeon CPU E5540 2.53GHz	72GB each node	100 TB+

We used various real-world and synthetic datasets to evaluate our inference engine. Lehigh University Benchmark [4] is used frequently to evaluate performance of semantic stores; we evaluated against LUBM1000, LUBM8000, LUBM25K and LUBM50K where each has 133M, 1.1B, 3.3B and 6.6B triples respectively. We used Yago (20M), OpenCyc [24] (1.5M), Wordnet (1.9M) and UniProt [2] (two versions, one of 80M, another of 740M) as real-world datasets.

6.2. Parallel Inference Evaluation

We evaluated parallel inference on UniProt 740M and various sizes of LUBM. On server class machine S3, we measure the performance improvement as DOP changes from 1 to 8. Figure 1 shows that performance improves drastically as we move from a serial execution to parallel execution with DOP set to 8.

The evaluation results achieved using machine S2 results are shown in Figure 2. In the case of LUBM8000, inference time drops from 42 hours to 11 hrs when using parallel inference. We observe similar improvements with LUBM25000. In all cases, the parallel inference is run with DOP=4.

Using the parallel inference optimizations, we are able to achieve comparable performance to other triple stores while using much weaker hardware: e.g., BigOWLIM uses a server machine while reporting similar inference performance numbers to ours for LUBM25000 and LUBM8000 [15].

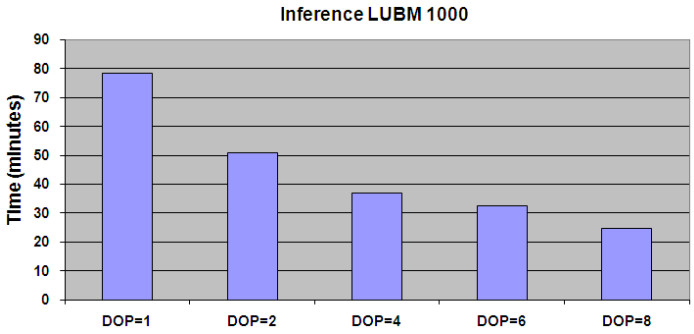

Figure 1 Inference Performance on server S3

[9] http://www.oracle.com/technology/products/bi/db/exadata/pdf/exadata-technical-whitepaper.pdf

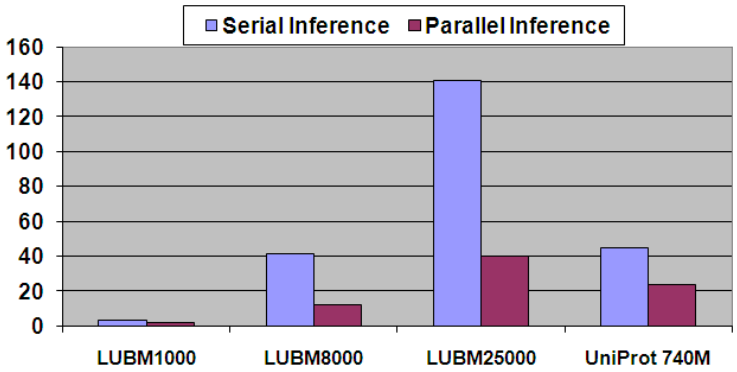

Figure 2 Parallel Inference Performance when DOP=4

We also evaluated parallel inference performance on server-class machine S4. Due to time limitations, we only collected a few data points. The results nonetheless prove the effectiveness of the parallel inference engine inside Oracle database and the scalability of the particular server-class machine tested.

Benchmark	Parallel Inference Time with S4
LUBM 8000, DOP = 64	46 minutes and 23 seconds
LUBM 25000, DOP = 32	247 minutes and 9 seconds

6.3. Incremental Inference Evaluation

This section contains the incremental inference performance results. The evaluation was done on server S1. For each dataset, we removed a number of triples, performed inference on the remaining dataset and then added back the removed triples in batches of various sizes while measuring the time needed to update the inference graph. We performed this three times and measured the average incremental inference time for each batch. Results are presented in Figure 3.

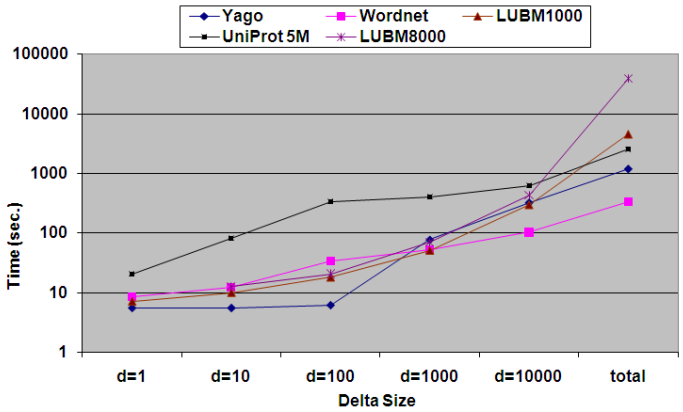

Figure 3 Incremental Inference Evaluation. As a reference, we also show total (non-incremental) inference time when building the inference graph from scratch.

Our evaluation shows that updating the inference graph is orders of magnitude faster using the incremental inference techniques. For instance, even in the case of a 1 billion triple dataset like LUBM8000, we are able to update the inference graph in less than 20 seconds if the delta is less than 100. Even when adding 10,000 triples the inference update time takes only a few minutes (compared to 11 hours needed to build LUBM8000 inference graph from scratch).

6.4. Optimized owl:sameAs Handling Evaluation

To evaluate our optimized owl:sameAs handling techniques, we used the following datasets: UniProt 80 million, UniProt 740M and OpenCyc. All three of these datasets have more than 100,000 asserted owl:sameAs triples. Performance results are shown in Table 3. Without the optimized owl:sameAs handling, UniProt 80M and OpenCyc did not finish inference: they exhausted disk space (500GB) after running for 40+ hrs, and UniProt 740M took 24 hours to finish inference.

Dataset (#triples)	UniProt (80 Million)	UniProt (740 Million)	OpenCyc (1.5 Million)
owl:sameAs Closure # of Triples	2,129,166,152	42,159,397	295,540,812
owl:sameAs Compressed Closure # of Triples	766,905	12,282,537	395,527
Inferred Triple Size	63,161,568	740,269,215	91,192,106
Cliques Building Time	29 sec	6 min	39 sec
Total Inference Time	25min 48sec	4hr 14min	13 hr 47min

Table 3. Performance Results for Optimized owl:sameAs Handling.

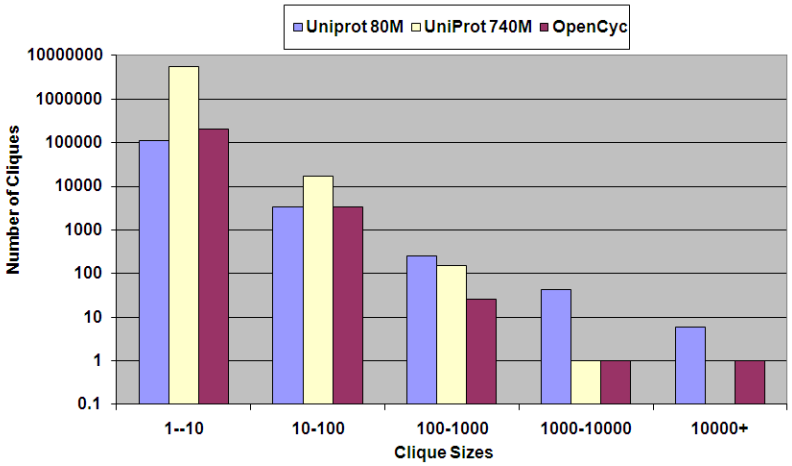

Figure 4. Distributions of Clique Sizes.

Figure 4 shows the distribution of number of cliques across bins of various clique sizes. As expected, most of the cliques in all three datasets have less than 10 mem-

bers. Interestingly, UniProt80m and OpenCyc have a surprising number of cliques larger than 1000. In the case of UniProt80m, the largest clique is of size 22,065 and can blow up to 486 million triples when fully materialized. In the more recent, updated version of UniProt (with 740million triples), the modeling issues leading to these large cliques seems to have been fixed; almost all cliques have less than 1000 resources.

The latest version of OpenCyc [24], on the other hand, seems to contain some modeling issues. Apart from the largest clique containing 17,030 resources, many of the resources in that clique are plain literals with distinct values.

7 Related Work

Due to a lack of space, in this section we only provide a survey of the semantic stores and inference engines most closely related to our work.

Jena [11,14] is a Java framework for Semantic Web Applications. In addition to providing an API for RDF, RDFS, OWL and SPARQL, it includes a rule-based inference engine; the inference engine can use both forward and backward chaining, and it supports the most common OWL constructs. Additionally it allows users to define their own custom rules, however it does not natively support any constructs introduced in OWL 2. Like our inference engine Jena supports incremental maintenance (when the forward-chaining RETE-based engine is used); unlike our engine Jena does not optimize owl:sameAs handling.

Sesame [13] is a semantic data repository for RDF and RDFS. Inference wise, it does not support OWL and OWL 2 constructs as we do. It provides an inference engine for RDFS that uses forward chaining and materialization of the data. In [13], an algorithm is proposed for truth maintenance of RDFS data, which could be used to optimize reasoning after updates.

BigOWLIM [3] is a semantic repository that is fully compatible with the Sesame RDF framework. It supports RDFS, some OWL constructs, and extensions with user-defined rules. BigOWLIM's inference engine materializes inferred triples using forward chaining.

BigOWLIM has reported results for the LUBM benchmark [15]. For example, BigOWLIM 3.1 can load, infer, and store the LUBM 8000 dataset in 14.4 hours on a desktop machine. However, their approach seems to require a much larger memory footprint when operating against large ontologies [15] compared to ours.

AllegroGraph [1] is a persistent triple store that can handle large RDF knowledge bases. It has inferencing capabilities that extend beyond RDFS, including custom rules and some OWL constructs, but does not natively support any constructs introduced in OWL 2. Virtuoso Open Link Server [18] is a persistent triple store that scales well on multiple machines but it also provides limited inference support (rdfs:subClassOf, owl:sameAs and rdfs:subPropertyOf constructs).

The Web-Scale Parallel Inference Engine (WebPIE) - although not a true RDF repository because it lacks query capabilities - shows the power of massive parallelism for OWL reasoning. WebPIE [25] is able to infer 4.97 billion triples from a 10 billion-triple LUBM data set in 4.06 hours, using a 64-node cluster. As a comparison, in an inference run using the server-class machine S4, Oracle's parallel inference engine is

able to infer, in one inference round, 5.5 billion triples from a 13 billion-triple LUBM data set in 1.97 hours, using DOP=32. The same university ontology and the same OWL Horst semantics were chosen to make the comparison meaningful. We plan to do more testing using high performance platforms like S4.

While our inference engine is to able to cover the whole OWL 2 RL profile, for applications that need additional expressiveness there has been recent work in coupling OWL 2 DL reasoner Pellet [19] with the OWLPrime inference engine in Oracle Database 11g. The scalable-yet-expressive engine PelletDB [20] uses Pellet to compute the class hierarchy and Oracle for Abox reasoning and instance query answering.

Recently a system has been proposed for parallel inference in shared-nothing clusters using existing systems for local computation on each node [16]. The parallelism within our system could work with this to take advantage of multi-core machines within a shared-nothing cluster.

8 Conclusions and Future Work

This paper described the next generation OWL 2 RL inference engine, implemented in the Oracle Database, capable of handling ontologies with billions of triples. We described a number of techniques that we developed to make this engine enterprise-scale, incremental and parallelized. Additionally, to accommodate the high degree of owl:sameAs interlinking between semantic datasets, we implemented a novel scalable, hybrid in-memory/disk-based approach that can compute compact equivalence closures. Using this owl:sameAs approach we were able to discover some modeling issues in real world datasets (e.g., OpenCyc). Our final contribution consists of a thorough evaluation of all our techniques on large-scale real world and synthetic RDF/OWL datasets.

As part of future work, we plan to develop an efficient technique to update inference graphs in presence of deletions. Additionally, we plan to investigate how we can generalize our approach and extend our inference engine to cover the remaining OWL 2 profiles (EL and QL). Finally, the optimization techniques described in this paper are only applied to the axiomatic rules of OWL 2 RL, OWLPrime and RDF(S). We plan to generalize the approach to cover user-specified rules and evaluate it using the OpenRuleBench suite [28].

Acknowledgement: We thank Jay Banerjee for his continuous support and suggestions. We thank Tim Cline for his help in providing server-class machines S3 and S4.

9 References

[1] AllegroGraph. [Online]. Available at: http://www.franz.com/products/allegrograph/

[2] The UniProt Consortium. The Universal Protein Resource (UniProt). In *Nucleic Acids Res. 36:D190-D195(2008)*. Available at: http://www.UniProt.org/

[3] A. Kiryakov, D. Ognyanov, and D. Manov, "OWLIM – a Pragmatic Semantic Repository for OWL", in P*roc. International Workshop on Scalable Semantic Web Knowledge Systems (SSWS 2005)*, New York City, USA.

[4] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. Journal of Web Semantics 3(2), 2005, pp. 158-182.

[5] Diego Calvanese, Giuseppe de Giacomo, Domenico Lembo, Maurizio Lenzerini, Riccardo Rosati . *Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family*. J. of Automated Reasoning 39(3):385–429, 2007
[6] M. Stocker, M. Smith, *Owlgres: A Scalable OWL Reasoner*. In Proc. Of OWL Experiences and Directions EU (OWLED-EU), 2008.
[7] Franz Baader, Sebastian Brandt, and Carsten Lutz. *Pushing the EL Envelope*. In Proc. of the 19th Joint Int. Conf. on Artificial Intelligence (IJCAI 2005), 2005
[8] R. Tarjan. *A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets*. Journal of Computer and System Sciences, 18(2):110-127,1979.
[9] Boris Motik, Peter F. Patel-Schneider, Bijan Parsia, eds. *OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax*. Latest version available at http://www.w3.org/TR/owl2-syntax/.
[10] Zhe Wu, George Eadon, Souripriya Das, Eugene Inseok Chong, Vladimir Kolovski, Melliyal Annamalai, Jagannathan Srinivasan, *Implementing an Inference Engine for RDFS/OWL Constructs and User-Defined Rules in Oracle*, ICDE, pp.1239-1248, 2008 IEEE 24th International Conference on Data Engineering, 2008
[11] Jena Framework. [Online]. Available: http://jena.sourceforge.net/
[12] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, Carsten Lutz, eds .*OWL 2 Web Ontology Language: Profiles*.. Latest version available at http://www.w3.org/TR/owl2-profiles/.
[13] J. Broekstra, F. van Harmelen, and A. Kampman, “Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema”, in Proc. First ISWC, 2002.
[14] K. Wilkinson, C. Sayers, H. Kuno, H., and D. Reynolds, “Efficient RDF storage and retrieval in Jena”, in Proc. VLDB Workshop on Semantic Web and Databases, 2003.
[15] OWLIM: LUBM Tests. Available at: http://ontotext.com/owlim/benchmarking/lubm.html
[16] S. Narayanan, U. Catalyurek, T. Kurc, J. Saltz, "Parallel materialization of large ABoxes," in Proc. of the 2009 ACM symposium on Applied Computing, Honolulu, Hawaii, pages 1257-1261.
[17] Oracle SQL Parallel Execution. http://www.oracle.com/technology/products/bi/db/11g/pdf/twp_bidw_parallel_execution_11gr1.pdf
[18] Virtuoso Universal Server Platform. Available at: http://virtuoso.openlinksw.com/
[19] Pellet – Open Source OWL DL Reasoner. Available at: http://clarkparsia.com/pellet/
[20] PelletDB. More information at http://clarkparsia.com/pelletdb
[21] S Ceri, G Gottlob, L Tanca, *What you always wanted to know about Datalog (and never dared to ask)*. IEEE Transactions on Knowledge and Data Engineering 1(1), 1989.
[22] Patrick Hayes, Editor *RDF Semantics*, W3C Recommendation. Latest version available at http://www.w3.org/TR/rdf-mt/ .
[23] Oracle Semantic Technologies: http://www.oracle.com/technology/tech/semantic_technologies/index.html
[24] OpenCyc. Available at: http://www.opencyc.org/downloads
[25] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, H. Bal, “OWL reasoning with WebPIE: calculating the closure of 100 billion triples” in Proceedings of the ESWC '10
[26] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, D. Reynolds. SPARQL Basice Graph Pattern Optimization Using Selectivity Estimation. In Proc. of the World Wide Web Conference (WWW2008), April 21-15, 2008, Beijing, China.
[27] Christophe Fiorio and Jens Gustedt. Memory Management for Union-Find Algorithms. In Proceedings of the 14th Annual Symposium on theoretical Aspects of Computer Science. LNCS 1200 (1997). 67-79.
[28] Senlin Liang, Paul Fodor, Hui Wan, Michael Kifer. OpenRuleBench: An Analysis of the Performance of Rule Engines. In WWW'09. ACM Press (2009) 601-610.