Type-based Semantic Optimization for Scalable RDF Graph Pattern Matching

HyeongSik Kim
North Carolina State University, Raleigh, USA
hkim22@ncsu.edu

Padmashree Ravindra*
Microsoft Corporation, Redmond, USA
paravin@microsoft.com

Kemafor Anyanwu
North Carolina State University, Raleigh, USA
kogan@ncsu.edu

ABSTRACT

Scalable query processing relies on early and aggressive determination and pruning of query-irrelevant data. Besides the traditional space-pruning techniques such as indexing, type-based optimizations that exploit integrity constraints defined on the types can be used to rewrite queries into more efficient ones. However, such optimizations are only applicable in strongly-typed data and query models which make it a challenge for semi-structured models such as RDF. Consequently, developing techniques for enabling type-based query optimizations will contribute new insight to improving the scalability of RDF processing systems.

In this paper, we address the challenge of type-based query optimization for RDF graph pattern queries. The approach comprises of (i) a novel type system for RDF data induced from data and ontologies and (ii) a query optimization and evaluation framework for evaluating graph pattern queries using type-based optimizations. An implementation of this approach integrated into Apache Pig is presented and evaluated. Comprehensive experiments conducted on real-world and synthetic benchmark datasets show that our approach is up to 500X faster than existing approaches.

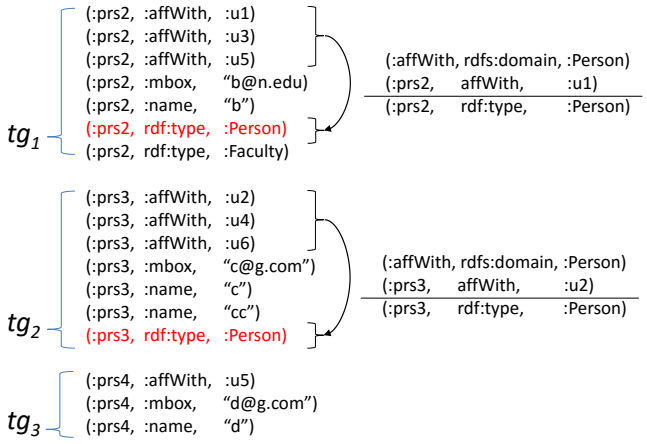

Figure 1: A set of example RDF triples describing an academic domain. Black curved arrows denote that target triples can be inferred from source triples using entailment rules shown over the arrows.

1. INTRODUCTION

A key factor in achieving scalable query processing is the ability to prune out query-irrelevant data as early in the processing pipeline as possible. Traditionally, optimizers do this by using statistical data distributions to estimate the most selective processing path for a query as well as indexed-based algorithms for limiting the search space of individual query operators.

RDF data management systems have towed similar lines in query optimization adopting multi-indexing [20, 22], cost-based optimization [20, 22], and sideways information passing [20]. However, large-scale processing of RDF remains a challenge because its graph-structured nature demands significantly more processing than relational models and, many of these optimization techniques transfer only in a limited way to distributed contexts, thereby limiting their impact.

Another class of optimization techniques that is yet to be explored is called *Semantic Query Optimization - SQO*. SQO uses domain knowledge captured as semantic integrity constraints to rewrite queries into more efficient ones. The rewritten queries may introduce new predicates to make a query more selective and offer opportunities for alternate access paths or may allow expensive predicates such as joins to be removed completely from a query expression. The latter is of particular interest in the context of RDF query processing due to its join-heavy nature.

To illustrate this with relational database example, consider two relations `EMP` and `EMPBEN` which keep information about employees (part time and full time) and their benefits (policies and dependent covered) respectively. In order to find employees in CA that have life insurance, we may write the following SQL query:

```
SELECT * FROM EMP E, EMPBEN EB WHERE
E.city = 'CA' AND E.eid = EB.eid AND
EB.ptype = 'life insurance';
```

However, assuming that it is known that *only full-time employees have life insurance and all employees have some*

*This research was done while the second author was a student at the Department of Computer Science, North Carolina State University.

WWW 2017, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/3038912.3052655

kind of life insurance, then we can exploit this information to reformulate the query into:

```
SELECT * FROM EMP E WHERE
E.city = 'CA' AND E.status = 'FT';
```

As can be seen, the rewritten query will produce an equivalent result but using a different but more efficient query that avoids a join operation. Such semantic query optimizations require a fairly-structured data model consisting of delineated relation types on which semantic integrity constraints can be defined. Unfortunately, the weak typing model and semi-structured nature of RDF makes applying similar techniques a challenge.

However, it may be possible to induce a stronger type system over RDF in a manner similar in spirit to the idea of characteristics sets [19]. Further, rules that apply to specific types can be derived from axioms in associated ontologies. Such rules can then be used as integrity constraints as the basis for developing type-based semantic query optimization techniques. As an example of the potential for this, suppose that we view RDF data in terms of the groups of similar structures. While there are many possible structure granularities that could be chosen, let us consider a very natural and fundamental one - resources with similar property types. Essentially, given that a resource's description is fundamentally the set Properties it has, we can consider all similarly described resources as constituting a type.

For example, Fig. 1 shows collection of triples describing three resources :prs2, :prs3 and :prs4. These triples consist of the same set of Properties :affWith, :mbox, :name, excluding rdf:type. Hereafter, we denote a type for these triplegroups as τ_{amn} in shorthand. Let us assume that we have an ontology with an axiom about the domain of Property affwith. rdfs:domain in RDFS [2] defines the domain of a Property as the set of classes that it can be applied to, e.g., the ontological axiom (:affWith, rdfs:domain, :Person) implies that resources with Property :affWith can be inferred to be of class :Person. With this axiom, we can induce a rule that derives a triple with Property rdf:type such as

$$\mathtt{x} : \tau_{amn} \rightarrow (\mathtt{x_S}, \mathtt{rdf{:}type}, \mathtt{:Person})$$

where $\mathtt{x} : \tau_{amn}$ denotes a collection of triples typed as τ_{amn}, such as tg_1. $\mathtt{x_S}$ is the Subject of triples such as :prs2.

To exploit such a rule for optimization, we consider a scenario where we have a graph pattern with a triple pattern whose Property is rdf:type and Object is :Person. We could rewrite the query without such a triple pattern, essentially eliminating a join operation. On the data side, we could also avoid explicitly representing such inferable rdf:type triples in the data model. Rather, we could capture this as a meta-rule that applies to the set of similar resources that have this characteristic. However, the issue of optimizing physical storage models based on dematerializing such triples and a representation scheme for them is an important issue that impacts query processing but is outside the scope of this paper. That issue has been addressed in a companion paper which can be found at our project website [1].

It is important to differentiate between the semantic query optimization being proposed here and semantic query answering or query answering with entailment. In the latter, facts that are implicit are made explicit. On the other hand, semantic optimization makes explicitly asserted triples implicit, representing them as meta-rules.

Challenges. The above discussion hints at the possibilities of strong typing that builds on the equivalence relation, which may lead to semantic type-based optimizations for scalable query processing. However, several issues may arise when considering how to practicalize this new type system for RDF.

1. First, many types can be derived from data. Emergent relational schema [24] limited a range of types to relieve this issue, but we need a more holistic approach that can induce and manage all existing types automatically and efficiently.
2. Second, we need to consider corner cases that arise from type-based semantic optimizations. For example, we need to differentiate types that include derivable triples from types that don't.
3. Finally, queries need to be rewritten into our type model and this rewriting process should be automatic because it may be impossible for users to be aware of all existing types.

This paper makes the following novel contributions:

1. *A Typing Model for RDF Data* called *R-Types* that is based on an aggregate data model of resource descriptions and uses ontological axioms to derive type-based meta-rules akin to integrity constraints.
2. *A translation of SPARQL graph pattern queries* into expressions over R-Types. Rewriting SPARQL graph pattern queries in terms of R-Types that have associated integrity constraints or meta-rules can be further optimized in the spirit of semantic query optimization by eliminating redundant expressions given the information implied by integrity constraint.
3. A comprehensive evaluation conducted on real-world and synthetic datasets (DBPSB and LUBM). The evaluation results show that our approach was up to $500X$ faster than existing ones.

The rest of the paper is organized as follows: Section 2 introduces the R-Type model and semantic optimizations based on R-Types. Section 3 presents a query processing model over R-Types and Section 4 overviews implementation. Section 5 discusses related work and Section 6 shows comparative evaluation between the proposed and other approaches. Section 7 concludes the paper.

2. A TYPED MODEL FOR RDF

As alluded to in the introduction, we would like to develop types on holistic descriptions of resources, i.e., the sets of Properties that describe resources, rather than in terms of individual triples. A good foundation for this idea is in [25, 15, 26] where an aggregate data model and corresponding algebra called the (*Nested Triplegroup Algebra - NTGA*) is presented. In NTGA, data is modeled using triplegroups (groups of triples with same Subject resource) as first-class citizens. NTGA also proposes a query algebra which has operators for manipulating triplegroups TG_Join ($\bowtie^{\gamma}$) for 'joining' triplegroups. However, the NTGA data model is not typed and consequently would need to be extended with types in order to advance towards the goal of typed-based semantic query optimization.

2.1 R-Type: A Typing Model for RDF

We begin by defining a universe PN of type names as a function of a set P of domain-specific Property names, $PN \subset 2^P$. Note that P does not include all the pre-defined Properties of RDF and RDFS such as `rdf:type` and `rdfs:domain`. We then define a set of R-Types $\Gamma = \{\tau_{\alpha_1}, \tau_{\alpha_2}, ..., \tau_{\alpha_i}\}$ where $\alpha_i \in PN$. Essentially, an R-Type captures a unique Property combination (denoted as its subscript). For example, given $P = \{\mathtt{a}, \mathtt{h}, \mathtt{l}, \mathtt{m}, \mathtt{n}, \mathtt{r}\}$, τ_a, τ_{ah}, ..., τ_{ahlmnr} are possible types. We also assume the existence of two convenience functions: $sig()$ returns the name of a type (i.e., an element of PN, while $sig^{-1}()$ takes a name and returns a type.

Based on this model, we can define a typing environment in which sets of triples or triplegroups are assigned types based on the description they confer on a resource. Given a set of triples whose Subjects are s_1 and Property types are τ_{p_1}, ..., τ_{p_i}, the corresponding triplegroup that comprises the same set of triples is assigned the R-Type $\tau_{p_1 p_2 ... p_i}$. More formally, let Γ be a typing environment that assigns a triple $\langle s_1, p_1, o_1 \rangle$ based on its Property type $\langle s_1, p_1, o_1 \rangle : \tau_{p_1}$. We then define a new typing environment Γ' for R-Types that types a triplegroup rooted at s_1 as follows:

Rule 1.

$$\frac{\Gamma \vdash \langle s_1, p_1, o_1 \rangle : \tau_{p_1}, \langle s_1, p_2, o_2 \rangle : \tau_{p_2}, ..., \langle s_1, p_i, o_i \rangle : \tau_{p_i}}{\Gamma' \vdash \left\{ \begin{matrix} \langle s_1, p_1, o_1 \rangle \\ \langle s_1, p_2, o_2 \rangle \\ ... \\ \langle s_1, p_i, o_i \rangle \end{matrix} \right\} : \tau_{p_1 p_2 ... p_i}}$$

It is straightforward to verify that such a typing model defines an equivalence relation on the set of triplegroups, i.e., each triplegroup (and therefore each triple) is assigned exactly one type.

Example 2.1. (R-Typed Triplegroups) The following types can be induced from the triplegroup model in Fig. 1 using *Rule 1*: $\{\tau_a, ..., \tau_{am}, ..., \tau_{amn}\}$. Types can be assigned to triplegroups as follows: $\mathtt{tg}_1$, $\mathtt{tg}_2$, $\mathtt{tg}_3$:τ_{amn}.

An insertion of a triple into a model may lead to a type reassignment for a triplegroup. For example, assume that a new triple $t = (s, p, o)$ needs to be added to an R-Typed model that contains a triplegroup `tg` with Subject s as a member of the type τ_α. The state of the triplegroup tg will be updated to $tg' = t \cup tg$, and its type reassigned to $\tau_{\alpha \cup p}$. The type reassignment in the case of a deletion can be defined analogously.

2.2 Extending R-Type Model with Ontological Axioms

In the introduction section, the motivating example suggested that it might be possible to derive rules similar to integrity constraints for resources with similar descriptions (i.e., having the same set of Property types) if ontological axioms associated with their Properties are considered. In particular, we may be able to determine that some explicitly stated `rdf:type` assertions included as part of their descriptions, are *inferable*. In other words, certain `rdf:type` assertions may be derived by considering some of the other Property types in their descriptions and ontological axioms associated with those Properties. Further, the issue of derivability applies to all resources that have similar descriptions, e.g., :`prs2` and :`prs3`. Consequently, we can capture this derivability as a meta-rule, much like an integrity constraint, that applies to all similar resources. In fact, there would be no need to maintain explicit assertions of such type statements for all such resources.

Our R-Type model provides the basis for reasoning about similar resources based on equivalence of their Property descriptions, since members of an R-Type all have the same distinct set of Property types. Capturing the 'integrity constraint' in the type model means addressing the role of the `rdf:type` Property type in our universe of types (thus far, `rdf:type` was not included as one of the Property types over which R-Types are defined). Essentially, our universe of types will need to be extended to include types whose names or signatures indicate the presence of these inferable `rdf:type` Properties. However, consideration of the `rdf:type` Property as part of the type namespace must address the following nuances:

1. Simply removing inferable triples could introduce ambiguities in determining matches since it may become difficult to distinguish between triplegroups with inferable triples and similar triplegroups that did not originally contain such triples. Suppose that our example model also had a triplegroup $\mathtt{tg}'_2$ that does not contain an `rdf:type` triple with the Object :`Person`. Superficially, $\mathtt{tg}'_2$ would be considered to be of the same type as $\mathtt{tg}_2$, when in fact they should not be.
2. Not all `rdf:type` assertions in a triplegroup may be inferable. Let us consider the case that any axioms for the class :`Faculty` do not exist in ontologies, which means that `rdf:type` triples with the Object :`Faculty` are not inferable. In other words, this leads to the case that the non-inferable triple exists in $\mathtt{tg}_1$. Therefore, our typing model should capture the distinction between inferable and non-inferable triples.

To address these nuances, we extend the R-Type universe to include types with names. Specifically, we refine the `rdf:type` Property by 'promoting' the associated classes (or Objects) to special Properties, e.g., the pair (`rdf:type`, :`Person`) is promoted to `rdf:type`$_{\text{:Person}}$ (or $\mathtt{t}_{\text{Person}}$ for brevity). Therefore, R-Types that contain such Properties integrate the promoted Properties into their type signatures, e.g., $\tau_{amnt_{Person}}$ for $\mathtt{tg}_2$. This refinement of R-Types then allows differentiation of types with inferable triples vs. similar types that never had such `rdf:type` triples in the first place.

We can now formalize the semantic optimization that identifies 'inferable' triples based on the above type refinement.

Definition 2.1. (D-inferable `rdf:type` Triples) Let H be a set of schema triples (triples whose Properties are defined in the namespace of RDFS), $type_c$ or t_c be a *class-type* triple (a triple of the form (s, `rdf:type`, c)), tg be a triplegroup containing triples t_1, t_2, t_3, ..., t_i, t_c that have Properties p_1, p_2, p_3, ..., p_i, *rdf:type*, respectively. t_c is then *Domain-inferable* or *D-inferable* if $\exists\,(p_j$, *rdfs:domain*, $c) \in H$ $(1 \leq j \leq i)$. Otherwise, t_c is *non-inferable*.

Based on the definition, we introduce a new type assignment rule that i) implicitly represents D-inferable triples in triplegroups and ii) modifies an R-Type by promoting D-inferable triples in triplegroups to special Properties. Let `type` and `domain` be abbreviations of `rdf:type` and `rdfs:domain` for brevity. Let also `tg` be a triplegroup that consists of i)

triples with Properties p_1, ..., p_k and ii) D-inferable class-type triples whose Objects are c_1, ..., c_l (their R-Types are $\tau_{p_1 \ldots p_k}$). Given tg and the entailment rule rdfs2 [4], the R-Type of tg can be reassigned as follows.

Rule 2.

$$\frac{\Gamma' \vdash \left\{\begin{array}{l}\langle s_1,\ p_1,\ o_1\rangle \\ \ldots \\ \langle s_1,\ p_k,\ o_k\rangle \\ \langle s_1, \texttt{type}, c_1\rangle \\ \ldots \\ \langle s_1, \texttt{type}, c_l\rangle\end{array}\right\} : \tau_{p_1 \ldots p_k}, \quad \frac{\langle p_i,\ \texttt{domain},\ c_j\rangle \quad \langle s_1,\ p_i,\ o_i\rangle}{\langle s_1,\ \texttt{type},\ c_j\rangle} \quad \begin{array}{l}\forall i \in [1 \ldots k]\ \&\\ j \in [1 \ldots l]\end{array}}{\Gamma' \vdash \left\{\begin{array}{l}\langle s_1,\ p_1,\ o_1\rangle \\ \ldots \\ \langle s_1,\ p_k,\ o_k\rangle \\ \cancel{\langle s_1, \texttt{type}, \&c_1\rangle} \\ \ldots \\ \cancel{\langle s_1, \texttt{type}, \&c_l\rangle}\end{array}\right\} : \tau_{p_1 \ldots p_k \& type_{c_1} \ldots \& type_{c_l}}}$$

Strikethrough lines over triples denote that they are implicitly represented and promoted as special Properties (denoted by &). Intuitively, the promotion of the 'objectfied' rdf:type statements into the type name is the R-Type model method of capturing integrity constraints over types.

Example 2.2. (Type Refinement) The class-type triple (:prs2, rdf:type, :Person) in a $\texttt{tg}_1$ is now implicitly represented and promoted as a special Property, t_{Person}. The type of $\texttt{tg}_1$ is then refined from τ_{amn} to $\tau_{amnt_{Faculty}\&t_{Person}}$ using Rule 2.

The next issue is how to handle R-Types with non-inferable triples. Similar to the D-inferable case, we promote pairs of Properties and Objects in non-inferable triples as special Properties. However, it is possible to cause over-discrimination even among 'equivalent' triplegroups with respect to certain queries. In other words, we may end up with too many types that consist of the same Properties with different rdf:types, e.g., $\tau_{amnt_{Faculty}\&t_{Person}}$, $\tau_{amnt_{\&}t_{Person}}$, and $\tau_{amnt_{Faculty}}$, and so on. To relieve the over-discrimination issue, we further partition a set of Properties that consist of types into two subsets: special Properties and other remaining ones. Therefore, triplegroups having such types are also horizontally partitioned into i) non-inferable triples corresponding to special Properties and ii) triplegroups that consist of remaining triples.

Example 2.3. (Partitioning Non-inferable Triples) The triplegroup $\texttt{tg}_1$ in Fig. 1 is partitioned into two group of triples and placed under different types: $\tau_{amn\&t_{Person}}$ and $\tau_{t_{Faculty}}$ where the latter represents the type for non-inferable triples whose Objects are Faculty.

3. R-TYPE QUERY PROCESSING MODEL

3.1 Rewriting Graph Patterns into R-Type Expressions

To support query execution, SPARQL queries need to be compiled into query expressions over R-Typed models. We begin by introducing some notations and convenience functions for the formalization of the query processing model. $props()$ is a function such that $props(tg)$ for a triplegroup tg, returns the distinct set of Property types in tg, e.g., $props(\texttt{tg}_2) = \{a, m, n, t\}$. [[.]] is an interpretation function

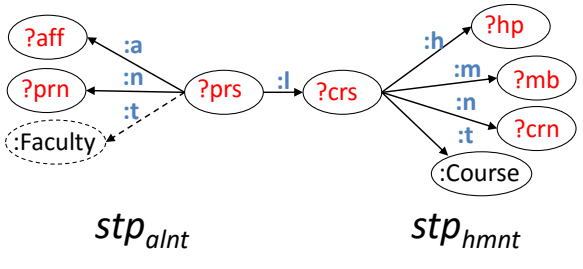

Figure 2: A graph representation of graph pattern query Q that retrieves faculties who lecture courses.

such that $[[\tau_i]]$ returns all member triplegroups for τ_i, i.e., triplegroups tg such that $props(tg) = sig(\tau_i)$. For example, $[[\tau_{amn}]] = \{\texttt{tg}_3\}$ for triplegroups in Fig. 1.

To formalize the semantics of graph pattern queries under an R-Type model, our model views graph patterns as composed of star patterns (sets of triple patterns that share common Subject variables), linked by combination operations such as JOIN, UNION, OPTIONAL, etc. Given a graph pattern GP that comprises the set of star patterns $stps = \{stp_1, stp_2, \ldots, stp_n\}$, GP can be expressed as a 2-tuple

$$(\{stp_1, stp_2, \ldots, stp_m\}, \{op_1, op_2, \ldots, op_n\}), n \geq m$$

where op_k $(k \in 1 \ldots n)$ is an NTGA binary operator such as $\bowtie^{\gamma}$ with a pair stp_i, stp_j as operands. For example, a query Q in Fig. 2 can be decomposed into stp_{alnt} and stp_{hmnt} connected via a common variable $?crs$. Consequently, given an R-Type model Γ that consists of a set of types $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_i\}$, we define the semantics of graph pattern queries under Γ inductively, beginning with the semantics of star graph patterns.

Definition 3.1. (Star Pattern Interpretation Under R-Typing) Given a star pattern stp, the semantics of stp or $[[stp]]$, is defined in terms of two query operators:

i) TG_Projection (π^{γ}): Given Γ and a set of Properties p_1, p_2, ..., $p_k \in P$, $\pi^{\gamma}_{p_1,p_2,\ldots,p_k}(\Gamma)$ returns the *projection* of $\{p_1, p_2, ..., p_k\}$ on Γ, defined as a set of triplegroups:

$$\{tg' \mid tg' \subseteq tg, tg \in [[\tau_i \in \Gamma]], props(tg') = \{p_1, p_2, ..., p_k\}\}$$

TG_Projection eliminates triples with Property types that are irrelevant to a star graph pattern. As an example, $\pi^{\gamma}_{prop(\texttt{stp}_{\texttt{mnt}})}(\Gamma)$ returns $\{\texttt{tg}'_1, \texttt{tg}'_2, \texttt{tg}'_3\}$ which correspond to $\texttt{tg}_1, \texttt{tg}_2, \texttt{tg}_3$ without irrelevant triples such as $(:prs2, :a, :u1)$, $(:prs3, :a, :u2)$, $(:prs4, :a, :u5)$, assuming that :a denotes :affWith .

ii) TG_TypeSelection (σ^{γ}): Given a universe of type names PN and a set of Properties $p_1, p_2, ..., p_k \in P$, $\sigma^{\gamma}_{p_1,p_2,\ldots,p_k}(PN)$ returns a set of names for types where

$$\{\tau_1, \tau_2, ..., \tau_i\},\ i \supseteq \{p_1, p_2, ..., p_k\}$$

TG_TypeSelection essentially computes a set of names for types that *match* a star pattern stp, i.e., a set of type names that contain the star pattern's Properties. For example, $\sigma^{\gamma}_{prop(\texttt{stp}_{\texttt{amnt}})}(PN)$ returns a set of names for types such as $\{\tau_{amn}, \tau_{amnr}, ...\}$.

$[[stp]]$ can be expressed in terms of the above operators as follows.

$$[[stp]] = \pi^{\gamma}_{prop(stp)}(sig^{-1}(\sigma^{\gamma}_{prop(stp)}(PN))) \qquad (1)$$

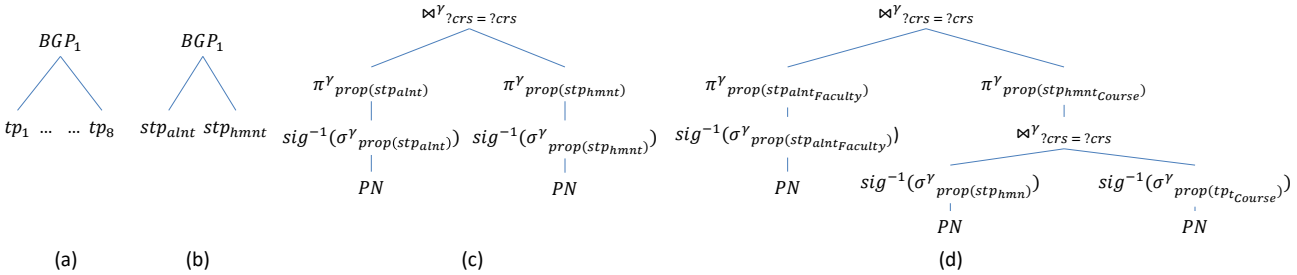

Figure 3: An illustration of query transformation process: (a) SPARQL S-Expression of Q, (b) reformulated S-Expression, (c) compiled R-Type expression, and (d) reformulated R-Type expression using semantic optimizations.

In other words, we first apply a meta-query that returns names of matching types, so that we can retrieve types based on the names using $sig^{-1}()$. We then project relevant subgroups from type-matching triplegroups.

A query model over R-Types is necessary for supporting graph pattern matching queries. Specifically, an interpretation of graph pattern queries over R-Types is needed to form the basis of query writing rules from typical SPARQL algebraic expression to an equivalent expression over R-Types. Fig. 3 shows how a graph pattern can be interpreted and transformed into a logical query plan that uses R-Type operators.

1. Our example query is a graph pattern Q that consists of two star patterns $\texttt{stp}_{\texttt{alnt}}$ and $\texttt{stp}_{\texttt{hmnt}}$ connected by a common variable ?crs, as shown in Fig. 2. We assume that Q does not include any modifiers but only consists of a set of triple patterns. A SPARQL parser such as Jena [17] parses a graph pattern and transforms the pattern into corresponding SPARQL Syntax-Expression (S-Expression), as shown in Fig. 3(a). The produced S-Expression consists of one BGP node (BGP), which is connected to a set of child nodes that denote triple patterns ($\texttt{tp}_1$, ..., $\texttt{tp}_8$).
2. A set of child nodes in S-Expression is then transformed into the ones representing star patterns. Specifically, a group-by operation is applied over these triple patterns, which produces sets of nodes representing triple patterns grouped by common Subject. These nodes are therefore replaced into a *star pattern node*, which is a special node representing a star pattern. Fig. 3(b) shows the result of this reformulation, i.e., one BGP node (BGP) with two star pattern nodes ($\texttt{stp}_{\texttt{alnt}}$ and and $\texttt{stp}_{\texttt{hmnt}}$). This allows us to extract sets of Properties representing star patterns from the tree, so that the sets can be used as parameters of R-Type operators.
3. The next step is to construct R-Type expressions based on the rewritten S-Expression above. For each star pattern node representing a star pattern *stp*, we assign a combination of our operators, i.e., π^{γ} and σ^{γ}. If there are join variables among star patterns, a join operator such as $\bowtie^{\gamma}$ is added, so that stars can be connected. This transformation leads to produce a R-Type expression tree as shown in Fig. 3(c).

3.2 Semantic Query Optimization of R-Type Expressions

The semantic optimization discussed in the previous section leads to the reassignment of R-Types for triplegroups that include class-type triples, therefore we also need to consider how the query processing model can cope with reassigned R-Types. There are two scenarios we need to discuss based on the inferability of class-type triple patterns.

First, star patterns in a query contain D-inferable triple patterns. In this case, because semantic optimizations discussed earlier leads to representing D-inferable triples in an implicit way, we do not need to process them anymore. In the viewpoint of other approaches, this may be considered as 'Join Elimination' among existing semantic optimization techniques [23] because triples matching to the D-inferable triple patterns do not need to be joined with other ones. However, in our typing model, such inferable triples are already materialized with other ones as triplegroups and later the inferable triples are represented in implicit way. Therefore, our technique essentially corresponds to 'Restriction Elimination', which eliminates redundant predicates to simplify queries using integrity constraints. Second, star patterns in a query contain non-inferable class-type triple patterns. In such a case, we need to introduce join operations to reassemble triplegroups and matching class-type triples. Finally, star patterns in a query do not contain any class-type triple patterns, which does not lead to any changes in query interpretation.

Example 3.1. (Reformulation of R-Type Expression) Fig. 3(d) shows how the R-Type expression can be reformulated based on the semantic optimizations. We assume that the ontological axiom (:lecture, rdfs:domain, :Faculty) exists in ontologies. The triple pattern $\texttt{t}_{\texttt{Faculty}}$ of the star pattern $\texttt{stp}_{\texttt{alnt}}$ in Fig. 2 then becomes D-inferable triple pattern because $\texttt{stp}_{\texttt{alnt}}$ contains a triple pattern with Property :lecture. Therefore, the parameters of operators in the left subtree are changed from $prop(\texttt{stp}_{\texttt{alnt}})$ to $prop(\texttt{stp}_{\texttt{aln\&t}_{\texttt{Faculty}}})$. Second, the star pattern $\texttt{stp}_{\texttt{hmnt}}$ in Fig. 2 contains non-inferable triple pattern, i.e., no axioms for $\texttt{t}_{\texttt{Course}}$. In such a case, σ^{γ} in the right subtree is reformulated into the subtree that consists of a join operator with two child σ^{γ}s. This allows us to compute names of the types matching $\texttt{stp}_{\texttt{hmn}}$ and $\texttt{t}_{\texttt{Course}}$ respectively and then join triplegroups having such types, which leads to the reconstruction of the triplegroups matching $\texttt{stp}_{\texttt{hmnt}}$.

Finally, it is possible that some queries could contain class-type triples with unbound Objects, which may need additional considerations. We do not discuss such cases in this paper due to space constraints, but details are available at our project website [1].

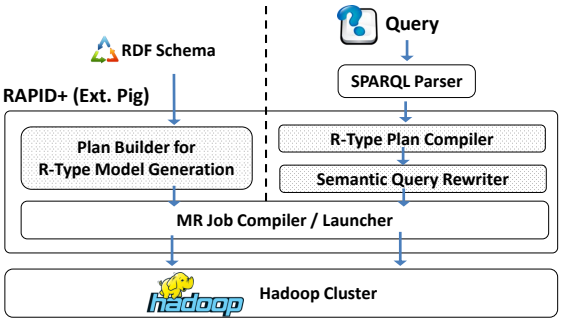

Figure 4: An overall architecture of RAPID+ with R-Type Model.

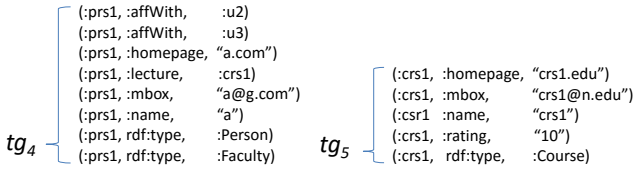

Figure 5: Additional triplegroups describing faculties and courses

4. IMPLEMENTATION OF R-TYPE MODEL

To process graph pattern queries using an R-Type model on Hadoop, we implement a new query processing layer that transforms queries into R-Type expressions on the system called RAPID+ [25]. It is an extension of Pig[1] that includes optimizations for RDF data processing. Fig. 4 shows that the query processing layer of Pig now includes two groups of new components (shaded ones).

The components of the left group are responsible of the model generation, i.e., *Plan Builder for R-Type Model Generation* accepts ontological axioms from RDF schema and builds a query plan that generates triplegroups and assigns R-Types for them. This query plan is then compiled into a MapReduce job by Pig's job compiler and launcher, so that the job can be executed on a Hadoop cluster. The components of the right group are for processing queries using R-Type model. Queries are translated into S-Expressions using SPARQL parser, which are then compiled into R-Type logical plan using *R-Type Plan Compiler*. The plan is then reformulated using *Semantic Query Rewriter* and compiled into the physical plan, so that queries can be executed on Hadoop. For this purpose, we develop a physical operator called `TG_IndexScan` which maps to logical operators discussed earlier. This operator selectively loads query-relevant triples from type-matching triplegroups on HDFS. The plan is finally packaged as MapReduce jobs using Pig's job compiler.

Fig. 6 shows an overall flow of query planning and execution on MapReduce. We assume the existence of additional triplegroups, as shown in Fig. 5. In the stage of query planning, we rewrite star patterns based on the inferability of class-type triples and launch the first MR job that computes name of matching types using σ^{γ}. For example, for star patterns $\texttt{stp}_{\texttt{alnt}}$ and $\texttt{stp}_{\texttt{hmnt}}$ in Fig. 2, we will retrieve names of (i) $\tau_{ahlmn\&t_{Faculty}t_{Person}}$ and $\tau_{almn\&t_{Faculty}}$ for $\texttt{stp}_{\texttt{alnt}}$

[1] https://pig.apache.org

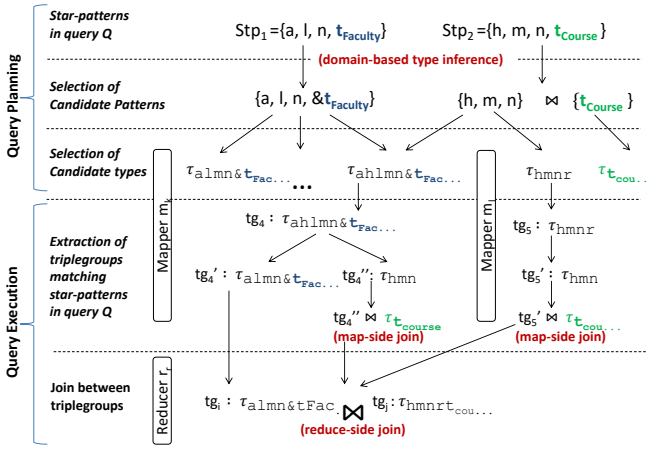

Figure 6: An overall processing flow of the query Q over the R-Type model with the semantic optimization

and (ii) τ_{hmnr} and $\tau_{t_{Course}}$ for $\texttt{stp}_{\texttt{hmnt}}$, respectively. In the query processing stage, τ_{hmnr} is joined with $\tau_{t_{Course}}$ using a map-side join and π^{γ} is applied over matching triplegroups to produce triplegroups that exactly match star patterns. Finally, triplegroups matching each star pattern are joined using $\bowtie^{\gamma}$.

We do not present all implementation details due to the lack of space, but interested readers can refer to the project website [1] for more information.

5. RELATED WORK

Mainstream approaches for RDF query optimization include logical data partitioning schemes such as Vertical Partitioning (VP) [5] which enable pruning of query-irrelevant triples if their properties are not present in a query. Other efforts such as multi-indexing or multiple sort orders [10, 20, 22] and co-partitioning of 'related' triples [10, 16, 12] have been used for join optimization by enabling indexed-joins or eliminating the need for re-partitioning in distributed settings. However, all these techniques manage data as triples which as fine-grained data units create the need for multiple join operations. The triple view also only enables very limited prunability of query-irrelevant triples due to the absence of a holistic picture of resource descriptions.

Alternative strategies have been investigated to tackle such issues, such as adopting a more holistic view of data by employing coarse-grained data units. Some approaches [25, 30, 8] are not only co-locating related triples but also managing them explicitly as pre-assembled logical units. Other approaches focus on compression techniques [9, 14, 21], which leverage graph redundancy to compress triples into subgraphs for compact representation and efficient access of data. All these efforts lead to eliminating the need for some join operations during query processing, but many of these efforts still require expensive scans over coarse-grained data units or compressed subgraphs, due to lack of discrimination among units.

As efforts to overcome the aforementioned limitations, some recent efforts employ structural summaries/signatures, similar to types. This enables indexing of exact [33] or similar [32] subgraphs, which achieves faster access to subgraphs matching subquery patterns. EAGRE [32] addition-

	Size (GB)	#Triples	#Properties	#Classes	#Types
LUBM 20k	450	815M	18	15	24
DBPSB 200	49	47M	39k	159k	42k

Table 1: Dataset characteristics

ally proposes a storage scheme for storing its subgraphs based on space-filling curve for optimizing queries with modifiers such as ORDER BY. Another relevant work is emergent schemas [24], which materializes frequently occurring subgraph patterns into a set of relations (Property Tables). However, emergent schemas are partial and do not completely cover the data graph. Further, users are required to express their queries in terms of these emerging schemas and therefore must be reasonably familiar with emergent schemas. This can be limiting because the number of potential patterns can be large for Big Data (e.g., $400K$ structures in DBPedia).

When ontologies are present, ontological axioms can be utilized for optimizations analogous to semantic query optimizations (SQO) in OODB [23] and XML [7]. In this context, some recent works [28, 29] study SQOs to minimize join expressions of inferable patterns using join eliminations. Our techniques go beyond SQOs by integrating typing knowledge and ontological constraints in type signatures, such as enabling efficient state-space restrictions as well as minimizing storage footprints.

6. EVALUATION

6.1 Testbed Setup

The goal of our evaluation was to compare the performance of our approach with other hadoop-based ones.

Benchmark Dataset and Queries. Our testbed consists of two benchmark datasets: (i) *Lehigh University Benchmark* (LUBM) [11], synthesized based on an ontology from academic domain and (ii) *DBPedia SPARQL Benchmark* (DBPSB) [18], generated based on DBPedia. The two datasets show contrasting characteristics as summarized in Table 1. While LUBM has small number of large-sized Properties / types, DBPSB has large number of sparse Property / types. Testbed queries were adapted from LUBM and DBPSB benchmarks to incorporate scenarios with `rdf:type` optimizations. Query selectivities were varied by binding Object fields of non-*rdf:type* triple patterns (high-selectivity, denoted with postfix a), and evaluating same query with unbound Object (low selectivity marked with b). Queries with significantly high execution time are marked using $\approx$ after confirming that correct answers were produced. Approaches that failed to produce answers due to errors such as MR job failures, are marked using a red-colored $\times$. Additional results with *Waterloo SPARQL Diversity Test Suite* (WatDiv) [6] are also available on the project site [1].

Evaluated Approaches. RAPID+ was used to study the effectiveness of the R-Type model (denoted as *RR*). Apache Hive (0.12.0) was employed for testing the vertical partitioning approach. Specifically, RDF triples were loaded into HDFS and imported into a ternary table T, which was then partitioned using the following approaches:

1. **Hive(VP)** simulated the vertical partitioning approach, which partitions table T based on Property types using a map-only job per Property type.
2. **Hive(VP-Bkt)** enhances Hive(VP) by bucketing Property relations on Subject, so that triples in each bucket were sorted to enable map-side sort-merge joins.

We also evaluated other recent Hadoop-based RDF query processing systems as follows.

3. **H_2RDF+** implements a multi-indexing scheme introduced in RDF-3x on HBase[2]. Each HBase table corresponds to a distributed sorted key-value map, thus 6 tables are assigned to store six permutations of RDF triples as keys with null values.
4. **CliqueSquare** co-locates all triples having the same Subject, Property, or Object within the same node, enabling construction of clique subgraphs using map-side joins.

Other Hadoop-based approaches such as EAGRE [32] were not publicly available for evaluation. We also considered systems such as HadoopRDF [13], PigSPARQL [27], Semantic Hash Partitioning [16], and RDFHive [3]. However, they were reported to be less performant than *H_2RDF+* [22] and *CliqueSquare*, and therefore not considered here. Though we evaluated only Hadoop-based processing systems, we additionally included RDF-3x for completeness, which is the state-of-the-art centralized RDF query processing system.

Cluster Configuration. Evaluation was conducted on 80-node and 25-node Hadoop clusters in VCL[3], with each node equipped with Xeon dual core x86 CPU (2.33 GHz), 4GB RAM, and 40GB HDD. While cluster sizes may appear larger than necessary, the 25-node cluster made available 2.5TB disk space (20GB per node) which was required to store the intermediate data materialization, e.g., preprocessing of the 450GB LUBM dataset materialized 2-3 times of the input size. For RDF-3x, a dedicated single physical node was used, with Xeon quad-core CPU (E5410, 2.33GHz), 40GB RAM, and 4TB HDD. All results were averaged over three or more trials. To ensure fairness in comparison, we report execution time of MR jobs to to minimize effects of irrelevant performance factors and preparation steps. For example, we do not consider the post-processing step in *H_2RDF+* such as decompressing and reverse-dictionary-mapping results in `SequenceFiles`.

6.2 Evaluation Results

6.2.1 Pre-processing Input Datasets

Both *H_2RDF+* and *CliqueSquare* failed to import DBPSB datasets due to data format issues caused by malformed triples. Additionally, *CliqueSquare* generated too many files, i.e., approx. 670k files were generated on the 20-node cluster burdening the namenode with meta information maintenance. We expect that the number of physical files in HDFS would increase further with the increase in number of nodes, likely to cause memory issues. We also note that *Hive(VP-Bkt)* is not practical for datasets such as DBPSB with high number of Property types, making it challenging to determine appropriate number of buckets, apart from the overhead of user having to write large number of table generation statements (each executed in 1 MR job).

Storage Requirement. The aggressive compression or encoding schemes in *H_2RDF+* and *RDF-3x* reduce disk space usage. For *RR* and all *Hive* approaches, we shortened URIs

[2] https://hbase.apache.org
[3] http://vcl.drupal.ncsu.edu

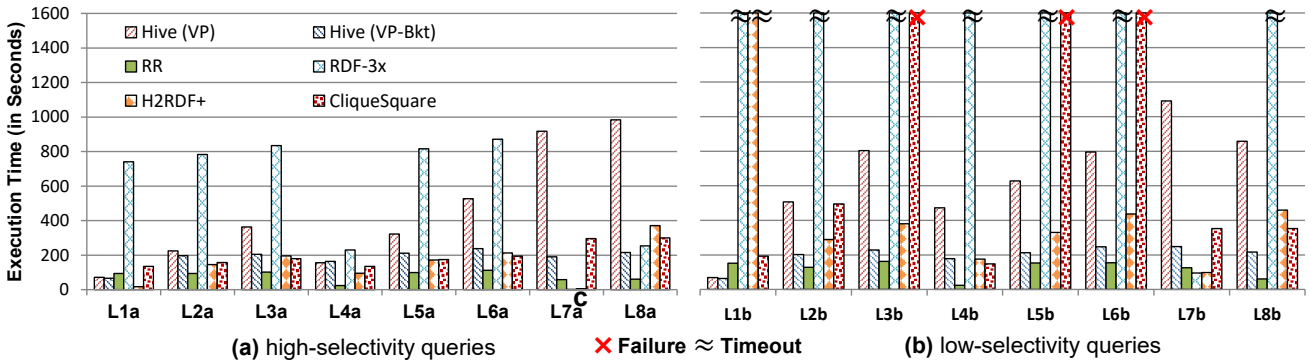

Figure 7: Comparisons of queries without `rdf:type` triple patterns using LUBM Univ 20k dataset on 80-node cluster: (a) high-selectivity queries and (b) low-selectivity queries

with pre-defined prefixes, producing QNames for URIs, e.g., <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> is mapped into `rdf:type`. In general, *RR* required less amount of storage compared to all Hive approaches by avoiding duplication of common Subjects in triplegroups. Reduced storage requirement in *RR* can also be attributed to optimization of `rdf:type` triples, which reduced the number of materialized triples by 17.4% and 6.2% for LUBM and DBPSB, respectively. *CliqueSquare* showed a high storage requirement due to replication of uncompressed triples for building three types of partitions across nodes.

Pre-processing Time. *Hive(VP)* and *Hive(VP-Bkt)* required multiple MR jobs to retrieve unique Properties / Classes in datasets and generate bucketed tables per Property. *H_2RDF+* took a significant amount of time for dictionary-encoding datasets as well as building six indexes and aggregated index statistics. Similar trends were also observed in *RDF-3x*. *RR* was mostly faster than other approaches due to a simple pre-processing step that uses a single MR job. *CliqueSquare* also used a single MR job, but was slow due to its high storage requirement. The details of a storage requirement and pre-processing time are available at our project site [1].

6.2.2 Varying Selectivity of Graph Patterns

This task studies the impact of selectivities of different Property types, e.g., Property `:name` is popular across classes while Property `:subOrganizationOf` is only associated with class `Organization`. Query *L1* consists of a single triple pattern with Property `:name`, while queries *L2*-*L4* include a second triple pattern with additional Properties that show increasing selectivity (decreasing popularity). Execution times for different approaches are shown in Fig. 7. Overall, *Hive(VP-Bkt)* was faster than *Hive(VP)* due to less expensive map-side sort-merge joins. *Hive(VP)*, *Hive(VP-Bkt)*, and *H_2RDF+* performed best for query *L1* that scans a single Property relation `:name`, while *RR* scanned all types including the Property `:name`. Note that *H_2RDF+* produced a sub-optimal plan with a centralized approach for query *L1b*.

Queries *L2*-*L4* demonstrate the benefit of *RR* due to selective scans of matching types leading to a reduction in disk read I/O. Other triple-based approaches including *RDF-3x*, *H_2RDF+*, and *CliqueSquare* tend to increase its execution time for *L1*-*L3*. All approaches were fast to process query *L4*, which involved a highly selective Property. Queries *L2*-*L3* were extended (*L5*-*L6*) to study the impact of denser star patterns. The gap between *Hive(VP)* and *RR* further increased due to scans for irrelevant triples in *Hive(VP)*. Two-star pattern queries included query *L7* (stars with same set of Properties) and query *L8* (stars differed in some Properties). In the case of *L7*, *RR* required the same set of matching types for both stars, enabling shared scans. Even for *L8*, *RR* shared scans of types that match both star patterns.

While *RDF-3x* showed impressive performance for high selective queries such as *L7a*, its performance degraded for queries such as *L1a* and *L1b* which return approximately 400*k* and 223*m* tuples, respectively. As observed in [13, 31], RDF-3X has an overhead of scanning a wide range of multi-indexes while other approaches benefit from parallel scans of horizontally partitioned datasets. *H_2RDF+* was able to process high-selectivity queries such as *L7a* very quickly using its centralized approach (Marked with 'C'). However, *H_2RDF+* showed poor performance for low-selectivity queries, mainly due to the overhead of scanning a significant amount of irrelevant triples. Additionally, multi-star queries were processed using a left-deep plan that translated to a sequence of jobs, one for each join operation [10]. For this reason, many jobs were often sequentially processed, resulting in a significant amount of execution time.

We also conducted this task using DBPSB datasets. For this purpose, we built (i) nine queries (*D1-9*) that consist of single star pattern and (ii) four queries (*D10-13*) that consist of two star patterns that connect star patterns in *D1-9*. Two principles were considered for designing these queries: (i) we chose 4-5 Properties that have different selectivities, so that we could study the effect of varying selectivity of graph patterns and (ii) we chose Properties that describe different kinds of entities (classes) in DBPedia such as *Place*, *Author*, *Artist*, *Actor*, and *Movie*, and so on. This allows us to mimic real-world queries as much as possible. Fig. 8(a) shows execution times of these queries. A general trend we observed was that other approaches often needed lengthy processing time if Properties in a query had very different selectivities, such as *D1*,*D3-5*, *D6*, and *D9*. In the case of processing such a query, other approaches needed to scan a large amount of irrelevant triples and prune them out expensive joins. However, our approach (*RR*) selectively retrieved matching types without wasting I/Os, which enables con-

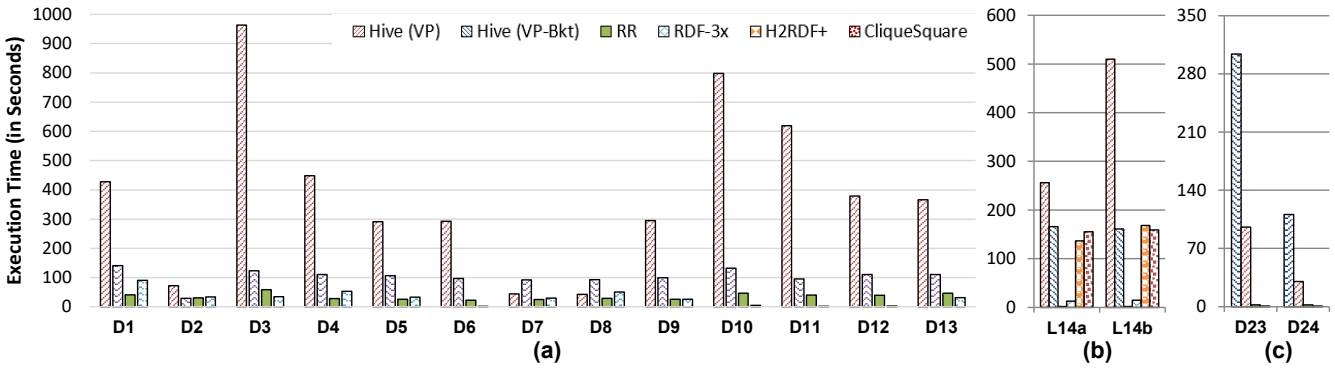

Figure 8: (a) A performance comparison of queries (DBSPB-200, 20-node cluster), (b-c) Answering negative queries (LUBM, DBPedia)

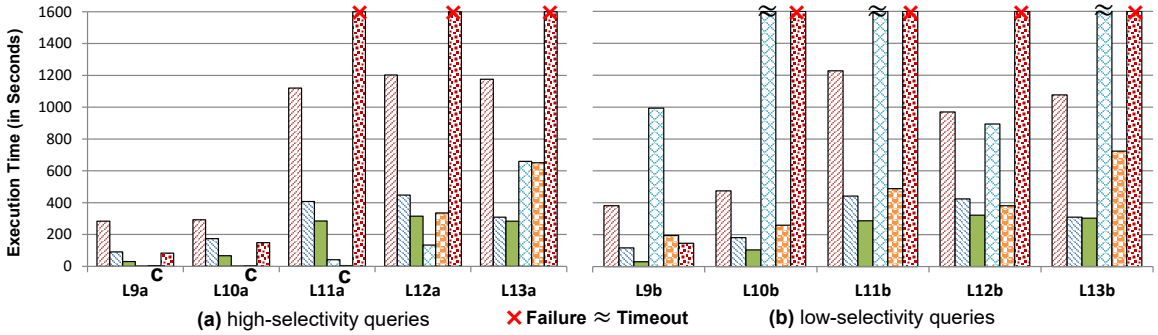

Figure 9: Evaluating queries with `rdf:type` triple patterns using LUBM datasets (Univ-20k, 80-node cluster)

sistent execution time across different queries. The similar trend was also observed for two star pattern queries.

6.2.3 *Answering Negative Queries*

While exploring large datasets with a substantial number of Properties and Classes, it is common to encounter queries that may not produce any results. A set of queries were designed to test such negative scenarios – queries *L14* (single star, LUBM) and *D23-D24* (single star, DBSPB). Fig. 8 (b-c) shows that all other approaches returned 0 results after spending time for scanning Property relations or multi-indexes and discarding unmatched results using expensive joins. On the contrary, *RR* was able to determine negative queries by checking the existence of matching types before accessing any triplegroups, therefore it halted the execution of MR jobs quickly. This allows our approach to be up to 500 times faster than other ones.

6.2.4 *Varying Number of D-inferable rdf:type Patterns*

This task studied the effectiveness of semantic optimizations. Queries *L9* and *L10* contain a single `rdf:type` triple pattern, and a non-type triple pattern that can be used to infer the `rdf:type` triple pattern. For reference, we also include queries that contain non-inferable `rdf:type` triple patterns, i.e., *L12* and *L13*. Other approaches scanned both `rdf:type` and non-`rdf:type` Property relations, e.g., *L9* reads `:teachingAssitantOf` (694MB) + $t_{\texttt{teachingAssistant}}$ (1,295MB) = 1,989MB for *Hive(VP)*. However, our approach only reads triplegroups from matching types, which is particularly effective for low-selectivity queries. Query *L11* that joins two star patterns ($L9 \bowtie L10$) also followed similar trends.

We note the effectiveness of the semantic optimizations is determined by the two main factors: (i) the number of `rdfs:domain` constraints in ontologies and (ii) the number of inferable statements satisfying the constraints. To measure the effectiveness, we additionally designed and evaluated a set of queries with inferable triple patterns, while varying the factors. The results of this evaluation are not presented due to space constraints, but they are available at our project website [1].

7. CONCLUSION AND FUTURE WORK

In this paper, we present an R-Type model based on a natural equivalence relation on R-Typed triplegroups. To exploit this typing for improving efficiency of graph pattern matching, we present a type-aware query processing model which translates queries into expressions on the typed model. We also introduce semantic optimizations that exploit ontological axioms to identify inferable triples for elimination of redundancies in query expressions.

Acknowledgement

The work presented in this paper is partially funded by NSF grant IIS-1218277 and CNS-1526113.

8. REFERENCES

[1] Project Webpage for R-Type: A Typing Model for RDF. http://research.csc.ncsu.edu/coul/RAPID+/SemStorm.

[2] RDF Schema 1.1. https://www.w3.org/TR/rdf-schema.

[3] RDFHive: A Distributed RDF Store Based on top of Apache Hive. http://tyrex.inria.fr/rdfhive/home.html.

[4] RDFS Entailment Pattern. http://www.w3.org/TR/rdf11-mt.

[5] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proc. VLDB*, 2007.

[6] G. Aluc, O. Hartig, M. T. Ozsu, and K. Daudjee. Diversified Stress Testing of RDF Data Management Systems. In *Proc. ISWC*, pages 197–212, 2014.

[7] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen. Optimizing XML Querying Using Type-based Document Projection. *ACM Trans. Database Syst.*, 38(1):4:1–4:45, 2013.

[8] L. Ding, T. Finin, Y. Peng, P. P. D. Silva, and D. L. Tracking RDF Graph Provenance using RDF Molecules. In *Proc. ISWC*, 2005.

[9] J. D. FernaNdez, M. A. MartiNez-Prieto, C. GutieRrez, A. Polleres, and M. Arias. Binary RDF Representation for Publication and Exchange (HDT). *Web Semant.*, 19:22–41, Mar. 2013.

[10] F. Goasdoué, Z. Kaoudi, I. Manolescu, J. Quiané-Ruiz, and S. Zampetakis. CliqueSquare: Efficient Hadoop-based RDF Query Processing. In *BDA - Journées de Bases de Données Avancées*, 2013.

[11] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Semantic Web Journal*, 2005.

[12] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4, 2011.

[13] M. Husain, J. McGlothlin, M. Masud, L. Khan, and B. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. 23, 2011.

[14] A. K. Joshi, P. Hitzler, and G. Dong. *Logical Linked Data Compression*, pages 170–184. 2013.

[15] H. Kim, P. Ravindra, and K. Anyanwu. Scan-Sharing for Optimizing RDF Graph Pattern Matching on MapReduce. In *Proc. CLOUD*, 2012.

[16] K. Lee and L. Liu. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *Proc. VLDB Endow.*, 6, 2013.

[17] B. McBride. Jena: A Semantic Web Toolkit. *Internet Computing, IEEE*, 6, 2002.

[18] M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. DBpedia SPARQL Benchmark: Performance Assessment with Real Queries on Real Data. In *Proc. ISWC*, 2011.

[19] T. Neumann and G. Moerkotte. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In *Proc. ICDE*, April 2011.

[20] T. Neumann and G. Weikum. RDF-3X: A RISC-style Engine for RDF. *Proc. VLDB Endow.*, 1(1), Aug. 2008.

[21] J. Z. Pan, J. M. G. Perez, Y. Ren, H. Wu, H. Wang, and M. Zhu. *Graph Pattern Based RDF Data Compression*, pages 239–256. 2015.

[22] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris. H2RDF+;: High-performance Distributed Joins Over Large-scale RDF Graphs. In *Proc. Big Data*, 2013.

[23] G. Paulley and G. Attaluri. Semantic Query Optimization in Object-Oriented Databases.

[24] M.-D. Pham, L. Passing, O. Erling, and P. Boncz. Deriving an Emergent Relational Schema from RDF Data. In *Proc. WWW*, 2015.

[25] P. Ravindra, H. Kim, and K. Anyanwu. An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In *Proc. ESWC*, 2011.

[26] P. Ravindra, H. Kim, and K. Anyanwu. Optimization of Complex SPARQL Analytical Queries. In *Proc. EDBT*, 2016.

[27] A. Schatzle, M. Przyjaciel-Zablocki, and G. Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In *Proc. SWIM*, 2011.

[28] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL Query Optimization. In *Proc. ICDT*, pages 4–33, 2010.

[29] G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and Minimization of RDF/S Query Patterns. In *Proc. ISWC*, pages 607–623, 2005.

[30] M.-E. Vidal, E. Ruckhaus, T. Lampo, A. Martinez, J. Sierra, and A. Polleres. Efficiently Joining Group Patterns in SPARQL Queries. In *Proc. ISWC*, pages 228–242, 2010.

[31] P. Yuan, P. Liu, B. Wu, et al. TripleBit: A Fast and Compact System for Large Scale RDF Data. *Proc. VLDB Endow.*, 6(7), May 2013.

[32] X. Zhang, L. Chen, Y. Tong, and M. Wang. EAGRE: Towards Scalable I/O Efficient SPARQL Query Evaluation on the Cloud. In *Proc. ICDE*, 2013.

[33] L. Zou, J. Mo, L. Chen, M. T. Ozsu, and D. Zhao. gStore: Answering SPARQL Queries via Subgraph Matching. *Proc. VLDB Endow.*, 4, 2011.