Stylus: A Strongly-Typed Store for Serving Massive RDF Data

Liang He[†][*], Bin Shao[‡], Yatao Li[‡], Huanhuan Xia[‡],
Yanghua Xiao[◇], Enhong Chen[†], Liang Jeff Chen[‡]
[†]University of Science and Technology of China, Hefei, China
[‡]Microsoft Research Asia, Beijing, China [◇]Fudan University, Shanghai, China
hshl05@mail.ustc.edu.cn, {binshao, yatli, lexi}@microsoft.com
shawyh@fudan.edu.cn, cheneh@ustc.edu.cn, jeche@microsoft.com

ABSTRACT

RDF is one of the most commonly used knowledge representation forms. Many highly influential knowledge bases, such as Freebase and PubChemRDF, are in RDF format. An RDF data set is usually represented as a collection of *subject-predicate-object* triples. Despite the flexibility of RDF triples, it is challenging to serve SPARQL queries on RDF data efficiently by directly managing triples due to the following two reasons. First, heavy joins on a large number of triples are needed for query processing, resulting in a large number of data scans and large redundant intermediate results; Second, weakly-typed triple representation provides suboptimal random access – typically with logarithmic complexity. This data access challenge, unfortunately, cannot be easily met by a better query optimizer as large graph processing is extremely I/O-intensive. In this paper, we argue that strongly-typed graph representation is the key to high-performance RDF query processing. We propose **Stylus** – a strongly-typed store for serving massive RDF data. Stylus exploits a strongly-typed storage scheme to boost the performance of RDF query processing. The storage scheme is essentially a materialized join view on entities, it thus can eliminate a large number of unnecessary joins on triples. Moreover, it is equipped with a compact representation for intermediate results and an efficient graph-decomposition based query planner. Experimental results on both synthetic and real-life RDF data sets confirm that the proposed approach can dramatically boost the performance of SPARQL query processing.

PVLDB Reference Format:
Liang He, Bin Shao, Yatao Li, Huanhuan Xia, Yanghua Xiao, Enhong Chen, Liang Jeff Chen. Stylus: A Strongly-Typed Store for Serving Massive RDF Data. *PVLDB*, 11(2): 203 - 216, 2017.
DOI: 10.14778/3149193.3149200

[*]This work was done in Microsoft Research Asia.

1. INTRODUCTION

As a W3C recommendation, Resource Description Framework (RDF) is a data model widely used for representing knowledge. An RDF data set is a collection of triples – ⟨*subject, predicate, object*⟩, where *subject* and *object* are entities or concepts and *predicate* is the relationship connecting them. Besides, inference rules are used to represent implicit triples when being applied to these explicit ones [20, 14]. The *de facto* language for querying RDF data is SPARQL [43], which generally does subgraph matching on an RDF knowledge graph.

Due to the simplicity and flexibility of RDF, a large number of RDF knowledge repositories are emerging for managing the knowledge from many fields, such as bioinformatics, business intelligence, and social networks [26, 33]. They enable machines to leverage the rich structured knowledge to better understand texts or provide intelligent services. More and more applications are powered by such RDF knowledge bases, including search engine, question answering [53], named entity linking [27, 56], query understanding [32], document representation [51], relation search, recommendation, and query expansion [21].

The size of the available knowledge built by human experts or extracted from large text corpora reaches an unprecedented scale. For example, Freebase [10] and DBpedia [9] have 1.9 billion and 3 billion triples respectively. Bio2RDF even has more than 10 billion triples. These knowledge bases, along with other open data sets, are further interlinked with each other, yielding tens of billions of facts [34].

The prevalence of RDF knowledge bases leads to many efforts on creating efficient systems for storing and querying large RDF data sets. There are many ways to organize RDF triples. The most straightforward way is to map the triples into a three-column table or its variants [28, 16, 8]. To reduce the storage redundancy for repeated subjects, a common practice is to define a generic data type, which can hold a collection of *predicate-object* pairs for an entity [55]. The storage schemes based on generic data types are generally weakly-typed – the triples and *predicate-object* pairs have to be generic enough to hold arbitrary entities.

Despite the success of existing RDF management systems, there is still large room for improvement. In this paper, we propose leveraging a strongly-typed storage scheme to serve graph queries for large RDF datasets. RDF entities are modeled by strongly-typed records. Each of them is stored as one physical record according to a pre-defined schema, which maps *predicates* to field names and *objects* to field values. The entity types defined by the pre-defined schema are essentially UDTs (user-defined types). Compared with weakly-typed schemes, the strongly-typed scheme has a few advantages. Two major ones are given as follows.

Less storage and access overhead. Under a weakly-typed storage scheme, the entity properties are usually stored as key-value pairs. The *key* of a property may be repeated many times. This may incur a large storage overhead. For a strongly-typed record, entities are managed with a pre-defined schema, which eliminates the need of using repeated predicates as property keys. Moreover, to access the value of a property under a weakly-typed scheme requires a global index lookup or a sequential scan on the entity property data, whose costs are proportional to the size of the entity. In comparison, with the aid of the pre-defined schema, this cost can be dramatically reduced.

RDF data sets contain not only triples, but also inference rules for them, for instance ρdf, RDFSPlus, RDFS Full, OWL DL, and even user-defined rule sets [45]. A large number of implicit triples are represented by applying these rules to the original triples, making RDF data different from static directed labeled graphs. For demonstration, an example RDF data set is presented in Figure 1. Those implicit triples can be derived by pre-processing in advance (forward-chaining), or by on-the-fly reasoning during query processing (backward-chaining), or a hybrid of both. The forward-chaining approach requires additional storage space for storing those derived triples. On the other hand, additional query processing cost will be incurred in the backward-chaining approach: typically, a query will be reformulated into multiple queries according to the rule sets [15, 20]. For example, given a query 'SELECT ?x WHERE { ?x niece Maria . }', the system will issue a set of additional queries, such as 'SELECT ?x WHERE { ?x marriedTo ?y . ?y niece Maria . }', besides the original one. In both cases, the overhead of storage and data access becomes an important factor to consider when realizing reasoning for real-life applications.

Less joins during query processing. Heavy joins are commonly required to process a large query. This incurs a large number of data or index scans and produces large intermediate results. Many efforts have been devoted to optimizing joins: systems such as RDF-3X [39] and Hexastore [48] build extensive indexes on all permutations of RDF elements for fast data accessing; systems such as SW-Store [2, 1] leverage a vertical partitioning approach and a column store to boost the join performance; BitMat [7] uses bit representation to reduce the join cost; TriAD [25], RDF-3X [39], gStore [57], and Trinity.RDF [55] use approaches such as graph summarization, sideways information passing, and graph exploration to do early pruning. Approaches such as cardinality estimations [38] are proposed to reduce the costs via join reordering [23]. Much has been done to reduce the join costs under a weakly-typed storage scheme, where index scans and multiple joins are needed to match the common predicates for two given entities. Here we argue that a strongly-typed storage scheme can greatly boost the query processing performance by reducing the number of joins dramatically. We use an example to illustrate the idea.

Suppose we are to answer the query 'SELECT ?x WHERE { ?x marriedTo ?s1. ?x govern ?s2. ?x starIn ?s3. ?x liveIn ? s4. }'. The query includes four triple patterns against the

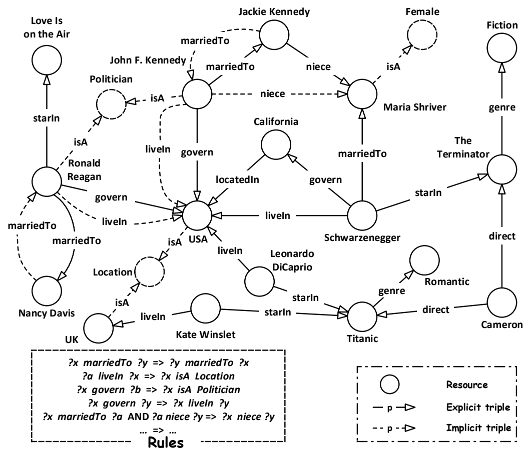


Figure 1: A Toy RDF Data Set

RDF data set shown in Figure 1. For a weakly-typed storage scheme, four steps are needed to get the final results: the first step is to scan the index of a selected predicate to initialize the candidates, followed by three steps to prune the candidates via joins. The join order may be optimized by cardinality estimation techniques. However, the size of the intermediate results produced and processed during query processing is still large. For this toy example, a typical query plan results in one index scan of 3 elements, and three merge joins of $(3, 3)$, $(2, 3)$, $(1, 3)$ elements. For a strongly-typed scheme, the process can be simplified into two steps. 1) select the entity types that contain all the four predicates appeared in the query; 2) return the instances of these entity types as final results. The process involves only entity type filtering and data access (loading the resulting entities).

Contributions. The contributions of this work are summarized as follows.

1. A strongly-typed RDF store named Stylus for querying RDF data in real time is proposed. The proposed RDF store leverages a highly optimized storage scheme for compact data storage and efficient data access. Meanwhile, an efficient query planner and a specially designed data structure are proposed for SPARQL query processing.
2. Extensive experiments are conducted to verify the scalability and efficiency of the proposed system. The experimental results show that a good storage scheme can boost the system performance by several orders of magnitude.

Paper Organization. This paper is organized as follows. The overall system design is described in Section 2. Section 3 presents the design of the strongly-typed storage scheme. SPARQL query processing on top of the proposed storage scheme is elaborated in Section 4. Section 5 and 6 present experimental results and related work respectively. Section 7 concludes.

2. SYSTEM OVERVIEW

In this section, we will introduce the overall system architecture of Stylus, which is a strongly-typed RDF store on top of a distributed in-memory infrastructure.

Stylus is built on top of a distributed in-memory key-value store that 1) supports *in-place* data access to the selected parts of a data record, instead of serializing or deserializing the whole KV pair; 2) supports *message passing* between distributed servers. An efficient distributed in-memory key-value store is an essential part of Stylus. On the one hand, efficient parallel processing of large graphs requires an efficient storage infrastructure that supports fast random data access of the graph data [35] and the main memory (RAM) is still the most viable approach to fast random access. On the other hand, the ever growing size of knowledge requires scalable solutions and distributed systems built using commodity servers are usually more economical and easier to maintain compared with scale-up approaches. Particularly, we build our RDF store on top of Microsoft Trinity Graph Engine [46, 42], which well meet the requirements discussed above.

Stylus compacts the storage by replacing RDF literals, which are used for values such as strings, numbers, and dates [41], by their integer ids. Stylus keeps a literal-to-id mapping table that translates literals of a SPARQL query into ids during query processing and maps the ids back to literals before returning results.

Most importantly, Stylus always models an RDF data set as a strongly-typed directed graph. Each node in the graph represents a unique entity using a record with several data fields. A graph node corresponds to either an *subject* or an *object* of the RDF data set. The storage scheme adopted by Stylus is given as follows: Given an RDF data set, Stylus will scan the data, extract metadata, and build a data schema for the data set. The generated schema contains all the strongly-typed data types needed for describing the data set. Stylus then stores each entity in a single record for fast data access according to the data schema. The details of the storage scheme will be elaborated in Section 3.

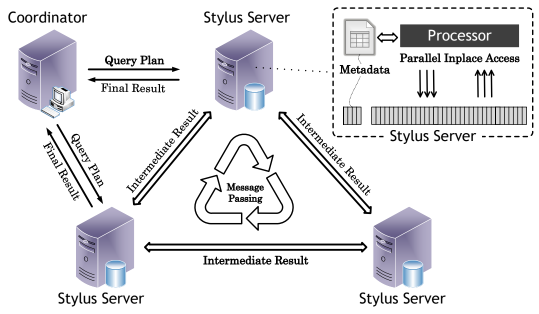


Figure 2: Overview of Stylus Architecture

Stylus is a distributed SPARQL engine on top of a strongly-typed storage scheme. The architecture of Stylus is shown in Figure 2. The whole RDF graph is partitioned over a cluster of the servers using random hashing. Each server has duplicated graph schema, but the data partitions are disjoint. A user submits a query to the query coordinator. The coordinator generates a query plan based on prepared statistics and indexes and distributes the query plan to all servers. Then, each server executes the query plan and send back the partial query results to the coordinator. On receiving all partial results, the coordinator aggregates them and return the final result to the user.

3. DATA MODELING

In this section, we elaborate the storage scheme of Stylus. The storage scheme is built on top of a key-value store. In what follows, a key-value pair is represented in the form of $(key,\ \langle value\rangle)$. To make it easier to look up the *subjects* of a given *object*, Stylus maintains a reverse triple $\langle o, p^{\intercal}, s\rangle$ for every *subject-predicate-object* triple $\langle s, p, o\rangle$, where $p^{\intercal}$ is the reverse predicate for predicate p.

3.1 A Strongly-Typed Storage Scheme for RDF

Real-life RDF data sets usually contain surprisingly a small number of distinct predicate combinations for their entities, even for the data sets that have a huge number of unique triples. For example, there are only 615 distinct combinations for more than 845 million triples in UniProt [38]. Moreover, it usually only needs a small number of predicate combinations to represent the majority of the entities, typically more than 90%. These observations have been reported for many real-life data sets, such as Yago, LibraryThings, Barton, BTC and UniProt [38].

Based on these observations, we know entities of the same category tend to share a schema "template". This motivate us to design a mechanism to extract such "templates" and use these "templates" as strongly-typed containers to represent and store knowledge graph entities. Stylus uses a data structure called xUDT to represent such "templates" – combinations of predicates. At the storage layer, each knowledge graph entity corresponds to such a combination. An xUDT consists of an identifier and a list of predicates:

$$(tid,\ \langle p_1, p_2, \ldots, p_t\rangle),$$

where *tid* is the xUDT's identifier and $\langle p_1, p_2, \ldots, p_t\rangle$ is an ordered predicate list.

Using xUDT, each knowledge graph entity is expressed as:

$$(id,\ \langle tid, offsets, obj\_vals\rangle),$$

where *id* is the identifier of an entity subject, and the tuple value represents the *predicate-object* pairs, as illustrated by Figure 3. In the value tuple, *tid* is the identifier of an xUDT; *obj_vals* represents all the *object* values as consecutive integers; *offsets* is a list of integers specifying the offsets of the *object* values for the predicates of the current entity.

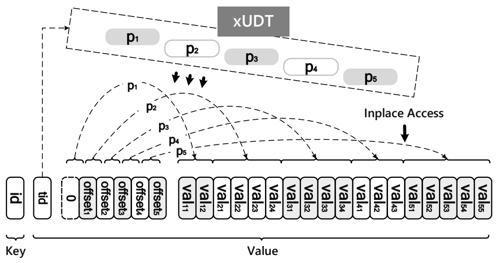


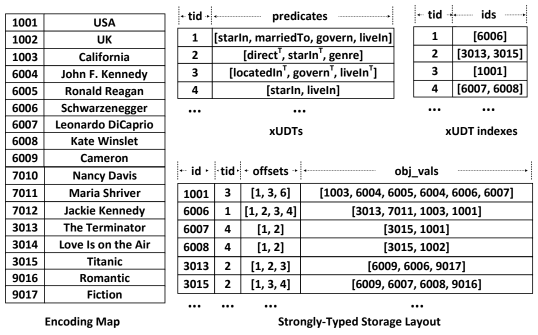
Figure 3: xUDT Illustration

This data structure can provide extremely good data retrieval performance when in-place data access is supported. The field *offsets* is designed to easily handle single-valued properties, multiple-valued properties, and even properties without an associated value. Stylus introduces a special auxiliary predicate that has no associated object, whose usage will be elaborated later. The field *obj_vals* is the concatenation of several lists, each storing *object* values associated with a predicate. Specifically, the object values in the range $[\mathit{offsets}_{i-1}, \mathit{offsets}_i)$ of *obj_vals* correspond to the i-th predicate p_i $(i > 0)$. Specially, the first predicate corresponds to the range $[0, \mathit{offsets}_0)$. Therefore, we can represent the triples with p_i $(i > 0)$ as:

$$\{\langle s, p_i, \mathit{obj\_vals}_k \rangle \mid \mathit{offsets}_{i-1} \leq k < \mathit{offsets}_i\}.$$

Then, we have

1) $\mathit{offsets}_{i-1} < \mathit{offsets}_i - 1$ for multi-valued properties;
2) $\mathit{offsets}_{i-1} = \mathit{offsets}_i - 1$ for single-valued properties;
3) $\mathit{offsets}_{i-1} = \mathit{offsets}_i$ for auxiliary properties without objects.



1001	USA
1002	UK
1003	California
6004	John F. Kennedy
6005	Ronald Reagan
6006	Schwarzenegger
6007	Leonardo DiCaprio
6008	Kate Winslet
6009	Cameron
7010	Nancy Davis
7011	Maria Shriver
7012	Jackie Kennedy
3013	The Terminator
3014	Love Is on the Air
3015	Titanic
9016	Romantic
9017	Fiction

tid	predicates
1	[starIn, marriedTo, govern, liveIn]
2	[direct^T, starIn^T, genre]
3	[locatedIn^T, govern^T, liveIn^T]
4	[starIn, liveIn]
...	...

tid	ids
1	[6006]
2	[3013, 3015]
3	[1001]
4	[6007, 6008]
...	...

id	tid	offsets	obj_vals
1001	3	[1, 3, 6]	[1003, 6004, 6005, 6004, 6006, 6007]
6006	1	[1, 2, 3, 4]	[3013, 7011, 1003, 1001]
6007	4	[1, 2]	[3015, 1001]
6008	4	[1, 2]	[3015, 1002]
3013	2	[1, 2, 3]	[6009, 6006, 9017]
3015	2	[1, 3, 4]	[6009, 6007, 6008, 9016]
...	...	...	...

Figure 4: Illustration of Stylus Storage Scheme

Figure 4 illustrates the storage scheme for the explicit triples shown in Figure 1. For the record 3015, we know its xUDT predicates are $[\mathit{direct}^{\intercal}, \mathit{starIn}^{\intercal}, \mathit{genre}]$ from its *tid* 2 and the corresponding *offsets* value is $[1, 3, 4]$, meaning the object values for these three predicates in *obj_vals* lie in the ranges $[0, 1)$, $[1, 3)$, and $[3, 4)$ correspondingly. Specifically, $\{6009\}$ for $\mathit{direct}^{\intercal}$, $\{6007, 6008\}$ for $\mathit{starIn}^{\intercal}$, and $\{9016\}$ for *genre*. The total storage cost is greatly reduced in this way. On the one hand, each xUDT record only needs to be kept once for all its instances. On the other hand, unlike the triple representation, predicates need not to be duplicated in entities' records.

xUDTs can be directly derived from the predicate combinations of a given set of entities. Stylus has a mechanism to limit the number of xUDTs when there are too many combinations. In this case, Stylus only keep the top-K most frequent predicates in xUDTs and use key-value pairs in the form of $(\mathit{id}, \langle \mathit{po\text{-}list} \rangle)$ to represents the remaining *subject-predicate-object* triples, where the key is the identifier of this entity and each element in $\langle$*po-list*$\rangle$ is a *predicate-object* pair.

3.2 Indexing and Storage Optimization

The records for xUDTs (denoted as T) are generally small in size and frequently used, which are cached and indexed on each server for fast access. In doing so we can efficiently access the content of an entity when its identifier is given.

Stylus stores a triple and its reverse triple within a single record so that the properties and property values associated with an entity can be easily retrieved. For fast data access, Stylus also builds the following indexes on each server: 1) An index that maps a predicate to the xUDTs containing the given predicate. 2) An index $I(t)$ for retrieving the entities with the xUDT t. With the index, we can enumerate the subjects and objects associated with a predicate. For example, to access $\{s \mid \langle s, p, o \rangle\}$ for p, we can translate this query to $\bigcup_{p \in t} I(t)$; while $\{o \mid \langle s, p, o \rangle\}$ for p can be answered by $\bigcup_{p^{\intercal} \in t} I(t)$; and we can use $\bigcup_{P \subseteq t} I(t)$ to retrieve all the entities with all the predicates in $P = \{p_1, \ldots, p_n\}$.

Encoding xUDT. Stylus uses 64-bit integers as the record ids. To enable fast type discovery for entities, Stylus uses several bits of the 64-bit ids as their xUDT flag. For example, masking higher 16 bits permits 65,536 flags and more than 10^{14} different IDs for each flag. Typically, each xUDT corresponds to one flag. When there are too many entities for a specific xUDT, that xUDT can associate with multiple flags.

Combining Low-Selectivity POs. While we aggregate the predicates in xUDTs, there are still lots of predicate-object pairs with low-selectivity, such as, '?x nationality USA', '?x gender Female', and '?x isA Person'. In this case, Stylus treats them as a single *auxiliary predicate* without object values: i.e. 'nationalityUSA', 'genderFemale', and 'isAPerson'. In this way, the costs of a lot of heavy joins can be greatly reduced.

3.3 Basic Data Access Operators

Assume P is a set of predicates $\{p_1, p_2, \ldots, p_n\}$, t is an xUDT, and s is an entity id. We provide the following data access operators for the SPARQL query processing module, which will be elaborated in Section 4:

- GetUDTs(P): Gets the xUDTs which are supersets of P;
- LoadEntities(t): Loads the instances of t from the indexes;
- GetObjects(s, p): Gets the objects associated with s and p;
- GetPreds(s): Gets the predicates associated with s;
- GetProps(s): Gets the *predicate-object* pairs associated with s.

The total number of xUDTs is small and Stylus has indexed the mapping between predicates and xUDTs, the operator GetUDTs(P) can be executed very fast. LoadEntities(t) loads all the entities associated with t from the prebuilt $I(t)$ index. Given s and p, GetObjects(s, p) retrieves all the objects that are associated with them. When p is not specified, all the *obj_vals* associated with s will be returned. The subjects associated p and o can be retrieved by GetObjects($o, p^{\intercal}$). In the case that the entity's predicates are unknown in advance, GetPreds(s) can list all the associated predicates. This operator is implemented by first checking the xUDT of s, and then listing all the predicates of this xUDT. In this case, one may explore the RDF graph by calling GetPreds(s) and GetObjects(s, p) iteratively. Particularly, GetProps(s) operator is built on these two operators to derive all the *predicate-object* pairs $\{\langle p, o \rangle\}$ associated with s. In summary, these operators are able to represent all the SPARQL triple patterns as listed in Table 1.

3.4 Runtime Data Update

Our storage scheme is update-friendly for the RDF data sets that may be updated constantly, such as knowledge base population, knowledge validation, and the Construct queries in SPARQL. These data updates will be handled by updating the data of xUDTs and their instances without violating the design of our storage scheme. Since the xUDTs themselves are stored as key-value pairs in memory, it supports flexible updates during runtime.

Table 1: Translating SPARQL Triple Patterns. S and P are the sets of all the subjects and predicates in the data set.

Patterns	Translated Operations
s, p, o	true if $o \in \mathsf{GetObjects}(s, p)$, otherwise false
$s, p, ?o$	$\mathsf{GetObjects}(s, p)$
$s, ?p, o$	$\{p \mid p \in \mathsf{GetPreds}(s) \wedge o \in \mathsf{GetObjects}(s, p)\}$
$?s, p, o$	$\mathsf{GetObjects}(o, p^\intercal)$
$s, ?p, ?o$	$\{\langle p, o\rangle \mid p \in \mathsf{GetPreds}(s) \wedge o \in \mathsf{GetObjects}(s, p)\}$
$?s, p, ?o$	$\{\langle s, o\rangle \mid s \in S \wedge p \in \mathsf{GetPreds}(s) \wedge o \in \mathsf{GetObjects}(s, p)\}$
$?s, ?p, o$	$\{\langle s, p\rangle \mid p \in P \wedge s \in \mathsf{GetObjects}(o, p^\intercal)\}$
$?s, ?p, ?o$	$\{\langle s, p, o\rangle \mid s \in S \wedge \langle p, o\rangle \in \mathsf{GetProps}(s)\}$

In Stylus, an entity is updated as follows: 1) when a new entity is added, its predicate combination is checked against the existing xUDTs. If it perfectly matches one of them, this entity's data will be kept according to this xUDT. While no matched xUDT found, a new xUDT is created and this entity is stored according to the new xUDT; 2) when an existing entity is updated (some triples of it are added or deleted), its new predicate combination is checked following the same procedure as that of adding a new entity. This procedure can be efficiently conducted by manipulating at most $2k$ in-memory key-value pairs, where k is the number of the entities to update; 3) when the number of xUDTs exceeds a maximum limit, the remaining entities that are not processed will be kept in the generic form of (id, $\langle po\text{-}list\rangle$). Notice that we still need to prepare/remove reverse triples for the new/deleted triples as we do for data loading.

Stylus monitors these combinations' counts and periodically swaps the xUDTs whose instance counts are less than those of the combinations stored with the generic form, along with their instances' storage layout transformed.

4. QUERY PROCESSING

We classify the subqueries of SPARQL queries into to two classes according to whether they will be matched against the preprocessed data (including indexes) or applied to the intermediate results. For the former one, we provide an optimized query planner to access the data for our storage scheme; For the latter, set operations will be performed to achieve final results. If there is no special description, we refer SPARQL queries to the first part, namely, those consisted of Basic Graph Patterns and Filters on them.

We represent a SPARQL query by a query graph $\mathcal{G}$. Nodes in $\mathcal{G}$ denote the *subjects* and *objects* appeared in the query. Directed edges in $\mathcal{G}$ denote predicates. With $\mathcal{G}$ defined, the problem of SPARQL query processing can be transformed into the problem of subgraph matching: we first decompose $\mathcal{G}$ into an ordered sequence of disjoint star-shaped patterns: $q_0, \ldots, q_{n-1}$. Such a decomposition needs to cover all the nodes and edges of $\mathcal{G}$, which resembles a typical *vertex cover* problem, as illustrated in Figure 7. Then, we find matches for q_0 and get the matches for q_i ($1 \le i \le n-1$) by exploring the graph or looking up the xUDT indexes. At the end, the query coordinator collects the partial results, aggregates them and generates the final results.

4.1 xUDT for Query Processing

xUDTs are small in size and can be processed very fast, Stylus makes extensive use of xUDTs for query processing.

Firstly, xUDTs can be directly used to answer certain queries. Many SPARQL queries are specified with constraints on predicates only. Such queries are prevalent in various applications and benchmarks, for instance, '?x p1 ?o1. ?x p2 ?o2', where ?o1 and ?o2 are only used to make sure the existence of the predicates p1 and p2. In such cases, the joins can be eliminated completely by checking the corresponding xUDTs. For this query, it can be transformed into $\bigcup_{t\in T} \mathsf{LoadEntities}(t)$, where $T = \mathsf{GetUDTs}(p1, p2)$.

Secondly, xUDTs can be used to simplify queries and reduce the number of joins. For instance, with the help of xUDTs, the query 'SELECT ?x ?y WHERE { ?x p1 ?o1. ?x p2 ?o2. ?x p3 ?y. ?y p4 ?o3. ?y p5 ?o4.}' can be simplified to 'SELECT ?x ?y WHERE { ?x p3 ?y. } ' with the constraints that the xUDT of ?x is in GetUDTs(p1, p2, p3) and that of ?y is in GetUDTs(p3$^\intercal$, p4, p5).

Thirdly, xUDTs can also be used to prune intermediate results during query processing. For example, assume the variable ?x is constrained by three triple patterns whose predicates are p1, p2, p3, then the xUDT of ?x's must be in GetUDTs(p1, p2, p3). Since xUDTs are encoded in the entity IDs, the entities that do not satisfy the constraints can be pruned immediately.

4.2 Twig Matching

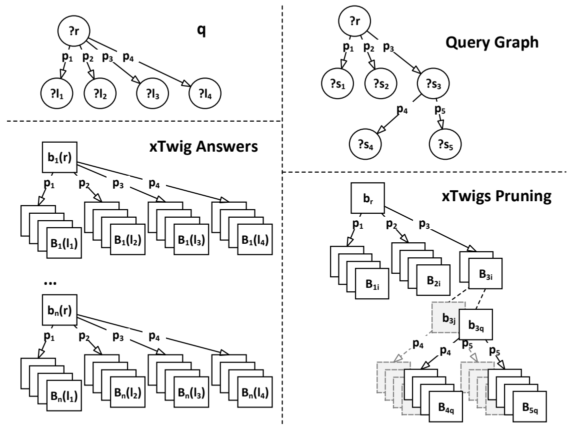


Figure 5: xTwig Examples

A Twig is a tree of height two. A Twig is a star-shaped query unit. We use $q = (r, P, L)$ to denote a Twig, where r is the root node, $P = \{p_1, \ldots, p_n\}$ are the edge labels (predicates), and $L = \{l_1, \ldots, l_n\}$ is the nodes those edges pointing to. We use $r(q), P(q), L(q)$ to represent these elements in q respectively. In addition, $V(q)$ denotes for all the variables in q.

The variables are not bound to any entities or predicates before query processing. The variables will be eventually bound to concrete entities/predicates. The matched entities or predicates for a given variable are called its bindings.

Stylus uses an extended Twig structure xTwig to postpone doing the Cartesian products during whenever possible. The matching results of xTwig can be represented as:

$$w_q = \{\langle r, \{b_r\}\rangle, \langle l_1, B_1\rangle, \ldots, \langle l_m, B_m\rangle, \ldots, \langle p_n, l_n, D_n\rangle\}$$

where $B_i = \mathsf{GetObjects}(b_r, p_i)$ for $i \in [1, m]$ while $D_i = \mathsf{GetProps}(b_r)$ for $i \in (m, n]$. The flattened representation of w_q can be written as $\{b_r\} \times B_1 \times \ldots \times B_m \times D_{m+1} \ldots \times D_n$, as illustrated by Figure 5 and Figure 6. Intuitively, this structure saves a lot of storage space: suppose $n = 3$ and

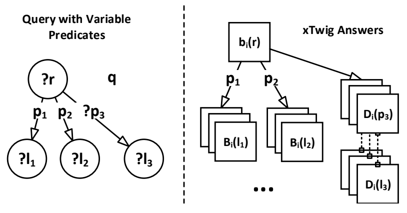


Figure 6: xTwig Example for Variable Predicates

$|B_i| = 10$ where all the predicates are specified, then the flattened representation has 10^3 records, including 4×10^3 elements, while the matching result of the xTwig contains only 61 elements.

During the query planning phase, we can infer the candidate xUDTs for each variable node in the query graph. These candidates can be either chosen as starting nodes or applied as filters to prune the intermediate results. The procedure of matching xTwigs is listed in Algorithm 1. Queries with variable predicates as well as specific predicates are matched by calling GetProps and GetObjects against the xUDT records. Before an answer to the query is derived, filters are applied to the bindings if there is any attached to the predicates or objects. A special case where a variable node appears in multiple leaves is also handled here, for example, bindings of ?o are filtered by each other for the triple patterns {?s p1 ?o . ?s p2 ?o}.

Algorithm 1 Match xTwig for $q = (r, P, L)$

```
1:  procedure Match_xTwig($q$)
2:      if $B(r(q)) = \emptyset$ then
3:          $T_q \leftarrow$ GetUDTs($P$)
4:          $B(r(q)) \leftarrow \bigcup_{t \in T_q}$ LoadEntities($t$)
5:      $W_q \leftarrow \emptyset$
6:      for $b_r \in B(r(q))$ do
7:          for $p_i \in P$ do
8:              if $p_i$ is variable then
9:                  $D_i \leftarrow$ GetProps($b_r$)
10:             else
11:                 $B_i \leftarrow$ GetObjects($b_r, p_i$)
12:         Apply the filter to each $B_i$ and each $D_i$ if any
13:         $w_q \leftarrow \{\langle r, \{b_r\}\rangle, \langle l_1, B_1\rangle, \ldots, \langle p_n, l_n, D_n\rangle\}$
14:         $W_q \leftarrow W_q \cup \{w_q\}$
15:     return $W_q$
```

4.3 Multiple Twigs Pruning

During query processing, the results of an earlier executed q can be used to filter the results of a later executed q'. This procedure involves at least two sets of xTwigs. We denote $W_{q_z} = \{w_{q_z}\}$ $(z = 1, 2)$ as such two sets of xTwigs. The binding of each Twig is in the form of $B_{q_z} = \{\langle x, B_{q_z}(x)\rangle \mid x \in V(q_z)\}$, and $B_{q_z}(x) = \{b_{q_z}(x)\}$ represents for the bindings of x in q_z. Notice that, the bindings of p_i and l_i for the same D_i are separated. Let the shared variables of the two Twigs be $V = V(q_1) \cap V(q_2) = \{v\}$. The join results of w_{q_1} and w_{q_2} are not empty if and only if $B_{q_1}(v) \cap B_{q_2}(v) \neq \emptyset$ for all $v \in V$. We can leverage this rule to prune the elements of xTwigs in order to reduce the intermediate results. The elements can be removed safely in the following two cases: 1) if $b(v) \in B_{q_1}(v)$ and $b(v)$ is not in any $B_{q_2}(v)$, we can remove $b(v)$ from $B_{q_1}(v)$, and vice versa; 2) if any $B_{q_1}(v)$ in w_{q_1} is empty after pruning, w_{q_1} can be removed from W_{q_1} totally, and vice versa.

This strategy can also be used to prune the intermediate results of multiple Twigs. For each variable node in $x \in V(\mathcal{G})$, we maintain a candidate set for it during the query execution procedure. At the beginning of the query processing, the candidate set for each variable is set to be $\bigcup_t$ LoadEntities(t), where $t \in$ GetUDTs(P_x) and P_x is the predicates associated with x in $\mathcal{G}$. As the process goes on, we will use them to explore and prune the intermediate results of a later Twig. The candidate set itself will be updated according to the new results. Specifically, suppose the candidate set of x in step i is $F^i(x)$, while the $i+1$ step produces $B^{i+1}(x)$ which is the union of all x's bindings in xTwigs in this step. Then we can update the set according to $F^{i+1}(x) = F^i(x) \cap B^{i+1}(x)$. $F^{i+1}(x)$ is used to prune the results of step $i+1$. Moreover, the values in $\Delta F^i(x) = F^i(x) - F^{i+1}(x)$ are removed from the previous xTwigs following the same pruning strategy as that for two xTwig sets.

4.4 Distributed Query Execution

In a distributed environment, the xTwig result for a query q is generated on the server where the candidate for $r(q)$ is placed. Then Stylus executes q in parallel on different servers. The overall procedure is listed as Algorithm 2. Once each server finishes processing q, the bindings of each variable in $V(q)$ are synchronized among the servers for further processing. Each xTwig is pruned according to the new candidate sets, and the remaining xTwigs are still kept on the same server. After all the q's in Q are matched, the cluster will prepare the partial results for joins on each server. Let $M_{k,i}$ be the set of remote xTwig results that the k-th server need to access for q_i. Typically, $M_{k,i}$ is determined by the local results of $q_1, \ldots, q_{i-1}$ and the remote results of q_i. If they have the same bindings for a shared variable, the remote results will be retrieved from other servers by message passing. In a worse case, all remote W_{q_i} sets need to be loaded. By this means, the derived R_k's are disjoint, that is, $R_k \cap R_{k'} = \emptyset$ for $k \neq k'$ so that no duplicated results are generated. The explanation is as follows. Since $M_{k,1} = \emptyset$, we have $R_k(q_1) = W_k(q_1)$, which means the matched results on each server for q_1 are the local results that will be joined with those of other subqueries. Since the RDF data set is disjointly partitioned in the cluster, we know the local results are disjoint, i.e., $W_k(q_1) \cap W_{k'}(q_1) = \emptyset$ for $k \neq k'$. As a result, R_k's are disjoint because the corresponding result of q_1 from each server is different from one another. As soon as the k-th server has prepared all the required xTwig results, the local joins will be performed to derive the partial results R_k. After all the partial results are generated, the coordinator collects them and generates the final results.

Algorithm 2 Distributed Query Execution

```
1:  procedure Execute_Query($\mathcal{G}, Q, K$)          ▷ $K$ is the cluster size
2:      Initialize $B(x)$ for all $x \in V(\mathcal{G})$
3:      for $q \in Q$, parallel do                       ▷ on each server
4:          $W_q \leftarrow$ Match_xTwig($q$) with $F(r(q))$
5:          Pruning $W_q$ with $F(x)$ for all $x \in V(q)$
6:          Sync and update all $F(x)$ by message passing
7:      for each server $k \in [1..K]$, parallel do
8:          for $q_i \in Q, i \neq 1$ do
9:              $R_k(q_i) = M_{k,i} \cup W_{q_i}$            ▷ message passing
10:         $R_k = R_k(q_1) \bowtie R_k(q_2) \cdots \bowtie R_k(q_n)$
11:     Aggregate all results by $R = \bigcup_{k \in [1..K]} R_k$
```

4.5 Cost Estimation

This subsection introduces how we estimate the costs of the query plans.

For each xUDT t, we denote the instance count of xUDT t in the data sets as $N(t)$. For each predicate $p \in t$, we also count the distinct object values associated with an instance of t and p as:

$$N(p|t) = |\{o \mid \langle s, p, o\rangle, s \in I(t)\}|.$$

For a variable predicate, its cardinality is estimated as the sum of the possible predicates.

For a query $q = (r, P, L)$ without constraint, the cardinality of its root r is calculated as

$$N(r|q) = \sum_{t \in T_q} N(t) = N(P(q))$$

where $T_q = \mathsf{GetUDTs}(P(q))$. Notice that this number is accurate if there is no instance of the generic xUDT. And the cardinality of leaf l_i in q is estimated by:

$$N(l_i|q) = \sum_{t \in T_q} N(p_i|t).$$

Meanwhile, the selectivity of predicate p_i in q is defined as the ratio of the leaf node cardinality in q to that of the root r, formally as:

$$S(p_i|q) = \frac{N(l_i|q)}{N(r|q)}.$$

Furthermore, we can estimate the cardinality of the node x in $V(\mathcal{G})$ as its minimum cardinality value in all the xTwigs which take it as either root or a leaf. When taken as a leaf, its new cardinality is estimated by the ratio of $\frac{N(P_x)}{N(p^{\intercal})}$ that is likely to remain after pruning. Formally, it can be written as:

$$N(x|Q) = \min_{q^r, q^l \in Q} \{N(x|q^r),\ N(x|q^l) \cdot \frac{N(P_x)}{N(p^{\intercal})}\}$$

where q^r is the query rooted at x, q^l can be any query ordered before q^r which takes x as a leaf through predicate p, and P_x represents the predicates associated with x in $\mathcal{G}$.

4.6 Query Optimization

As discussed above, given a query $\mathcal{G}$, Stylus always decomposes it into a set of subqueries q_i (Twigs). The root nodes of these subqueries form an ordered node list R. Different ways of decomposition may incur very different query processing costs. In Stylus, for each variable node $r_i \in R$, we mark it and collect the labels of unmarked edges as P_i and their end nodes other than r_i as L_i. Then we add $q_i = (r_i, P_i, L_i)$ to the query list Q to be executed. Notice that r_i may appear as objects in these edges, i.e., $\langle x, p, r_i\rangle$. In this case, we replace them with $\langle r_i, p^{\intercal}, x\rangle$ and add $p^{\intercal}$ and x to P_i and L_i respectively. The procedure is presented as DECOMPOSE procedure of Algorithm 3.

Thus, how to determine the order of R for query decomposition remains an important issue. For this purpose, we propose a Twig based query planning strategy. We take the root cardinalities of the Twigs as heuristics to derive a better query plan. The overall process is presented as procedure ORDER_SELECTION of Algorithm 3 and the statistics and metrics have been introduced in the previous subsection. There are two main loops in this algorithm. The first loop initializes each node's candidate set cardinality of their associated predicates in the query graph. After that, two different strategies are applied according the size of the query. When the nodes of the query are small in size, all permutations of these query nodes are estimated by exhaustedly enumeration to select the node order with least cost; Otherwise, each node of the query graph is set up as the first node in one iteration, a greedy strategy by picking the most selective remaining nodes is then applied to find the final order with minimum estimated cost. At the beginning of each iteration in this loop, we save the cardinalities prepared by the first loop. For each picked node r, we add r's cardinality to the total cost. Suppose q is the xTwig which covers all the edges of r except those the other ends are visited, we update the cardinality of each q's leaf according to the smaller value of the original cardinality and the one estimated by r and q. The iteration is done when all the query nodes are visited. Then, the cardinalities are restored for the next iteration. This loop ends up with all the nodes are visited, we pick the node order with least cost as the final order of query processing.

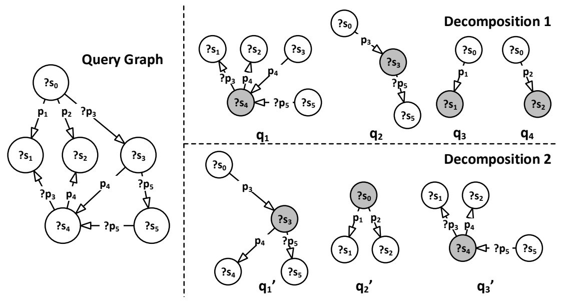


Figure 7: Query Decomposition

4.7 Answering General SPARQL Queries

Stylus supports general queries in the SPARQL 1.0 standard. While the important SPARQL component named BGP (i.e. conjunctive queries) with Filter on it is discussed above, we will discuss how the other main components of this standard are handled in Stylus as follows.

Three major query types, namely Select, Ask, and Construct, are supported by Stylus: 1) Select queries are handled as graph matching; 2) Construct queries are processed as runtime data updates; 3) Ask queries are answered by matching their patterns against the RDF data. Operators on BGPs, such as Union and Minus, are handled as set operations on the flattened results of xTwigs. When a triple pattern in a Twig query is specified as Optional, we process it as normal one but will not prune the answer if the binding for this pattern is empty (denoted as NULL). When the whole Twig query is marked as Optional, it can be handled in the same way but allows the entire xTwig answer to be empty.

As for SPARQL 1.1, we are working on the support of it. Currently, Stylus does not fully support the new features introduced by SPARQL 1.1 such as property paths yet.

Stylus has a hybrid strategy for combining the forward and backward chaining approaches to rule-based RDF reasoning/entailment. The rules are divided into two categories, i.e. forward and backward, when loaded to Stylus.

Table 2: Query Execution Time in Milliseconds on the WatDiv Data Sets. The data of RDF-3X, TripleBit, and gStore, as well as the tables of PostgreSQL, are placed on RAMDisk. Spark SQL tables are cached in memory in advance. $\mathsf{DB2RDF}_p$ and $\mathsf{DB2RDF}_s$ are DB2RDF with backends PostgreSQL and Spark SQL. △ represents **timeout** of 15 minutes and × represents **execution error**. Query categories: linear queries (L), star queries (S), snowflake-shaped queries (F), and complex queries (C). Generally, Stylus delivers better performance for most of the queries (51 out of 60 queries) and it is the only system that can answer all the queries for WatDiv-1000M.

[unit:ms]	**WatDiv-10M (10 million triples)**																			
	C1	**C2**	**C3**	**F1**	**F2**	**F3**	**F4**	**F5**	**L1**	**L2**	**L3**	**L4**	**L5**	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**
Stylus	$\underline{1}$	$\underline{5}$	$\underline{0.1}$	$\underline{0.3}$	$\underline{0.5}$	$\underline{0.6}$	$\underline{0.7}$	$\underline{0.6}$	$\underline{0.5}$	$\underline{0.5}$	$\underline{0.4}$	$\underline{0.5}$	$\underline{0.6}$	$\underline{0.4}$	$\underline{1}$	$\underline{2}$	0.6	$\underline{0.7}$	$\underline{0.3}$	$\underline{0.4}$
RDF-3X	31	387	32	12	21	23	28	25	14	12	11	7	9	27	15	12	10	11	9	10
TripleBit	△	△	△	37	23	49	63	63	28	16	6	12	37	45	17	49	$\underline{0.4}$	14	10	8
gStore	70	156	313	20	23	50	21	44	32	30	25	33	60	14	40	48	26	13	12	11
$\mathsf{DB2RDF}_p$ (10^3)	2.2	4.0	3.1	1.3	1.4	1.5	×	1.3	1.0	1.0	0.9	1.5	1.3	1.0	1.0	2.7	1.2	1.1	1.0	1.2
$\mathsf{DB2RDF}_s$ (10^3)	31	96	63	32	20	18	×	40	6	25	4	55	29	10	23	288	23	21	11	11

[unit:ms]	**WatDiv-100M (100 million triples)**																			
	C1	**C2**	**C3**	**F1**	**F2**	**F3**	**F4**	**F5**	**L1**	**L2**	**L3**	**L4**	**L5**	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**
Stylus	273	$\underline{78}$	$\underline{15}$	$\underline{2}$	$\underline{5}$	$\underline{117}$	$\underline{7}$	$\underline{0.8}$	$\underline{0.4}$	$\underline{2}$	$\underline{0.2}$	$\underline{3}$	$\underline{2}$	$\underline{0.3}$	$\underline{9}$	$\underline{23}$	$\underline{5}$	$\underline{3}$	$\underline{2}$	0.1
RDF-3X	$\underline{174}$	576	167	23	50	146	115	121	55	20	17	10	20	39	43	37	15	21	15	10
TripleBit	△	△	△	39	102	155	121	66	30	31	11	55	55	21	60	312	14	23	26	$\underline{0.01}$
gStore	488	1032	2589	13	139	386	75	127	101	120	88	152	312	29	229	251	79	35	45	30
$\mathsf{DB2RDF}_{p1}$ (10^3)	54	131	218	11	6	2	×	1.0	1.0	5	0.9	6	3	0.6	3	52	4	2	0.7	0.5
$\mathsf{DB2RDF}_{p2}$ (10^3)	9	12	34	1.0	1.4	1.1	×	0.8	0.6	0.6	0.5	1.9	1.3	0.5	0.8	11	3	1.0	0.5	0.5
$\mathsf{DB2RDF}_s$ (10^3)	123	260	328	518	△	125	×	374	52	39	38	△	△	171	235	△	65	840	169	167

[unit:ms]	**WatDiv-1000M (1 billion triples)**																			
	C1	**C2**	**C3**	**F1**	**F2**	**F3**	**F4**	**F5**	**L1**	**L2**	**L3**	**L4**	**L5**	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**
Stylus	2270	$\underline{1178}$	$\underline{199}$	$\underline{22}$	$\underline{50}$	$\underline{59}$	$\underline{84}$	$\underline{1.4}$	$\underline{0.8}$	772	$\underline{0.6}$	80	$\underline{16}$	$\underline{0.1}$	$\underline{143}$	312	91	$\underline{31}$	$\underline{19}$	4
RDF-3X	$\underline{1696}$	△	739	61	378	816	681	1155	368	70	15	$\underline{18}$	95	27	266	$\underline{190}$	$\underline{63}$	66	74	9
TripleBit	△	△	△	133	964	76	174	25	16	$\underline{223}$	14	206	81	10	7029	7157	107	95	45	$\underline{0.01}$

Table 3: The Counts of *Triple Patterns* (#TPs) and *Planned Twigs* on WatDiv

	C1	C2	C3	F1	F2	F3	F4	F5	L1	L2	L3	L4	L5	S1	S2	S3	S4	S5	S6	S7
#TPs	8	10	6	6	8	6	9	6	3	3	2	2	3	9	4	4	4	4	3	3
#Twigs of WatDiv-10M	3	5	1	3	3	3	3	3	3	3	2	2	3	2	3	2	3	2	2	2
#Twigs of WatDiv-100M	3	4	1	4	3	4	4	3	3	3	2	2	3	2	3	2	3	3	2	2
#Twigs of WatDiv-1000M	3	4	1	3	3	3	3	3	3	3	2	2	3	2	3	2	2	3	2	2

The overall reasoning procedure of Stylus is given as follows: **1)** After the data loaded, the forward rules for materialization are applied to the original triples, and the derived triples are stored in the same way Stylus stores the normal ones. An efficient in-memory materialization method is proposed on top of a vertical partitioning storage scheme in Inferray [45], which is realized in Stylus by mapping the *so* table for predicate p into $\{s, o \,|\, s \in \mathsf{LoadEntities}(t) \wedge o \in \mathsf{GetObjects}(s)\}$ for all $t \in \mathsf{GetUDTs}(p)$, and *os* table into that of all $t \in \mathsf{GetUDTs}(p^\intercal)$ accordingly. Meanwhile, the materialization process highly relies on the joins on subjects/objects of two triples, which could also benefit from our storage scheme design as discussed earlier. **2)** When a query is processed by Stylus, the query is reformulated into multiple sub-queries according to the backward rules, and these derived queries are executed sequentially to obtain the final result (for smarter query reformulation methods, please refer to [15, 14]).

5. EXPERIMENTAL EVALUATION

Since Stylus is designed to serve massive real-life RDF data sets, we want to answer the following additional questions: **1) Flexibility.** How fast Stylus and the other systems process different types of queries? **2) Scalability.** How large are the data sets that Stylus can scale to? What is the system speedup with regard to the number of machines in a distributed setting? **3) Real-life Performance.** What is the performance of Stylus compared with other systems on real-life data?

5.1 Setup

We choose four representative RDF stores belonging to different categories from the perspective of storage schemes – weakly-typed, strongly-typed, RDBMS based, and non-RDBMS based for comparison. These systems are publicly available and acknowledged to be fast for query processing recently. Specifically, we compare Stylus with RDF-3X [39], TripleBit [54], gStore [22, 57], and the DB2RDF [17, 12] with two different backends on Docker – PostgreSQL (denoted as DB2RDF$_p$) and Spark SQL (denoted as DB2RDF$_s$). In addition, we compare Stylus with Trinity.RDF [55] which is built on the same infrastructure to examine the performance improvement gained by the system design of Stylus.

Stylus is implemented in C# on top of Microsoft Graph Engine and open-sourced on GitHub [44]. In the experiments, we set the maximum number limit of xUDTs to 50,000. For the single-machine experiments on Windows, we set up a machine with 96 GB DDR3 RAM and two 2.67 GHz Intel(R) Xeon(R) X5650 CPUs, each of which has 12 threads, and one 15000-RPM SATA local disk. The operating system is 64-bit Windows Server 2008 R2 Enterprise. The Linux-based systems RDF-3X, TripleBit, gStore, DB2RDF$_p$, and DB2RDF$_s$ are deployed on machines running 64-bit Ubuntu 14.04 LTS with the same hardware specifications. Since both Stylus and Trinity.RDF are built on top of an in-memory graph engine, we use RAMDisk as the storage for RDF-3X, TripleBit, and gStore for a fair comparison. For DB2RDF$_p$, PostgreSQL does not support in-memory tables, we set up a RAMDisk folder as its database directory. For DB2RDF$_s$, tables are cached in memory for query processing in advance. A 5-machine cluster is used for the distributed experiments, each machine has the same specs as the machine used in the single-machine experiments. To measure the query execution time, we excluded the time for database connection, literal/ID mapping, query parsing and planning for all the systems.

Algorithm 3 Query Planning ($\mathcal{G}$)

```
1: procedure DECOMPOSE($\mathcal{G}$)
2:     $R \leftarrow$ ORDER_SELECTION($\mathcal{G}$)
3:     for $r \in R$ do
4:         $E(r) \leftarrow r$'s unmarked edges in $E(\mathcal{G})$
5:         Replace all edges of $\langle x, p, r \rangle$ by $\langle r, p^{\intercal}, x \rangle$ for $r$
6:         $P \leftarrow$ predicates in $E(r)$
7:         $L \leftarrow$ target node labels in $E$
8:         $q \leftarrow (r, P, L)$
9:         $Q \leftarrow Q \cup \{q\}$
10:    return $Q$
11:
12: procedure ORDER_SELECTION($\mathcal{G}$)
13:    for $r \in \mathcal{G}$ do
14:        $P_r \leftarrow r$'s associated predicates
15:        $N(r) \leftarrow \sum_{t \in \mathsf{GetUDTs}(P_r)} N(t)$
16:    $C_{min} \leftarrow MAX, R_{min} \leftarrow \emptyset$
17:    if $|V(\mathcal{G})|$ less than a threshold then
18:        for $R \in$ permutations of $V(\mathcal{G})$ do
19:            $C \leftarrow 0, V \leftarrow \emptyset$
20:            Backup $N(\cdot)$
21:            for $r \in R$ do
22:                $V \leftarrow V \cup \{r\}, C \leftarrow C + N(r)$
23:                $q \leftarrow$ the Twig covers all the edges of $r$
24:                for $l_i \in L(q) - V$ do
25:                    $N(l_i) \leftarrow \min\{\, N(l_i),\ N(r) \cdot S(p_i|q) \,\}$
26:            Restore $N(\cdot)$
27:            if $C < C_{min}$ then
28:                $C_{min} \leftarrow C, R_{min} \leftarrow R$
29:    else
30:        for each $r_0 \in V(\mathcal{G})$ do
31:            $C \leftarrow N(r_0), R \leftarrow \{r_0\}$
32:            Backup $N(\cdot)$
33:            while $|R| < V(\mathcal{G})$ do
34:                $r \leftarrow \arg\min_r N(r)\ s.t.\ r \in V(\mathcal{G}) - R$
35:                $R \leftarrow R \cup \{r\}, C \leftarrow C + N(r)$
36:                $q \leftarrow$ the Twig covers all the edges of $r$
37:                for $l_i \in L(q) - R$ do
38:                    $N(l_i) \leftarrow \min\{\, N(l_i),\ N(r) \cdot S(p_i|q) \,\}$
39:            Restore $N(\cdot)$
40:            if $C < C_{min}$ then
41:                $C_{min} \leftarrow C, R_{min} \leftarrow R$
42:    return $R_{min}$
```

Table 4: Statistics of the Data Sets

Data Sets	#Triples	#S/O	#P
WatDiv-10M	10,916,457	1,052,571	86
WatDiv-100M	108,997,714	10,250,947	
WatDiv-1000M	1,092,155,948	97,390,412	
Yago2s	173,033,130	43,509,579	100
LUBM-40	5,475,475	1,309,072	17
LUBM-80	11,067,027	2,644,415	
LUBM-160	22,016,066	5,259,588	
LUBM-320	43,939,328	10,494,125	
LUBM-640	88,099,158	21,037,012	
LUBM-10240	1,410,024,788	336,711,191	

Table 5: Load Time (minutes)

[unit:min]	WatDiv			Yago2s
	-10M	-100M	-1000M	
Stylus	3	39	981	92
RDF-3X	3	36	467	73
TripleBit	2	35	485	47
gStore	3	59	Aborted	1551
DB2RDF$_p$	95	985	Aborted	Aborted
DB2RDF$_s$	32	728	Aborted	1083

Table 6: Storage Overhead (GB)

[unit:GB]	WatDiv			Yago2s
	-10M	-100M	-1000M	
Raw Data	1.4	14.5	148.0	24.7
Stylus	0.9	2.9	24.0	5.3
RDF-3X	0.5	5.1	60.4	9.2
TripleBit	0.2	2.3	25.0	5.7
gStore	0.7	6.7	–	27.6
DB2RDF$_p$	14.0	138.0	–	–
DB2RDF$_s$	3.4	34.1	–	60.1

Table 7: Storage Overhead (GB) of Stylus for LUBM

LUBM- [unit:GB]	40	80	160	320	640	10240
Raw Data	0.9	1.8	3.5	7.1	14.1	227.9
Stylus	0.8	1.1	1.5	2.5	4.4	41.9

5.2 Data Sets

We use a real-life data set Yago2s [52] and two RDF benchmarks, namely LUBM [24] and WatDiv [4] in the experiments. The reasons are as follows: **1)** WatDiv is proposed for benchmarking RDF management systems across a wide spectrum of SPARQL queries, including linear, star, snowflake-shaped, and very complex ones. Real-life applications are diverse at structures [18, 4]. It is hard to consistently achieve the same level of good performance for all these queries if the storage scheme is workload-oblivious [5]. **2)** Yago2s is a large real-life data set [18]. It consists of facts extracted from Wikipedia and integrated with WordNet and GeoNames. **3)** LUBM is a RDF benchmark widely used for comparing the performance of RDF stores [12, 7, 54, 55, 25]. A nice feature of this benchmark is that the data metrics are kept linear to the data scales.

The LUBM data sets of different sizes are generated by the LUBM data generator v1.7. The WatDiv data sets of the same sizes are obtained from the WatDiv website [47]. The statistics of these data sets are listed in Table 4.

For LUBM data sets, we choose the same 7 queries (Q1-Q7) as given in [7] and prepare an auxiliary predicate for each predicate-object pair which takes *rdf:type* as predicate due to their low selectivity. For the WatDiv data sets, we generate 20 queries for both data sets according to the query templates of the following four categories: linear queries (L), star queries (S), snowflake-shaped queries (F), and complex queries (C). WatDiv query templates (C1-C3, F1-F5, L1-L5, S1-S7) are obtained from the WatDiv benchmark website. There are no existing queries available for Yago2s, we designed 12 queries of different structures for the experiments. All the queries used in our experiments can be found in the code repository of Stylus. Especially, since the predicates may be unknown in real-life scenarios, two queries with variable predicates are included, one of which contains joins on the variable predicates.

Loading. All data sets are loaded into the systems using their own data loaders, the load times taken by the WatDiv and Yago2s data sets are listed in Table 5. Stylus relies on paired triples which are grouped (no need to sort) by the first column (subjects and reversed objects) for loading. The load times taken by Stylus for loading WatDiv-10M, WatDiv-100M, and Yago2s are close to those of RDF-3X and TripleBit. And it is able to load WatDiv-1000M within a reasonable time compared to that of TripleBit. All these systems are able to load WatDiv-10M and WatDiv-100M. DB2RDF$_p$ and DB2RDF$_s$ take much longer time for all the data loading. Compared with DB2RDF$_s$, DB2RDF$_p$ takes more time to build indexes and is faster to answer queries as we will see later. For Yago2s, gStore spends longer time for data loading, DB2RDF$_p$ failed to load it while DB2RDF$_s$

Table 8: Query Execution Time in Milliseconds on Yago2s. The data of RDF-3X, TripleBit, and gStore are placed on RAMDisk, while the tables of DB2RDF$_s$ are cached in advance. Queries Y11 and Y12 have variable predicates and Y11 requires joins on the variable predicates. △ represents for **timeout** of 15 minutes and × represents **execution error**. Stylus is the winner of 9 out of 12 queries and is able to complete Y11 and Y12 within a second.

[unit:ms]	Y1	Y2	Y3	Y4	Y5	Y6	Y7	Y8	Y9	Y10	Y11	Y12
Stylus	$\underline{25}$	$\underline{7}$	12	$\underline{75}$	55	$\underline{1454}$	$\underline{0.07}$	$\underline{0.3}$	$\underline{6}$	17	$\underline{152}$	$\underline{935}$
RDF-3X	82	98	54	86	$\underline{29}$	2264	7	8	16	$\underline{10}$	54322	43334
TripleBit	66	12	$\underline{8}$	177	61	3471	0.3	11	207	345	×	×
gStore	47	307	348	114	283	3431	4	16	188	168	1642	6081
DB2RDF$_s$ (10^3)	△	△	266	625	339	608	18	40	184	218	×	×

Table 9: The Counts of *Triple Patterns* (#TPs) and *Planned Twigs* on Yago2s

	Y1	Y2	Y3	Y4	Y5	Y6	Y7	Y8	Y9	Y10	Y11	Y12
#TPs	8	10	6	5	6	5	1	2	3	3	2	5
#Twigs	5	7	6	2	4	4	1	2	2	2	2	2

finished the loading after about 16 hours. RDF-3X takes almost the same time to load WatDiv-1000M as TripleBit, while gStore fails to load this data set. DB2RDF$_p$ and DB2RDF$_s$ aborted the data loading for WatDiv-1000M after a few days. Thus gStore, DB2RDF$_p$, and DB2RDF$_s$ are excluded from the experimental evaluation on WatDiv-1000M.

5.3 Evaluation

We have conducted a set of experiments to answer the three questions raised at the beginning of this section.

Flexibility. To answer the first question, the performance of all the systems on three WatDiv data sets is compared in Table 2. Since the data tables of DB2RDF$_p$ for WatDiv-100M data set are too large to fit into the main memory, we run each query twice (denoted as DB2RDF$_{p1}$ and DB2RDF$_{p2}$ respectively) consecutively to make the queries are warm-cached.

Generally, RDF-3X is very stable at answering a wide range of queries on WatDiv-10M and WatDiv-100M, and outperforms the other three systems for query C1 on WatDiv-100M. The performance (response time) of TripleBit is close to that of RDF-3X but with a larger variance: Compared to RDF-3X, the performance of some queries of TripleBit is better, for instance, query L3 for WatDiv-10M. It even achieves the best performance for query S4 on WatDiv-10M. Meanwhile queries such as F2, L4, and S3 for WatDiv-100M take much longer time to finish. It even fails to answer queries C1-C3 for the three WatDiv data sets within 15 minutes. In contrast, gStore is able to answer all these three queries although it generally takes slightly longer time to answer other queries. Compared to RDF-3X and TripleBit, it has the best performance for queries C2, F4, S1 on WatDiv-10M, and queries F1, F4 on WatDiv-100M. DB2RDF$_p$ and DB2RDF$_s$ complete these queries in seconds for WatDiv-10M and WatDiv-100M. As we pointed out earlier, DB2RDF$_p$ takes more time to load data, but has much better query processing performance than DB2RDF$_s$. Both RDF-3X and TripleBit leverage their compact representations for query processing, their execution procedures use joins for processing triple patterns. The DB2RDF systems take the advantages of mature database techniques, however, their representations are neither compact (for fast I/O operations) nor strongly-typed (for fine-grained data access), thus take relatively a long time to answer the queries.

In comparison, Stylus is built on a compact and strongly-typed storage scheme. It outperforms the other state-of-the-art systems for 51 out of 60 queries, many of which are orders of magnitude faster. In terms of flexibility, Stylus is able to serve the WatDiv-1000M data set and handle complex queries C1-C3 efficiently compared to TripleBit (3 timeouts) and RDF-3X (1 timeout), which can also load the data set.

Scalability. We conducted a set of experiments to verify the scalability of Stylus. Five LUBM data sets are chosen for this purpose.

The query execution times of Stylus on LUBM data sets of different sizes are compared in Table 13. The results of Q3-Q6 are almost the same. As we can see, for these selective queries (Q4-Q6), the performance of Stylus is very stable no matter how large the data sets are. For Q1, Q2 and Q7, Stylus can complete the queries within a reasonable time; the response times are basically linear to the data sizes.

We also examined the speed-up for each LUBM query by varying the number of servers from 2 to 8 as shown in Figure 8. As we can see, the performance of Stylus is stable for selective queries Q3-Q6 even the network communication costs are included. For Q1 and Q2, the response times decrease dramatically with respect to the number of machines. This confirms that Stylus can efficiently utilize the parallelism of a distributed system. As for Q7, the size of the final result is very large, where aggregating those results from the servers dominates the response time.

Real-life Performance. To evaluate the real-life performance, the query execution times of these systems are compared in Table 8. Consistent with our previous observation, the execution times of RDF-3X and TripleBit are close except that TripleBit has a larger variance. Specifically, RDF-3X outperforms the other three systems for Y5, Y10, while TripleBit performs very good for Y2 and Y3.

Compared to RDF-3X and TripleBit, the execution times of gStore are close to that of TripleBit, but worse for Y2-Y10 compared to RDF-3X. And it gets the best performance compared to RDF-3X and TripleBit for Y1. In addition, gStore is able to answer the two queries with variable predicates (Y11 and Y12) within a few seconds where RDF-3X spends tens of seconds to respond and TripleBit even fails to process. DB2RDF$_s$ takes much longer time to answer these queries compared the other systems and fails to answer Y11 and Y12. In this experiment, Stylus delivers good performance for nearly all the queries, outperforms RDF-3X, TripleBit, gStore, and DB2RDF$_s$ for 9 out of 12 queries. For some of the queries, specifically, Y2, Y7-Y9, Y11, and Y12, Stylus is orders of magnitude faster.

5.4 Discussion

In this subsection, we discuss the factors that contribute to the overall performance of Stylus.

Compact Storage and Fast Data Access. We measured the sizes of the storage space of the systems used in our experiments for different data sets. For Stylus, the xUDT meta records are also included the storage overhead. The results are listed in Table 6 and 7. As we can see, the storages of Stylus are more compact than those of gStore, DB2RDF$_p$, and DB2RDF$_s$. Compared with the other two systems that heavily rely on customized compressions, Stylus can still keep a quite compact storage for these data sets due to its strongly-typed storage scheme.

Fast data access is a key performance factor, as illustrated by the response times of L1-L5 queries for WatDiv data sets and Y3, Y7-Y8 for Yago2s. Due to the query decomposition strategy of Stylus, these queries are all decomposed into single leaf Twigs. In these cases, Stylus matches triple patterns in the same way as other systems, therefore the performance gaps mainly come from data access.

The effort of fast data access was also illustrated in the comparison with Trinity.RDF by the results shown in Table 10 for the single-machine experiments and Table 12 for the distributed experiments. Note that those results on LUBM-10240 are all conducted on the distributed cluster with the same configuration. Given the same underlying infrastructure, Stylus outperforms Trinity.RDF for Q4-Q6 which are all selective queries. The big performance gap mainly comes from the storage scheme because Stylus and Trinity.RDF has the same underlying infrastructure – Trinity. Note for Q1 and Q7, Stylus outperforms Trinity.RDF on LUBM-160, but performs worse on LUBM-10240, however this is not caused by data access and we will discuss this later.

Less Joins. In the experiments, we counted the number of joins for each query by '#TPs' and '#Twigs' as shown in Table 3, 9, and 11, where '#TPs' represents the number of triple patterns (for the compared systems) and '#Twigs' represents those of decomposed Twigs (for Stylus). Typically, the join numbers are $n - 1$, where n is #TPs or #Twigs. An obvious observation is that the more joins in the queries, the longer time these systems take to complete the queries. As we can see, Twigs are much less than triple patterns for most queries, meaning there are usually less joins in Stylus. The Twig counts of some queries are even reduced to 1, indicating that joins are completely eliminated (query C3 for WatDiv). For Q3 on the LUBM data sets, Stylus detected an empty xTwig result during execution and completed the query immediately without further joins by leveraging the strongly-typed storage scheme.

Further Improvement. Trinity.RDF performs better than Stylus for Q1 and Q7 on LUBM-10240. This is because the Twig based planner cannot give an optimal query execution plan under some special circumstances, for instance, a query with a triangle structure (say a, b, c), where the best plan is to match '$a \rightarrow b, b \rightarrow c, c \rightarrow a$' in order. However, Stylus will always decompose it into two Twigs in this case, which may lead to a suboptimal plan, such as those of Q1 and Q7. When the data size is small, Stylus is able to complete the query very fast if the data access cost dominates the overall cost. However, the triple based planner might become better when the data sets become larger. To solve this problem, Stylus needs a more sophisticated planner to better decompose queries. This is our ongoing work.

6. RELATED WORK

According to their storage schemes of RDF stores, we classify existing RDF stores into two categories: *weakly-typed* and *strongly-typed*. Under a weakly-typed storage scheme, entities are stored either as a collection of triples or by an entity-based generic type holding a collection of predicate-object pairs; while under a strongly-typed one, entities are modeled and stored via the user-defined types (UDTs).

6.1 Weakly-Typed Schemes

Weakly-typed schemes are widely adopted due to its simplicity. Here we discuss three representative weakly-typed schemes: triple table, weakly-typed graph, and DB2RDF.

Triple tables, such as 3store [28], Oracle [16], and Redland [8], store RDF data in a large table in relational databases using a three-column schema (*subject, predicate, object*). An entity is separated into s records where s is the triple count of this entity. To speed up data retrieval in triple tables, indexes are usually built on all possible combinations of (*subject, property, object*) [48, 30, 19, 39, 50].

Due to the graph nature of RDF data, graph-based storage schemes [55, 57, 11, 31, 36, 6] are proposed to model and store RDF data. The performance of graph operations is improved thanks to the graph-structured layout. In Trinity.RDF [55], nested key-value pairs are used within entities' records to maintain the predicate-object values, the same as a generic type scheme thus is weakly-typed.

The storage model proposed in [12] leverages an optimized generic type scheme. An entity's collection of predicate-

Table 10: Query Execution Time (ms) on LUBM-160

[unit:ms]	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Stylus	213	25	0.04	0.05	0.04	0.6	536
Trinity.RDF	281	132	110	5	4	9	630

Table 11: The Counts of *Triple Patterns* (#TPs) and *Planned Twigs* on LUBM-160

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
#TPs	6	2	6	5	2	4	6
#Twigs	2	1	2	1	1	2	2

Table 12: Query Execution Time (ms) on LUBM-10240

[unit:ms]	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Stylus	20976	4440	7	2	2	5	31764
Trinity.RDF	12648	6018	8735	5	4	9	31214

Table 13: Query Execution Time (ms) of Stylus on LUBM

[unit:ms]	Q1	Q2	Q3	Q4	Q5	Q6	Q7
LUBM-40	38	3	0.02	0.05	0.03	0.6	174
LUBM-80	92	6	0.03	0.04	0.03	0.3	278
LUBM-160	213	25	0.04	0.05	0.04	0.6	536
LUBM-320	464	54	0.03	0.05	0.07	0.4	912
LUBM-640	1368	111	0.03	0.04	0.03	0.5	1763

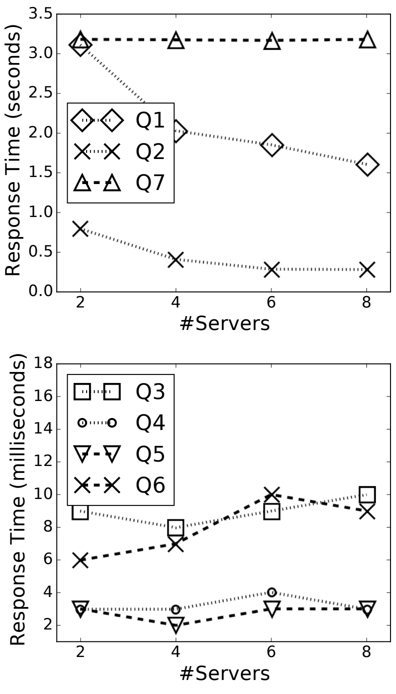


Figure 8: Parallel Speed-up on LUBM-10240

object pairs are stored in one big table with a fixed number of columns. The first column is set for the subject and others for the associated predicate-object pairs. Predicate-object pairs are assigned to the columns according to the hash values on the predicates. If hash collisions happen among the predicates within an entity, multiple records are leverage to store the data of this entity. This model speeds up the query processing by the hash mechanism. It proves more efficient than traditional RDBMS based RDF stores, especially those built on a triple table storage scheme [12].

6.2 Strongly-Typed Schemes

The idea of defining entity types using grouped predicates has already been adopted in the RDBMS based models. However, they failed to deliver high performance due to the following reasons. First, the multi-valued properties are hard to arrange in relational tables as faced by property table based methods. In this case, an individual table is necessary for storing them, but multi-valued properties spreading all over the data sets reduce this model back to a big triple table approach; second, real-life entities may be of multiple types. A common practice is to span the entity's data across several records. However, additional joins are inevitable for aggregating entity segments in this case. The more records those entities are partitioned, the more joins are needed for aggregation; third, a fixed schema agnostic of the data set is likely to produce lots of NULLs for the absent properties of entities. It is very costly to store those unnecessary NULLs, especially for a large number of predicates.

According to the number of types with which a predicate can be associated, strongly-typed storage schemes are further classified into *partitioned schemes* and *overlapped schemes*. In a partitioned scheme, including *property table*, *wide table*, and *vertical partitioning scheme*, each predicate is associated to one user defined types; while in an overlapped scheme, predicates can be associated to more than one types.

Property table schemes and techniques [13, 3, 49, 29] are proposed to address the query performance issues in triple table schemes. In a property table, predicates are partitioned into disjoint clusters, each of which corresponds to a table, but it is usually not strictly strongly-typed in practice. An extreme case of property table schemes is a very *wide table* which has only one UDT. The UDT contains all the predicates and each of them corresponds to a column in the wide table. Bit based solutions, such as BitMat [7] and RDFCube [37], are proposed for compact representation.

Vertical partitioning [1, 2] is another extreme case of property table schemes. For each predicate, a table with two columns is created to store the subject-object pairs. In this way, multi-valued properties can be well handled by splitting them to multiple rows in a table and NULL values are eliminated. However, an entity with multiple predicates will be distributed in multiple tables. Queries on this kind of entities have to aggregate data from multiple tables and thus have the performance issue in practice.

Although partitioned schemes are strongly-typed, their types are defined by partitioning the predicates into one or more disjoint clusters. In contrast, overlapped schemes group predicates and define types around the entities in a data set. For instance, in [38], predicates that are shared among multiple entities are grouped together into a Characteristic Set (CS) to form a type, which may overlap with another on the same predicates. In this way, an entity will have less chance of being divided into multiple tables than in partitioned schemes. While CSs have been used for efficient join ordering [23], a CS-based scheme based on relational tables has been exploited for reducing joins on subjects [40] during RDF query processing. However, multi-valued properties are still not handled gracefully. Our proposed xUDTs go further than these methods, a tailored storage scheme is proposed for fast data access and less joins.

7. CONCLUSION

In this paper, we proposed a high-performance RDF store called Stylus for serving SPARQL queries on massive RDF data in nearly real time. Stylus adopts a strongly-typed storage scheme for modeling and storing RDF entities in a very compact manner. The carefully designed storage scheme and a highly optimized SPARQL query processor enable Stylus to serve a wide range of SPARQL queries efficiently. Extensive experiments have been conducted to evaluate the proposed system. The experimental results show that Stylus is not only efficient for answering SPARQL queries but also scalable for handling very large RDF data sets.

8. ACKNOWLEDGEMENT

This work was partially supported by grants from the National Science Foundation for Distinguished Young Scholars of China (Grant No. 61325010), the National Natural Science Foundation of China (Grant No. U1605251), and the National Key Foundation for Exploring Scientific Instrument of China (Grant No. 61727809).

9. REFERENCES

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

[2] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Sw-store: A vertically partitioned dbms for semantic web data management. *The VLDB Journal*, 18(2):385–406, Apr. 2009.

[3] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *International Semantic Web Conference*, 2001.

[4] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of RDF data management systems. In *International Semantic Web Conference*, pages 197–212. Springer, 2014.

[5] G. Aluç, M. T. Özsu, and K. Daudjee. Workload matters: Why RDF databases need a new design. *PVLDB*, 7(10):837–840, 2014.

[6] R. Angles and C. Gutiérrez. Querying RDF data from a graph database perspective. In *ESWC*, 2005.

[7] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data. In *WWW*, 2010.

[8] D. J. Beckett. The design and implementation of the redland RDF application framework. In *World Wide Web Conference Series*, 2001.

[9] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia-a crystallization point for the web of data. *Web Semantics: science, services and agents on the world wide web*, 7(3):154–165, 2009.

[10] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1247–1250. ACM, 2008.

[11] V. Bonstrom, A. Hinze, and H. Schweppe. Storing RDF as a graph. In *Web Congress, 2003. Proceedings. First Latin American*. IEEE, 2003.

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

[13] J. Broekstra, A. Kampman, and F. V. Harmelen. *Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema*. 2002.

[14] D. Bursztyn, F. Goasdoué, and I. Manolescu. Optimizing reformulation-based query answering in RDF. In *EDBT: 18th International Conference on Extending Database Technology*, 2015.

[15] D. Bursztyn, F. Goasdoué, and I. Manolescu. Reformulation-based query answering in RDF: alternatives and performance. *PVLDB*, 8(12):1888–1891, 2015.

[16] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. *PVLDB*, 1(1):1216–1227, 2005.

[17] `https://github.com/Quetzal-RDF/quetzal`, 2017-06-17.

[18] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 145–156. ACM, 2011.

[19] O. Erling and I. Mikhailov. *Virtuoso: RDF support in a native RDBMS*. Springer, 2010.

[20] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. View selection in semantic web databases. *PVLDB*, 5(2):97–108, 2011.

[21] S. Gottipati and J. Jiang. Linking entities to a knowledge base with query expansion. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 804–813. Association for Computational Linguistics, 2011.

[22] `https://github.com/Caesar11/gStore`, 2017-07-31.

[23] A. Gubichev and T. Neumann. Exploiting the query structure for efficient join ordering in SPARQL queries. In *EDBT*, volume 14, pages 439–450, 2014.

[24] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2), 2005.

[25] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 289–300. ACM, 2014.

[26] M. Hammoud, D. A. Rabbou, R. Nouri, S.-M.-R. Beheshti, and S. Sakr. DREAM: distributed RDF engine with adaptive query planner and minimal communication. *PVLDB*, 8(6):654–665, 2015.

[27] X. Han, L. Sun, and J. Zhao. Collective entity linking in web text: a graph-based method. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 765–774. ACM, 2011.

[28] S. Harris and N. Gibbins. 3store: Efficient Bulk RDF Storage. In *PSSS*, 2003.

[29] S. Harris, N. Lamb, and N. Shadbolt. 4store: The design and implementation of a clustered RDF store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, 2009.

[30] A. Harth and S. Decker. Optimized Index Structures for Querying RDF from the Web. In *Latin American Web Congress*, 2005.

[31] J. Hayes and C. Gutierrez. Bipartite graphs as intermediate model for RDF. In *ISWC*, 2004.

[32] J. Hu, G. Wang, F. Lochovsky, J.-t. Sun, and Z. Chen. Understanding user's query intent with Wikipedia. In *Proceedings of the 18th international conference on World wide web*, pages 471–480. ACM, 2009.

[33] J. Kim, H. Shin, W.-S. Han, S. Hong, and H. Chafi. Taming subgraph isomorphism for RDF query processing. *PVLDB*, 8(11):1238–1249, 2015.

[34] `http://lod-cloud.net/`.

[35] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01), 2007.

[36] A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura. A path-based relational RDF database. In *Proceedings of the 16th Australasian database conference-Volume 39*. Australian Computer Society, Inc., 2005.

[37] A. Matono, S. M. Pahlevi, and I. Kojima. *RDFCube: A P2P-Based Three-Dimensional Index for Structural Joins on Distributed Triple Stores*. 2006.

[38] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *2011 IEEE 27th International Conference on Data Engineering*, pages 984–994. IEEE, 2011.

[39] T. Neumann and G. Weikum. RDF-3X: A RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.

[40] M.-D. Pham. Self-organizing structured RDF in MonetDB. In *Data Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on*, pages 310–313. IEEE, 2013.

[41] `https://www.w3.org/TR/rdf11-concepts/#section-Graph-Literal`.

[42] B. Shao, H. Wang, and Y. Li. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD '13*, pages 505–516, New York, NY, USA. ACM.

[43] `https://www.w3.org/TR/rdf-sparql-query/`.

[44] `https://github.com/SoulSight/Stylus`.

[45] J. Subercaze, C. Gravier, J. Chevalier, and F. Laforest. Inferray: fast in-memory RDF inference. *PVLDB*, 9(6):468–479, 2016.

[46] `https://github.com/Microsoft/GraphEngine`.

[47] `http://dsg.uwaterloo.ca/watdiv/`.

[48] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[49] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *International Semantic Web Conference*, 2003.

[50] D. Wood, P. Gearon, and T. Adams. Kowari: A platform for semantic web storage and analysis. In *XTech 2005 Conference*, 2005.

[51] R. Xie, Z. Liu, J. Jia, H. Luan, and M. Sun. Representation learning of knowledge graphs with entity descriptions. 2016.

[52] `http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/`, 2017.

[53] P. Yin, N. Duan, B. Kao, J. Bao, and M. Zhou. Answering questions with complex semantic constraints on open knowledge bases. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 1301–1310. ACM, 2015.

[54] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. TripleBit: A fast and compact system for large scale RDF data. *PVLDB*, 6(7):517–528, 2013.

[55] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. *PVLDB*, 6(4):265–276, 2013.

[56] Z. Zheng, F. Li, M. Huang, and X. Zhu. Learning to link entities with knowledge base. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 483–491. Association for Computational Linguistics, 2010.

[57] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: Answering SPARQL queries via subgraph matching. *PVLDB*, 4(8):482–493, 2011.