Advancing the Enterprise-class OWL Inference Engine in Oracle Database

Zhe Wu, Karl Rieb, George Eadon
Oracle Corporation
{alan.wu, karl.rieb, george.eadon}@oracle.com

Ankesh Khandelwal, Vladimir Kolovski
Rensselaer Polytechnic Institute, Novartis Institutes for Bio-
medical Research
ankesh@cs.rpi.edu, vladimir.kolovski@novartis.com

Abstract.

OWL is a standard ontology language defined by W3C that is used for knowledge representation, discovery, and integration. Having a solid OWL reasoning engine inside a relational database system like Oracle is significant because 1) many relational techniques, including query optimization, compression, partitioning, and parallel execution, can be inherited and applied; and 2) relational databases are still the primary place holder for enterprise information and there is an increasing use of OWL for representing such information. Our approach is to perform data intensive reasoning as close as possible to the data. Since 2006, we have been developing an RDBMS-based large scale and efficient forward-chaining inference engine capable of handling RDF(S), OWL 2 RL/RDF, SKOS, and user defined rules. In this paper, we discuss our recent implementation and optimization techniques for query-rewrite based OWL 2 QL reasoning, named graph-based inference (local inference), and integration with external OWL reasoners.

1   Introduction

OWL [1] is an important standard ontology language defined by W3C and it has a profound use in knowledge representation, discovery, and integration. To support OWL reasoning over large datasets we have developed a forward-chaining rule-based inference engine [2] on top of the Oracle Database. By implementing the inference engine as a database application we are able to leverage the database's capabilities for handling large scale data. The most recent release of our engine, with support for RDFS, OWL 2 RL/RDF [3], SKOS and user-defined rules, is available as part of Oracle Database 11g Release 2 [4, 5, 7].

In our system semantic data is stored in a normalized representation, with one table named LexValues providing a mapping between lexical values and integer IDs and

another table named IDTRIPLES enumerating triples or quads in terms of IDs, similar to other systems [8, 9]. Inference engine rules are translated to SQL and passed to Oracle's cost-based optimizer for efficient execution. For notational convenience, SQL queries in this paper are written in terms of placeholders ID(x) and <IVIEW>. In the implementation ID(x) is replaced by the ID for the given lexical value x, which can be found by querying our LexValues table, and <IVIEW> is replaced by an inline view that unions the relevant triples (or quads) in IDTRIPLES with the inferred triples (or quads) computed so far.

Additional features of our inference engine include: (1) for built-in OWL constructs we manually craft the SQL and algorithms that drive the inference, optimizing for special cases including transitive properties and equivalence relations such as owl:sameAs, (2) user-defined rules are translated to SQL automatically, (3) we leverage Oracle's parallel SQL execution capability to fully utilize multi-CPU hardware, and (4) we efficiently update the materialized inferred triples after additions to the underlying data model, using a technique based on semi-naive evaluation [7].

Our engine has been used in production systems since 2006, and has proven capable of handling many real-world applications. However, challenges remain:

- Despite support for efficient incremental inference, fully materializing inferred results via forward chaining can be a burden, especially for large frequently-updated data sets. Therefore we are introducing backward-chaining into our system with a query-rewrite-based implementation of OWL 2 QL reasoning [2].
- Some applications need to restrict inference to a single ontology represented by a named graph. For these applications inference should apply to *just* the assertions in each named graph and a common schema ontology (TBox). This kind of inference is therefore local, as opposed to the traditional global inference, and it is called Named Graph based Local Inference (NGLI) by Oracle.
- Some applications need the full expressivity of OWL 2 DL. To satisfy these applications, we have further extended our inference engine by integrating it with third-party complete OWL 2 DL reasoners like PelletDB [6].

In this paper we present our recent advances. Section 2 describes our query-rewrite implementation of OWL 2 QL reasoning. Section 3 describes our implementation of named graph local inference. Section 4 describes integration with external third-party OWL reasoners. Section 5 presents a performance evaluation using synthetic Lehigh University Benchmark (LUBM) datasets. Section 6 describes related work. Finally, Section 7 concludes this paper.

2   Support of OWL 2 QL in the Context of SPARQL

OWL 2 QL is based on the DL-Lite family of Description Logics, specifically DL-Lite*R* [10]. OWL 2 QL is an important profile because it has been designed so that data (Abox) can be queried through an ontology (Tbox) via a simple rewriting mechanism. Queries can be expanded to include the semantic information in Tbox, using query-rewrite techniques such as the PerfectRef algorithm [10], before executing them against the Abox. The query expansion could produce many complex queries, which presents a challenge for scalable QL reasoning. There have been several pro-

posals for optimizing and reducing the size of rewritten queries; see Section 6 for a brief discussion. Most of these techniques return a union of conjunctive queries (UCQ) for an input conjunctive query (CQ). Rosati et.al. [11] proposed a more sophisticated rewriting technique, the `Presto` algorithm, that produces non-recursive datalog (nr-datalog), instead of UCQ. OWL 2 QL inference is supported in the Oracle OWL inference engine based on the Presto algorithm. We elaborate on this and other optimizations for efficient executions of query rewrites in Section 2.1. To meet the requirements of enterprise data, OWL 2 QL inference engine must handle arbitrary SPARQL queries. We discuss some subtleties to handling arbitrary SPARQL queries by query expansion in Section 2.2. We will be using the following OWL 2 QL ontology to illustrate various concepts. It is described in functional syntax and has been trimmed down for brevity.

```
Ontology (SubDataPropertyOf(:nickName  :name)
    SubClassOf(:Married ObjectSomeValuesFrom(:spouseOf :Person))
    SubObjectPropertyOf(:spouseOf :friendOf)
    DataPropertyAssertion(:name :Mary "Mary")
    ClassAssertion(:Married :John)
    DataPropertyAssertion(:name :Uli "Uli") )
```

2.1 Optimizing Execution of Query Rewrites

As noted in the introduction, Oracle OWL inference engine implements an approach that is similar to that of rewriting CQs as nr-datalog. We will illustrate our approach through some examples. Consider the conjunctive query (CQ),

```
select ?x ?y ?n where { ?x  :friendOf  ?y .   ?y  :name  ?n },
```

and its equivalent datalog query `q(?x, ?y, ?n) :- friendOf(?x, ?y), name(?y, ?n)`. Note that `friendOf(?x, ?y)` and `name(?y, ?n)` are referred to as atoms of the query. `q(?x, ?y, ?n)` can be translated into following nr-datalog using the `Presto` algorithm, and the example Tbox.

```
q(?x, ?y, ?n) :- q1(?x, ?y), q2(?y, ?n) .
q1(?x, ?y)    :- friendOf(?x, ?y) .
q1(?x, ?y)    :- spouseOf(?x, ?y) .
q2(?y, ?n)    :- name(?y, ?n) .
q2(?y, ?n)    :- nickName( ?y, ?n) .
```

The nr-datalog can be represented by a single SPARQL query as shown below. The heads of the nr-datalog, `q1(?x, ?y)` and `q2(?y, ?n)` for example, define a view for the atoms of the query, which can be represented via UNION operation. The conjunctions in the body of nr-datalog rules can be represented via intersections of views. We refer to the resulting form of SPARQL query as the Joins of Union (JoU) form. (UCQs, in contrast, are of the form Unions of Joins (UoJ).)

```
select ?x ?y ?n where { {{?x  :friendOf ?y} UNION {?x  :spouseOf ?y}}
                        {{?y  :name  ?n}    UNION {?y  :nickName ?n}}}
```

The JoU form of SPARQL queries generated from query rewrite often contains many UNION clauses and nested graph patterns that can become difficult for the query optimizer to optimize. To improve the quality of query plans generated by the

query optimizer, our latest inference engine rewrites the UNION clauses using FILTER clauses. For example, the query above is rewritten as follows, using `sT(...)` as notational shorthand for the SPARQL operator `sameTerm(...)`. Recall that `sameTerm(A, B)` is true if `A` and `B` are the same RDF term.

```
select ?x ?y ?n where
 {{?x ?p1 ?y FILTER(sT(?p1, :friendOf) || sT(?p1, :spouseOf ))}
  {?y ?p2 ?n FILTER(sT(?p2, :name)    || sT(?p2, :nickName))}}.
```

As mentioned earlier, UNION clauses correspond to views for atoms in the query. Rules in the nr-datalog that define view for an atom of the query contain single atom in their body [11]. The former atom is entailed by the latter atoms. For example, the (SPARQL) CQ atom `{?x p ?y}` can be entailed from atoms of types `{?x q ?y}` (by sub-property relationships) and `{?y q ?x}` (by inverse relationships), and also of type `{?x rdf:type c}` (by existential class definition) if `?y` is a non-distinguished non-shared variable. There may be more than one atom for each type. In that case, the view corresponding to atom `{?x p ?y}` can be defined via filter clauses as follows. (Any of n2, n3 may be zero in which case corresponding pattern is omitted; $q1_1$ equals p.)

```
{{?x ?q ?y FILTER( sT(?q, q1₁) || ... sT(?q, q1ₙ₁))} UNION
  {?y ?q ?x FILTER( sT(?q, q2₁) || ... sT(?q, q2ₙ₂))} UNION
  {?x rdf:type ?c FILTER( sT(?c, c1₁) || ... sT(?c, c1ₙ₃))}}
```

Note that the unions above can be further collapsed using more general filter expressions and the result of query-rewrite is an expanded query (that is no rules are generated). A key benefit of treating UCQ as JoU is that the SQL translation of a JoU query is typically more amenable to RDBMS query optimizations because it uses fewer relational operators, which reduces the optimizer's combinatorial search space; and the JoU, together with the filter clause optimization, will typically execute more efficiently in an RDBMS because the optimizer can find a better plan involving fewer operators, which reduces runtime initialization and destruction costs. Take the Lehigh Benchmark (LUBM) Query 5 and 6 for example. The JoU approach takes both less time and fewer database I/O requests, as shown in the following table, to complete the query executions against the 1.1 billion-assertion LUBM 8000 data set. The machine used was a Sun M8000 server described in Section 5.

LUBM8000 (1.1B+ asserted facts)	JoU with FILTER Optimization		No Optimization	
	Time	# of DB I/O	Time	# of DB I/O
Q5 (719 matches)	98.9s	73K	171.1s	271K
Q6 (63M+ matches)	25.68s	48K	28.7s	73K

Table 1 Effectiveness of JoU with FILTER optimization

2.2   SPARQLing OWL 2 QL Aboxes

The main mechanism for computing query results in the current SPARQL standard is subgraph matching, that is, simple RDF entailment [12]. Additional RDF statements

can be inferred from the explicit RDF statements of an ontology using semantic interpretations of ontology languages such as OWL 2 QL. The next version of SPARQL (SPARQL 1.1) is in preparation and various entailment regimes have been specified that define basic graph pattern matching in terms of semantic entailment relations [13]. One such entailment regime is the OWL 2 Direct Semantics Entailment Regime (ER), which is relevant for querying OWL 2 QL Aboxes. The ER specifies how the entailment is used.

The entailed graphs may contain new blank nodes (that are not used in the explicit RDF statements). ER, however, restricts semantic entailments to just those graphs which contain no new blank nodes. In other words, all the variables of the query are treated as distinguished variables irrespective of whether they are projected. This limits the range of CQs that can be expressed using SPARQL 1.1 ER. For example, consider following two queries that differ only in projected variables.

```
select ?s ?x  { ?s  :friendOf ?x .}
select ?s     { ?s  :friendOf ?x .}
```

Per ER, both queries have empty results. However, if viewed as CQs, `?x` is a non-distinguished variable in the second query and `[?s` → `:John]` is a valid result. Therefore, the second CQ cannot be expressed in SPARQL 1.1 ER.

For practical reasons, we would like to be able to express all types of CQs to OWL-2 QL Aboxes using SPARQL, especially when there are well-defined algorithms such as `PerfectRef` for computing sound and complete answers for CQs. We thereby adopt, in addition to ER, another entailment regime for Abox queries to OWL 2 QL ontologies, namely *OWL 2 QL Entailment Regime (QLER)*. QLER is similar to ER except that non-projected variables can be mapped to new blank nodes (not specified in the explicit triples of the Abox or Tbox). Projected variables cannot be mapped to new blank nodes under both ER and QLER. QLER, unlike ER, is defined only for Abox queries, and property and class expressions are not allowed (that is, only concept and property IRIs may be used). Note that the results obtained under ER are always a subset of the results obtained under QLER.

Now, any CQ can be expressed as BGP SPARQL query under QLER (unlike ER). The BGP query can be expanded, as discussed in Section 2.1, such that the query results can be obtained from the expanded query by standard subgraph matching. SPARQL, however, supports more complex queries than BGPs and union of BGPs such as accessing graph names, filter clauses, and optional graph patterns. Thus, a query-rewrite technique for complex SPARQL queries is also required.

Under ER, since all variables are treated as distinguished variables, individual BGPs of a complex query can be expanded separately and replaced in place. For example, SPARQL query select ?s ?n {?s :friendOf ?x} OPTIONAL {?x :name ?n} can be expanded as,

```
select ?s ?n {{ {?s :friendOf ?x} UNION {?s :spouseOf ?x} }
   OPTIONAL {{ ?x :name ?n } UNION {?x :nickname ?n}}}.
```

This expansion strategy is, however, not valid under QLER. `?x` is a non-distinguished variable under QLER, and the expanded form by that strategy will be,

```
select ?s ?n {{{?s :friendOf ?x}
   UNION {?s :spouseOf ?x} UNION {?s rdf:type  :Married} }
   OPTIONAL { { ?x :name ?n } UNION {?x :nickname ?n}}}.
```

The query above produces two incorrect answers, [?s → :John; ?n → "Uli"] and [?s → :John; ?n → "Mary"], and the source of the incorrect answers is that binding [?s → :John] is obtained from the pattern {?s rdf:type :Married}, and then ?x, which is implicitly bound to some new blank node, is explicitly bound to nothing (that is ?x is null). A left outer join with bindings from {?x :name ?n } produces erroneous results because null value matches any value of ?x from the optional pattern.

The way around that problem is to bind ?x to a new blank node using SPARQL 1.1 assignment expression [14] BIND(BNODE(STR(?s)) AS ?x) as shown below. The BGP { ?s :friendOf ?x } is expanded into

```
{{ ?s :friendOf ?x }  UNION { ?s  :hasSpouce ?x } UNION
  { ?s rdf:type :Married . BIND(BNODE(STR(?s)) AS ?x). }}.
```

The bindings for a non-distinguished variable are also lost when two similar atoms of a CQ are replaced by their most general unifier; cf. reduction step of the PerfectRef algorithm. Let ?x be a non-distinguished variable that is unified with term t of other atom, which may be a variable or a constant, then the binding for ?x can be retained by using SPARQL 1.1 assignment expression BIND(t as ?x), in a manner similar to that used in the above example.

So, the query-rewrite technique for complex SPARQL queries under QLER consists of the following steps: 1) identify distinguished variables for all BGPs of the query, 2) expand BGPs separately using the standard query-rewrite techniques (for CQs), including the one described in Section 2.1, 3) make the bindings for non-distinguished variables explicit whenever they are not using SPARQL 1.1 assignment expressions as discussed above, and 4) replace the expanded BGPs in place. Steps 2) and 3), even though presented sequentially, are intended to be performed concurrently. That is the bindings may be made explicit in the expansion phase for BGPs.

3   Named-Graph based Local inference

Inference is typically performed against a complete ontology together with all the ontologies imported via owl:imports. In this case inference engines consolidate all of the information and then perform tasks like classification, consistency checking, and query answering, thereby maximizing the discovery of implicit relationships.

However, some applications need to restrict inference to a single ontology represented by a named graph. For example, a health care application may create a separate named graph for each patient in its system. In this case, inference is required to apply to *just* the assertions about each patient and a common schema ontology (TBox). This kind of inference is therefore local, as opposed to the traditional global inference, and it is called Named Graph based Local Inference (NGLI) by Oracle. NGLI together with the use of named graphs for asserted facts modularizes and improves the manageability of the data. For example, one patient's asserted and inferred information can be updated or removed without affecting those of other patients. In addition, a

modeling mistake in one patient's named graph will not be propagated throughout the rest of the dataset.

One naïve implementation is to run the regular, global, inference against each and every named graph separately. Such an approach is fine when the number of named graphs is small. The challenge is to efficiently deal with thousands, or tens of thousands of named graphs. In the existing forward-chaining based implementation, Oracle database uses SQL statements to implement the rule set defined in the OWL 2 RL specification. To add the local inference feature, we have considered two approaches. The first approach re-implements each rule by manually adding SQL constructs to limit joins to triples coming from the same named graphs. Take for example a length 2 property chain rule defined as follows:

```
    ?u1 :p1 ?u2, ?u2 :p2 ?u3      ➜    ?u1 :p ?u3
```

This rule can be implemented using the following SQL statement. Obviously this rule applies to all assertions in the given data set <IVIEW>, irrespective of the origins of the assertions involved.

```
   select distinct m1.sid, ID(p), m2.oid  from <IVIEW> m1, <IVIEW> m2
 where m1.pid=ID(p1) and m1.oid=m2.sid and m2.pid=ID(p2)
```

To extend the above SQL with local inference capability, the following additional SQL constructs (in *Italic* font) are added. The assumption here is that <IVIEW> has an additional column, `gid`, which stores the integer hash ID values of graph names. Also, as a convention, the common schema ontology is stored with a NULL gid value in the same <IVIEW>. This allows an easy separation of the common schema ontology axioms from those assertions made and stored in named graphs.

```
  select distinct m1.sid, ID(p), m2.oid,  nvl(m1.gid,m2.gid) AS gid
    from <IVIEW> m1, <IVIEW> m2
  where m1.pid=ID(p1) and m1.oid=m2.sid and m2.pid=ID(p2)
     and ( m1.gid = m2.gid or m1.gid is null or m2.gid is null)
```

In the above SQL statement, a new projection of gid column is added to tag each inferred triple with its origin. This is very useful provenance information. Also, a new Boolean expression is added to the end of the SQL statement. This new expression enforces that the two participating triples must come from the same named graph or one of them must come from the common schema ontology. Note that when dealing with more complex OWL 2 RL rules, the number of joins increases and this additional Boolean expression becomes more complicated. As a consequence, it is error prone to manually modify all existing SQL implementations to support the local inference. This motivated an annotation-based approach, where each existing SQL statement is annotated using SQL comments. Using the above example, the annotation (in Italic font) together with the original SQL statement looks like:

```
  select distinct m1.sid, ID(p), m2.oid    /* ANNOTATION: PROJECTION */
    from <IVIEW> m1, <IVIEW> m2  where m1.pid=ID(p1) and m1.oid=m2.sid
     and m2.pid=ID(p2)            /* ANNOTATION: ADDITIONAL_PREDICATE */
```

At runtime, the above dummy annotation texts will be replaced with proper SQL constructs, similar to those described before. Those automatically-generated SQL constructs are based on the number of joins in a rule implementation, and the set of

view aliases used in the SQL statement. Compared to the first approach, this annotation based approach is easier to implement and much more robust because all the actual SQL changes are centralized in a single function.

4   Extensible Inference

We realize in practice that, to be enterprise ready, an inference engine has to be extensible. Our engine natively supports RDFS, SKOS, OWLPrime [3], OWL 2 RL which is a rich subset of the OWL 2 semantics, and a core subset of OWL 2 EL that is sufficient to classify the well-known SNOMED ontology, in addition to user-defined positive Datalog-like forward-chaining rules to extend the semantics and reasoning capabilities beyond OWL. This is sufficient to satisfy the requirements of many real-world applications. However, some application domains need the full expressivity of OWL 2 DL. To satisfy these applications, we have further extended our inference engine by integrating it with third-party complete OWL 2 DL reasoners like PelletDB [6]. A key observation has been that even when dealing with a large-scale dataset which does not fit into main memory, the schema portion, or the TBox, tends to be small enough to fit into physical memory. So the idea is to extract the TBox from Oracle database via a set of Java APIs provided in the Jena Adapter [5], perform classification using the in memory DL reasoner and materialize the class and property hierarchies, save them back into Oracle, and finally invoke Oracle's native inference API to perform reasoning against the instance data, or the ABox.

This approach combines the full expressivity support provided by an external OWL 2 DL reasoner and the scalability of Oracle database. Such an approach is general enough and can be applied to other well-known OWL reasoners including Fact++, HermiT, and TrOWL. It is worth pointing out that such an extension to Oracle's inference capability is sound, but completeness in terms of query results cannot be guaranteed. Nonetheless, users welcome such an extension because 1) in-memory solutions simply cannot handle a very large dataset that exceeds the memory constraint, 2) more implicit relationships are made available using additional semantics provided by external reasoners.

5   Performance Evaluation

In this section, we evaluate the performance of Oracle's native inference engine. Most tests were performed on a SPARC Enterprise M8000 server with 16 SPARC 64 VII+ 3.0GHz CPUs providing a total of 64 cores and 128 parallel threads. There is 512 GB RAM and two 1-TB F5100 flash arrays incorporating 160 storage devices. Note that the performance evaluation is focused on local inference performance. A systematic evaluation of SPARQL query answering under QL semantics is ongoing.

Benchmark Data Generation. We are using the well-known, synthetic Lehigh University Benchmark (LUBM) to test the performance because 1) a LUBM dataset can be arbitrarily large; and 2) it is quite natural to extend a LUBM dataset from tri-

ples to quads. The existing LUBM data generator produces data in triple format. However, the triples are produced on a per university basis, so it is straightforward to append university information to the triples to yield quad data.

Local Inference Performance. In the following table, we compare the performance of named-graph based local inference against that of the regular, global inference. Three benchmark datasets are used and the dataset size is between 133 million and 3.45+ billion asserted facts. Such a scale is sufficient for many enterprise-class applications. The second and third columns list the number of inferred triples and elapsed time for global inference. The last two columns list the number of inferred new quads and elapsed time for local inference. Note that a parallel inference [7] with a degree of 128 was used for both the global and local inferences. In addition, at the end of both global and local inference process, a multi-column B-Tree index is built so that the inferred data is ready for query. The only factor that stopped us from testing even bigger ontologies was the 2-TB disk space constraint.

Benchmark/Inference Type	Global Inference		Local Inference	
	New triples	Elapsed time	New quads	Elapsed time
LUBM1000[1]	108M	12m 15s	111M	13m 0s
LUBM8000	869M	33m 17s	892M	40m 3s
LUBM25000	2.71B	1h 44m	2.78B	2h 1m

Table 2 Performance comparison between global and local inference

The performance of local inference is a bit slower than but still quite comparable to that of the global inference. The performance difference comes from two places: 1) local inference deals with quads instead of triples and a quad dataset is larger in size than its triple counterpart because of the additional graph names, 2) the SQL statement is more complex due to the additional expressions.

It may be counter intuitive that local inference produced more inferred relationships than global inference. An examination of the inference results suggests that there are inferred triples showing multiple times in different named graphs even though any named graph contains only a unique set of triples.

With help from a customer, we conducted a performance evaluation of local inference using OWL 2 RL profile against large-scale *real-world* data[2] from the medical/hospital domain. The machine used was a quarter-rack Exadata x2-2. It is a 2-node cluster and each node has 96 GB RAM and 24 CPU cores. Detailed hardware specifications can be found here[3]. It took around 100 minutes to complete the local inference using a parallel degree of 48. Inference generated a total of 574 million new quads.

Benchmark/Inference Type	Local Inference	
	New quads	Elapsed time
Real-world Medical/Hospital Dataset	574M	100m 28s

Table 3 Local inference performance against real-world quad dataset

---

[1] LUBM1000, LUBM8000, and LUBM25000 datasets have 133M+, 1.1B+, and 3.45B+ facts asserted, respectively.

[2] Private data. It has 1.163B+ quads asserted.

[3] http://www.oracle.com/technetwork/database/exadata/dbmachine-x2-2-datasheet-175280.pdf

Parallel Inference. Oracle's inference engine has benefited greatly from the parallel execution capabilities provided by the database. The same kind of parallel inference optimization, explained in [7], applies both to the regular, global inference and the local inference. Figure 1 shows the local inference performance improvement as the degree of parallelism goes higher. LUBM 25K benchmark was used for this experiment. Note that most improvement was achieved when the parallel degree went up from 24 to 64. After that, only marginal improvement was observed. This is due to the fact that the Sun M8000 has 64 cores.
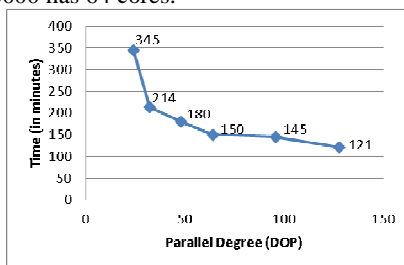

Figure 1 Local inference elapsed time versus degree of parallelism

6   Related Work

We will discuss works related to query rewriting as required for OWL 2 QL inference, implementations for OWL 2 QL reasoning. (We will be focusing on theory and techniques and not so much on relative performances).

Several techniques for query-rewriting have been developed since the `PerfectRef` algorithm was introduced in [10]; see [15] for a nice summary. The given CQ is reformulated as a UCQ by means of a backward-chaining resolution procedure in the `PerfectRef` algorithm. The size of the computed rewriting increases exponentially with respect to the number of atoms in the given query, in the worst case. But as observed by others, many of the new queries that were generated were superfluous, for example some of the CQs in a UCQ may be subsumed by others in the UCQ. An alternative resolution-based rewriting technique was proposed in [16] which avoids many useless unifications and thus UCQs are smaller even though they are still exponential in the number of atoms of the query. This alternative rewriting technique is implemented in the Requiem system[4]. Rosati et al. [11] argued that UCQs are reasons for exponential blow up, and have proposed a very sophisticated rewriting technique, the Presto algorithm, which produces a non-recursive Datalog program as a rewriting, instead of a UCQ. As noted before, we deploy the Presto algorithm for optimal performance.

The W3C's OWL implementations page[5] lists four systems that support OWL 2 QL reasoning: QuOnto[6], Owlgres[7], OWLIM[8], Quill[9].

QuOnto, Quill and Owlgres implement the `PerfectRef` query-rewrite technique, but Quonto implements an optimized `PerfectRef` query-rewrite technique, `QPerfRef` [10], and Quill in addition to query-rewrite, transforms ontology into a semantically approximate ontology [17].

Owlgres is an RDBMS-based implementation [18]. It deploys PerfectRef query-rewrite technique, with some optimizations such as Tbox terms with zero occurrences in Abox are identified in a preprocessing step and CQs of a UCQ that contain such Tbox terms are discarded, in contrast to the Presto algorithm. Furthermore, the UCQs are translated into a single SQL query that is a union of SQL queries, which is reminiscent of UoJ form. In contrast, we translate UCQs into more efficient JoU form, and the unions are collapsed into compact FILTER clauses.

OWLIM supports forward-chaining style rule-based reasoning, wherein blank nodes can be inferred during rule evaluation. OWL 2 QL reasoning is supported in OWLIM by defining new ruleset that captures OWL 2 QL semantics [19], and using the same forward chaining mechanism.

7   Conclusions

This paper described the recent advances in our OWL inference engine, which is implemented on top of the Oracle Database. We described optimizations for rewrite-based backward-chaining implementation of OWL 2 QL. We showed that conjunctive queries for OWL 2 QL knowledge bases cannot be expressed in SPARQL 1.1 using its entailment regimes because the regimes are very restrictive towards bindings to new blank nodes. We introduced a new regime to overcome that and described a query-rewrite technique for general SPARQL queries (which may contain constructs such as optional graph patterns). We introduced the concept of "named-graph based local inference" and described our implementation. We described the motivation for integrating a third-party OWL reasoner in our system, and described our implementation. Finally, we evaluated the performance of named-graph based local inference as compared to traditional global inference on synthetic data sets.

Reference

1. OWL 2 Web Ontology Language Direct Semantics. http://www.w3.org/TR/owl2-direct-semantics/
2. Oracle Database Semantic Technologies.
   http://www.oracle.com/technetwork/database/options/semantic-tech/index.html

[7] http://pellet.owldl.com/owlgres

[8] http://www.ontotext.com/owlim/

[9] http://kt.abdn.ac.uk/wiki/Projects/Quill

3. OWL 2 Web Ontology Language Profiles. http://www.w3.org/TR/owl2-profiles/

4. Wu, Z., Eadon, G., Das, S., Chong, E.I., Kolovski, V., Annamalai, M., Srinivasan, J.: "Implementing and Inference Engine for RDFS/OWL Constructs and User-Defined Rules in Oracle" IEEE 24th Intl. Conf. On Data Engineering (ICDE) 2008

5. Oracle Database Semantic Technologies Developer's Guide 11g Release 2 (11.2) http://docs.oracle.com/cd/E11882_01/appdev.112/e11828/toc.htm

6. Introducing PelletDb: Expressive, Scalable Semantic Reasoning for the Enterprise http://clarkparsia.com/files/pdf/pelletdb-whitepaper.pdf

7. Kolovski, V., Wu, Z., Eadon, G.: Optimizing Enterprise-Scale OWL 2 RL Reasoning in a Relational Database System. International Semantic Web Conference (1) 2010: 436-452

8. J. Broekstra, F. van Harmelen, and A. Kampman, "Seasme: A Generic Architecture for Storing and Querying RDF and RDF Schema". International Semantic Web Conference (ISWC) 2002.

9. L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu, "RStar: An RDF Storage and Querying System for Enterprise Resource Management". CIKM 2004.

10. Calvanese, G. deD., Giacomo, D.G., Lembo, M.D., Lenzerini, R.M., Rosati ", R.: Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family" In J.. Journal of Automated Reasoning 39(3):) (October 2007) 385---429, 2007.

11. Rosati, R., Almatelli, A.: Improving Query Answering over DL-Lite Ontologies. In Proceedings of the 12th International Conference on Principles of Knowledge Representation and Reasoning. KR, AAAI Press (2010).

12. SPARQL Query Language for RDF. W3C Recommendation 15 January 2008. http://www.w3.org/TR/rdf-sparql-query/ Last accessed 18-April-2012.

13. SPARQL 1.1 Entailment Regimes. W3C Working Draft 05 January 2012. http://www.w3.org/TR/sparql11-entailment/ Last accessed 18-April-2012.

14. SPARQL 1.1 Query Language. W3C Working Draft 05 January 2012. http://www.w3.org/TR/sparql11-query/ Last accessed 18-April-2012.

15. Gottlob, G., Schwentick, T.: Rewriting Ontological Queries into Small Nonrecursive Datalog Programs. In Proceedings of the 24th International Workshop on Description Logics (DL 2011), Barcelona, Spain, July 13-16, 2011.

16. Pe'rez-Urbina, H., Motik, B., Horrocks, I.: Tractable Query Answering and Rewriting under Description Logic Constraints. Journal of Applied Logic 8(2) (2010) 186—209.

17. Pan, J.Z., Thomas, E.: Approximating OWL-DL Ontologies. In: Proceedings of the 22nd National Conference on Artificial Intelligence - Volume 2. AAAI'07, AAAI Press (2007) 1434—1439.

18. Stocker, M., Smith, M.: Owlgres: A Scalable OWL Reasoner. In Proceedings of the Fifth OWLED Workshop on OWL: Experiences and Directions, Karlsruhe, Germany, October 26-27, 2008.

19. Bishop, B., Bojanov, S.: Implementing OWL 2 RL and OWL 2 QL. In Proceedings of the 8th International Workshop on OWL: Experiences and Directions (OWLED 2011), San Francisco, California, USA, June 5-6, 2011.

20. Narayanan, S., Catalyurek, U., Kurc, T., Saltz, J.: Parallel Materialization of Large ABoxes. In: Proceedings of the 2009 ACM symposium on Applied Computing. SAC'09, New York, NY, USA, ACM (2009) 1257—1261.

21. Urbani, J., Kotoulas, S., Massen, J., van Harmelen, F., Bal, H.: Webpie: A web-scale parallel inference engine using mapreduce. Web Semantics: Science, Services and Agents on the World Wide Web 10 (2012).

22. Hogan, A., Pan, J., Polleres, A., Decker, S.: SAOR: Template Rule Optimisations for Distributed Reasoning over 1 Billion Linked Data Triples. In: 9th International Semantic Web Conference (ISWC). (November 2010).