Efficient Distributed Query Processing for Autonomous RDF Databases

Fabian Prasser*† prasser@in.tum.de
Alfons Kemper* kemper@in.tum.de
Klaus A. Kuhn† klaus.kuhn@lrz.tum.de

*Technische Universität München
Department of Computer Science
85748 Garching, Germany

†Technische Universität München
University Hospital (Klinikum rechts der Isar)
81675 München, Germany

ABSTRACT

The inherent flexibility of the RDF data model has led to its notable adoption in many domains, especially in the area of life-sciences. Some of these domains have an emerging need to access data integrated from various distributed sources of information. It is not always possible to implement this by simply loading all data into one central RDF store. For example, in the context of inter-institutional collaboration for drug development and clinical research participants often want to maintain control over their local databases. Alternatively, distributed query processing techniques can be utilized to evaluate queries by accessing the remote data sources only on demand and in conformance with local authorization models. In this paper we present an efficient approach to distributed query processing for large autonomous RDF databases. The groundwork is laid by a comprehensive RDF-specific schema- and instance-level synopsis. We present an optimizer that is able to utilize this synopsis to generate compact execution plans by precisely determining, at compile-time, those sources that are relevant to a query. Furthermore we present a tightly integrated query engine that is able to further reduce the volume of intermediate results at run-time. An extensive evaluation shows that our approach improves query execution times by up to two and transferred data volumes by up to three orders of magnitude compared to a naïve implementation.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Distributed databases*; H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Algorithms, Performance

Keywords

RDF, SPARQL, Distributed Query Processing

EDBT 2012, March 26–30, 2012, Berlin, Germany.

1. INTRODUCTION

1.1 Background and Motivation

The Resource Description Framework (RDF) data model offers flexible means to collect data without the need to explicitly specify a database schema. An RDF dataset is defined by a set of RDF *Triples*. Each triple consists of a *Subject*, a *Predicate* and an *Object*, expressing that the "subject" has the property "predicate" with value "object". *Resources* are named by globally unique *Uniform Resource Identifiers* (URIs) which are a proper superset of URLs. Subjects and predicates are always resources whereas objects are either resources or literals. *Literals* are atomic values with optional type or language information.

Propofol	state	„Liquid“ .
Propofol	formula	„$C_{12}H_{18}O$“ .
Propofol	sideEffect	Apnea .
Apnea	label	„Apnea“ .
Propofol	contraindication	Hypovolemia .
Hypovolemia	label	„Hypovolemia“ .

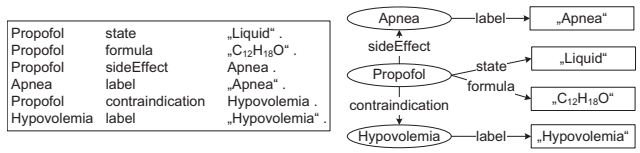

Figure 1: Example RDF dataset and graph[1]

A set of RDF Triples (e.g., stored in one single table with three columns) can also be seen as a directed, labeled graph. The graph can be derived by interpreting each triple as an edge labeled by the predicate that reaches from the subject to the object [3]. As our target discipline is biomedical research, our examples are drawn from this domain. A dataset describing drugs and their side-effects as well as the resulting RDF Graph is shown in Figure 1.

```
SELECT ?compound ?formula ?sideeffect
WHERE {
  ?compound   contraindication   Hypovolemia .
  ?compound   formula             ?formula .
  ?compound   sideEffect          ?sideeffect .
}
```

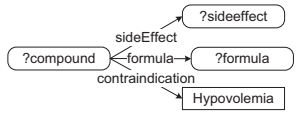

Figure 2: Example SPARQL query[1]

The default query language for RDF, SPARQL, is centered around the concept of pattern matching. A basic SPARQL query is constructed by combining a set of *Triple Patterns* to form a *Basic Graph Pattern* (BGP). Each triple pattern is an RDF Triple in which the subject, predicate or object can be replaced by variables. Joins between triple patterns are defined implicitly by using the same variable names [6]. When evaluating a query, an RDF database returns all possible substitutions for the query's variables such that the resulting pattern is contained in the queried RDF

[1]URIs have been abbreviated for better readability

graph. Figure 2 shows a SPARQL query against our example dataset as well as the graph pattern contained. It asks for the name, chemical formula and possible side-effects of all drugs with contraindication "Hypovolemia". The returned variable binding is: `?compound=Propofol, ?formula=` "$C_{12}H_{18}O$", `?sideeffect=Apnea`.

Because RDF data is modeled as a network of objects with well-defined semantics it is supposed to be well suited for the canonical representation of heterogeneous, disparate data sources and structures. In order to foster interoperability, common vocabularies (e.g., RDFS [4], OWL [2]) have been developed that provide further means to describe the semantics and structure of an RDF graph. The fact that these vocabularies are themselves expressed in RDF enables consistent management of data and metadata. The flexibility of RDF simplifies the management of arbitrary graph structures, which has led to its notable adoption in the area of life sciences (e.g., [10]). In a project at one of our university hospitals we are developing innovative methods to integrate and manage distributed, heterogeneous information sources for biomedical research. We currently focus on RDF and related technologies as the described properties render them interesting for an adoption in this domain.

Due to its schema-relaxed nature the efficient management of large amounts of RDF data is a challenging task. It has therefore attracted attention from the database community during recent years which has led to the development of highly scalable RDF stores [7, 16, 22, 23, 25]. But in the context of biomedical research it is often not possible to load the entire data into a central repository due to legal and regulatory requirements or retentions caused by issues regarding intellectual property rights. This is especially important within the scope of inter-institutional collaboration where participants often want to maintain control over their local databases. Therefore distributed query processing techniques [17] are often implemented that allow to evaluate queries by accessing the remote data sources only on demand. Because data can thereby be kept at its origin it is much easier to preserve access autonomies, e.g., by incorporating local authorization models. Furthermore techniques for querying distributed RDF repositories are also relevant in other areas such as query processing for Linked Data [1].

1.2 Related Work and Contribution

So far, only few generic concepts for querying distributed RDF databases have been proposed. Specialized systems implement approaches oriented towards multidatabase languages (e.g., [11]), the Semantic Web (a large number of small data sources, e.g., [21]) or Linked Data (a semantic web in which resources have dereferencable URIs, e.g., [15, 18]). Most of the more generic solutions implement a mediator/wrapper architecture as shown in Figure 3. With the exception of [9], which uses an external search engine, the mediator normally maintains a global synopsis which is implemented by some sort of index structure. This index is used to decompose queries into local subqueries which are then processed by wrappers that harmonize and extend the interfaces of the remote systems. The approaches presented in [27] and [24] are based on indexing the data sources on schema level (RDF predicates). As these systems decide which parts of a query have to be evaluated at which endpoint based on the predicates of the query's triple patterns, they are limited to queries with bound predicates. Although at least [24] stores further (manually defined) statistics, query optimization is very difficult due to the lack of comprehensive instance-level information. In [19] the authors describe a system which uses histograms on instance-level to perform various query optimization techniques. As these histograms are built for instances of classes and their properties, data sources have to be annotated with and adhere to an explicit schema definition. This contradicts the schema-free nature of RDF. In [14] an optimizer has been presented which overcomes these limitations by utilizing a combined schema- and instance-level index, referred to as *RDF Data Summaries*. It is oriented towards a Linked Data scenario and is able to optimize arbitrary queries (i.e., with unbound predicates) over arbitrary datasets (i.e., without schema information). An index is built by independently hashing each of the subject, predicate and object of the triples contained in the datasets. The resulting three-dimensional points are then approximated by *Minimum Bounding Boxes* (MBBs) each of which summarizes a certain set of triples. When optimizing a query, constants contained in the triple patterns are hashed and the resulting range-queries are issued against the index. As a result for each triple pattern these queries return a set of MBBs. Dependencies between triple patterns are taken into account by executing the same operations (e.g., joins) on the returned MBBs that would have been performed on variable bindings during query execution. The set of relevant sources is then extracted from source identifiers contained in the MBBs (see section 2).

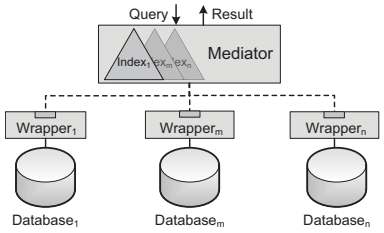

Figure 3: Common architecture with global synopsis

Accurate data localization is a major challenge when querying a distributed RDF graph because multiple sources are often able to answer a contained triple pattern although they are not able to contribute to the overall result of a query. This is due to the schema-relaxed nature of SPARQL and the widespread use of some vocabularies. RDF Data Summaries provide means to identify relevant sources more accurately than previous work, but have some limitations. As any type information is lost and triples are only approximated by ranges of hash values, the approach can only handle simple queries and smaller datasets due to significant main memory requirements and running times. In this work we adopt this concept in order to implement efficient distributed query processing over large autonomous RDF databases. This includes the following scientific contributions: *i)* a comprehensive RDF-specific synopsis that allows the optimizer to efficiently and accurately determine, at compile-time, sources that are relevant for answering a query, *ii)* a tightly integrated proof-of-concept query engine that is able to further prune irrelevant variable bindings at runtime, *iii)* an extensive evaluation based on a comprehensive workload and a large distributed RDF graph consisting of several real-world biomedical knowledge bases, *iv)* a performance comparison of our approach for three different types

of data distribution that model different ways in which we expect data to be collected. The experiments show that our approach improves query execution times by up to two and transferred data volumes by up to three orders of magnitude compared to a naïve implementation.

The rest of this paper is structured as follows: Section 2 describes the basic idea and presents in-depth information on the index structure developed, whereas section 3 covers query optimization. An approach to query execution that is tightly integrated with the optimizer is described in section 4. The system is evaluated in section 5 and finally in section 6 results and directions for future work are discussed.

2. INDEXING

The basic idea of an RDF Data Summary is to project the data onto a set of three-dimensional points by computing independent hash values for each triple's subject, predicate and object. These points are then approximated by a spatial index structure called Q-Tree [14]. Q-Trees are a variant of R-Trees that have a constant number of leaf nodes and do not reference the data items themselves but summarize them, e.g., by storing counts. Data items are approximated by *Minimum Bounding Boxes* (MBBs) on leaf level. A set of nodes is itself summarized by a MBB on a higher level, up to one single root node. Figure 4 shows an example of a two-dimensional Q-Tree.

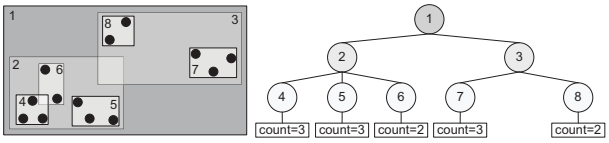

Figure 4: A two-dimensional Q-Tree

A data summary is a three-dimensional Q-Tree for all data sources, which is built by iteratively transforming the triples from each source and inserting them into the tree. Each leaf node stores a set of source identifiers, including one for each source of a triple approximated by the node. During query optimization a range-query is built from each triple pattern's constants and variables and executed against the index. The result of such a query is a set of MBBs derived from valid leaf nodes. If the result is not empty, every system that is referenced by any of the resulting MBBs could potentially return variable bindings for the according triple pattern.

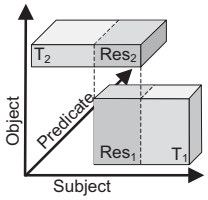

Figure 5: Joining two MBBs

The dependencies between individual triple patterns are taken into account by joining the MBBs as defined in the query. In doing so only those sources remain relevant that are referenced in any MBB resulting from applying all operators (e.g., joins). An example for a join between two MBBs (T_1 and T_2) over the subject dimension (resulting in Res_1 and Res_2) is shown in Figure 5. For more details the interested reader is referred to [14].

Although this approach is highly flexible, it is not well suited for complex queries and large datasets. As the nodes of the indexed RDF datasets are hashed, any type information is lost. However, the preservation and incorporation of type information would lead to more accurate results (e.g., resources can only be joined with resources) and the ability to consider further SPARQL operators (e.g., selecting results with filter expressions). Furthermore it is very inaccurate to approximate several triples by ranges of hash values, as this often leads to overlapping MBBs that do not approximate common RDF nodes. This is reinforced by the fact that Q-Trees are designed to have a predefined, constant number of leaf nodes regardless of the size of the dataset. In general there are therefore considerably more join partners for MBBs than there would be for the approximated triples. This leads to inaccurate results and limits the optimizer to simple queries and small datasets due to significant main memory requirements and running times. In the following sections we will present some extensions and modifications that help to overcome these limitations.

2.1 Type Information

By preserving type information and implementing value- or order-preserving transformations for some literal data types, the optimizer is able to produce more accurate results and consider further operators during query optimization. We represent type information by two-byte integers and are therefore able to handle 65536 different data types. Some identifiers are reserved for common data types of literals (such as integers, strings or calendar dates). As we use eight-byte hash values, literals with a fixed-length representation (such as integers) can be kept as is whereas other values need to be transformed by applying a hash function.

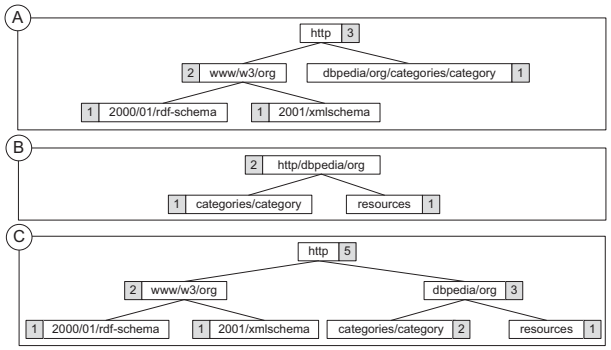

Figure 6: Local and global prefix trees

Type information is not only preserved for literals but also extended to cover resources by encoding a prefix of the resources' URIs. In contrary to literals, type identifiers can not be assigned statically to prefixes as there might be too many of them (e.g., a heterogeneous collection of URLs). However, as even datasets that do not adhere to a specific schema very often preferably use certain namespaces, most URIs in an RDF dataset share a set of *common prefixes*. To determine the common prefixes of an individual dataset we normalize the triples' URIs and split them into a list of path components. We then add all components (excluding the last element) to a radix tree and count the number of their occurrences (trees (a) and (b) in Figure 6). Each leaf node represents one prefix which can be built by appending all components on its path to the root node. Finally, a global view is needed as this allows to map the same prefixes from different datasets onto the same type identifiers. Local radix trees are computed in parallel for each data source

(trees (a) and (b) in Figure 6) and merged into a global tree (see tree (c) in Figure 6). If the number of prefixes is larger than what can be encoded by the available two bytes, the top-k prefixes can be selected by iteratively removing the leaf node with the lowest frequency until only k leaf nodes remain. The organization of common prefixes in a radix tree also allows to efficiently determine the type for a given normalized URI as radix trees support an efficient longest-prefix matching operation. Parts of the URI not contained in the longest prefix are transformed by applying a hash function. To cover unknown or infrequent types we further reserve default identifiers. In the following sections we assume that the function $type(n)$ returns a two-byte type identifier and $hash(n)$ returns an eight-byte hash value for any resource or literal n.

2.2 Index Organization

The proposed index structure does not implement a single synopsis for all datasets, but one index per data source. This allows for independent updates and increases the accuracy as any MBB can be uniquely associated with a single endpoint. Furthermore we apply vertical partitioning [7] to each index. This concept is based on the observation that for real-world (e.g., biomedical) RDF datasets the number of distinct predicates is very small compared to the number of triples. We therefore group triples that share the same predicate into a common partition, efficiently eliminating the need to store most of the predicates. This enables us to reduce the amount of redundant information and at the same time preserve the predicates' hash values. This further increases the accuracy on schema-level, which is important because predicates are rarely unbound in common SPARQL queries [8]. Furthermore vertical partitioning allows to efficiently handle type information. For this purpose we partition the dataset not only by predicate hash value, but also by subject, predicate and object type. As a result, a partition $P = (type_s, type_p, type_o, hash_p)$ references a two-dimensional spatial index structure, approximating subject and object hash values for any triple $t = (t_s, t_o, t_p)$ with $type(t_s) = type_s$, $type(t_p) = type_p$, $hash(t_p) = hash_p$ and $type(t_o) = type_o$.

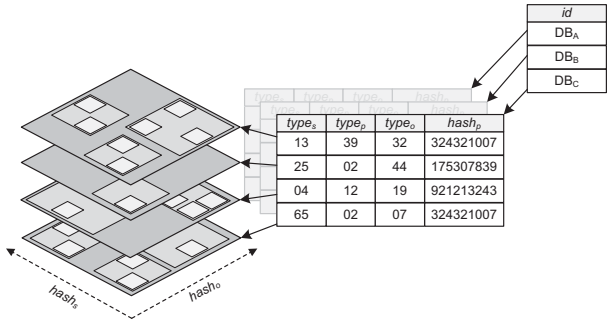

Figure 7: PARTrees, partition- and system tables

An example for such an index is show in Figure 7. The system table references a list of partitions (partition table) for each indexed data source. Each of these partitions again references a spatial index structure that will be explained in more detail in the following section.

2.3 Partition-Trees

In order to further increase the accuracy of our index we developed an extension of Q-Trees, which we call Partition-Trees or PARTrees for short, that store more comprehensive information about the data elements in one partition. Conceptually, sets of integers are added to the leaf nodes which store the two least significant bytes (LSBs) of the indexed hash values. E.g., for the subject-dimension the set b_s contains the value $(hash(t_s) \mod 2^{16})$ for each triple $t = (t_s, t_p, t_o)$ approximated by the leaf node. Resulting in only a tiny space overhead, it is further possible to preserve the correlation between the LSBs of the hash values for the different dimensions. For this purpose we store a set of points $n.lsb = \{(hash(t_s) \mod 2^{16}, hash(t_o) \mod 2^{16})\}$ that approximate the subject and object hash value of each triple $t = (t_s, t_p, t_o)$ indexed by a leaf node n. Because each of these points occupies two bytes for the subject and object dimension respectively, the index consumes at least four bytes of memory per triple. As can be seen in the following sections this growth in memory consumption is legitimate because the stored information helps to determine sources for triple patterns more accurately and drastically reduces the number of join partners for most MBBs. In contrast to Q-Trees, which have a fixed number of leaf nodes, we allow our trees to grow with the number of indexed data elements. In order to control space consumption we define a maximum fanout for inner nodes and leaf nodes which describes the maximum number of child nodes (or approximated triples) per node. During indexing the PARTrees are generated locally by applying the Sort-Tile-Recursive (STR) bulk-loading algorithm [20] and transmitted to the mediator on a per-partition basis where they are kept in main memory.

3. QUERY OPTIMIZATION

Query optimization starts with creating an *initial execution plan*. This step includes the process of selecting potential sources for the individual triple patterns in a query. Afterwards the query is *simplified* by pruning parts of the query execution plan that have been proven to not contribute to the final result. In a *postprocessing* step the plan is reorganized and subqueries are generated.

3.1 Initial Plan Generation

The initial execution plan is generated by selecting possible sources for each triple pattern. To this end type information and hash values are derived from the constants contained in the patterns. The global index is now queried in order to determine potential sources. A partition $p = (type_s, type_p, type_o, hash_p)$ is valid for a triple pattern $t = (t_s, t_p, t_o)$ if its referenced PARTree might approximate a triple that satisfies t. This means that the types and hash value defined by the partition p match the triple pattern's constants, i.e., all of the following conditions hold true:

1. t_s is a variable, or $type(t_s) = p.type_s$,
2. t_p is a variable, or ($type(t_p) = p.type_p$ and $hash(t_p) = p.hash_p$),
3. t_o is a variable, or $type(t_o) = p.type_o$.

For each system the partition table is checked and range- or point-queries are executed against the PARTrees referenced by valid partitions. Similar to R-Trees, the trees are traversed from the root node to the leaf nodes on each path consisting of valid nodes n with $n.min_d \leq hash(t_d) \leq n.max_d$

for each constant dimension $d \in \{s, o\}$. A leaf node $n_l = (min_s, max_s, min_o, max_o, lsb)$ is valid if its MBB also covers $hash(t_d)$ for each constant dimension $d \in \{s, o\}$ and the stored LSBs match the hash value(s), i.e., the following holds true:

1. $t.s$ and $t.o$ are variables, or
2. $t.o$ is variable and $(hash(t.s) \mod 2^{16}, y) \in n_l.lsb$, or
3. $t.s$ is variable and $(x, hash(t.o) \mod 2^{16}) \in n_l.lsb$, or
4. $(hash(t.s) \mod 2^{16}, hash(t.o) \mod 2^{16}) \in n_l.lsb$.

Based on these preconditions it is possible to determine the set of potential sources for each triple pattern from the global index which is sketched in Algorithm 1. If the optimizer is not able to determine at least one potential source for each of the query's triple patterns, the query can not yield any results and further optimization and query execution can be omitted.

Algorithm 1: SOURCE SELECTION

```
Input: Triple pattern t
Result: All possible sources for t
1 begin
2     S ← ∅
3     foreach system s do
4         foreach partition p of s that is valid for t do
5             if tree of p contains valid leaf node for t then
6                 S ← S ∪ {s}
7                 break
8     return S
```

For our example query from Figure 2 sources for the triple pattern `(?compound, formula, ?formula)` can be found by only checking the partition tables as both subject and object are variables. For the triple pattern `(?compound, contraindication, "Hypovolemia")` additional range queries with a constant at the object dimension have to be performed.

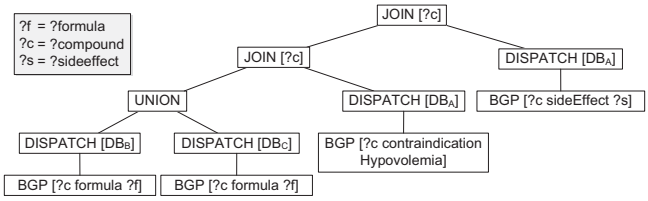

Figure 8: Initial plan for the example query

An initial execution plan is built from the set of potential sources by modeling dependencies between patterns as joins over its variables. The operator $Join(V_j)$ denotes that the variable bindings returned by its children are to be joined over variables V_j. An operator $Dispatch(s)$ states that its subtree is to be evaluated by the remote system s. The case in which there is more than one source for solutions to a triple pattern is modeled by unifying the results of different dispatch operators. The operator $Union$ therefore denotes that the variable bindings returned by its children are to be unified. The initial execution plan for our example query is shown in Figure 8. We assume a scenario consisting of three data sources, DB_A, DB_B and DB_C in which the triple pattern `(?compound, formula, ?formula)` can be answered by the systems DB_B and DB_C whereas the other two triple patterns can only be answered by the system DB_A.

3.2 Plan Simplification

In the next optimization phase the initial execution plan is simplified by "simulating" its execution based on the information provided by the index structure. The variable bindings returned by evaluating a triple pattern at a remote system are represented by *Approximate Variable Bindings* (AVBs). Each AVB b approximates a set of bindings for a set of variables V by defining type information, boundaries for hash values and compressed bitsets for each variable dimension $v \in V$.

1. $b.min_v$ and $b.max_v$ are lower and upper bounds for the bindings' hash values.
2. $b.type_v$ defines the bindings' type.
3. $b.bitset_v$ encodes the bindings' LSBs.

The set of possible LSBs for the approximated variable bindings are encoded in a bitset (of length 2^{16}) because this allows to efficiently join two AVBs. As the bitsets are only sparsely populated they can also be compressed very well. Each AVB is initially being produced by a certain dispatch operator as explained below. When an AVB has been completely processed and reaches the top of the query execution plan there are in general multiple dispatch operators that have contributed to it. Each AVB stores a bitset $b.dispatches$ that is used to keep track of these contributing operators. In the following sections we will describe these operators in more detail. As a $Union$ operator simply unifies the two sets of AVBs produced by its children we focus on the $Dispatch$ and $Join$ operator.

Algorithm 2: PRODUCE AVBS

```
Input: System s, triple pattern t = (t_s, t_p, t_o)
Result: All AVBs for t from system s
1  begin
2      B ← ∅
3      foreach partition p of s that is valid for t do
4          foreach leaf node n in tree of p that is valid for t do
5              b ← new AVB
6              foreach d ∈ {s, p, o} such that t_d is variable do
7                  if d = p then
8                      b.type_{t_d} = p.type_p
9                      b.min_{t_d} = b.max_{t_d} = p.hash_p
10                     b.bitset_{t_d}.set(p.hash_p mod 2^16)
11                 else
12                     b.type_{t_d} = p.type_d
13                     b.min_{t_d} = n.min_d, b.max_{t_d} = n.max_d
14                     b.bitset_{t_d} = DERIVEBITSET(d, n, t)
15             B ← B ∪ {b}
16     return B
```

3.2.1 Dispatch Operator

When executed, the triple pattern referenced by a dispatch operator $Dispatch(s)$ is evaluated in the same way as during source selection but for a single source s. Similar to the initial plan generation process the partition table of s is checked and the referenced PARTrees are queried for valid leaf nodes. The resulting leaf nodes are then converted into AVBs representing sets of potential bindings for a triple pattern. This process is sketched in Algorithm 2. For each variable dimension boundaries for hash values as well as the data type and a bitset is defined. If the predicate is a variable this information is derived from the partition's values

for $type_p$ and $hash_p$. In case of a variable at the subject or object position the minimum and maximum boundaries for the bindings' hash values are defined by the leaf node's MBB and the type is defined by the partition.

Bitsets for the subject and object dimension can be derived directly from the set of points $n.lsb$ if both subject and object are variables. Otherwise the hash value of the constant dimension is used to project onto relevant bits for the variable dimension. If both, subject and object, are variables each dimension is transformed into a bitset by simply setting the bits as defined by the points' coordinates for the subject and object dimension. In case of one constant and one variable dimension only these bits are set for the variable dimension for which a point exists in $n.lsb$ that has a coordinate matching the constant dimension. Algorithm 3 implements this for a variable subject dimension, the object dimension is handled analogously.

Algorithm 3: DERIVE BITSET

```
Input: Dimension d ∈ {s, o}, leaf node n, triple pattern
       t = (t_s, t_p, t_o)
Result: Bitset for the dimension d
1  begin
2    b ← new bitset
3    if d = s then
4      /* handle subject dimension*/
5      if t.o is a variable then
6        foreach (x, y) ∈ n.lsb do
7          b.set(x)
8      else
9        foreach (x, hash(t_o) mod 2^16) ∈ n.lsb do
10         b.set(x)
11   else
12     /* handle object dimension*/
13   return b
```

The downside of this transformation is that the correlation between the LSBs is lost if both subject and object are variables, resulting in possible false positives. On the other hand AVBs can now be joined much more efficiently, as the intersection of two bitsets can be performed simply by computing a bitwise AND-operation. Ascending integers are assigned as unique identifiers to each dispatch operator in order to enable the tracking of contributing sources. To this end the i-th bit is initially set in $b.dispatches$ for each AVB b produced by a dispatch operator with identifier i.

3.2.2 Join Operator

Two MBBs can be joined if the defined ranges overlap for each join dimension. For AVBs we extend this by taking data types and bitsets into account. Two AVBs b_1 and b_2 can be joined if the following holds true for each join dimension $v \in V_j$:

- $b_1.min_v \leq b_2.max_v$ and $b_1.max_v \geq b_2.min_v$ (MBBs overlap),
- $b_1.type_v = b_2.type_v$ (types are equal),
- $(b_1.bitset_v$ AND $b_2.bitset_v) \neq 0$ (at least one equal bit is active in the bitsets).

The minimum and maximum boundaries for hash values resulting from a join between two AVBs are defined in the same way as for a spatial join of the MBBs involved. Furthermore the bitsets of the resulting AVB are defined as the result of a bitwise AND-operation on the bitsets of both AVBs for each dimension. Therefore the number of active bits in the AVBs' bitsets decreases as AVBs are passed upwards towards the root of the query execution plan. The properties of the AVB b_r resulting from a join between two AVBs b_1 and b_2 for each dimension $v \in V_j$ are defined as:

- $b_r.min_v = max(b_1.min_v, b_2.min_v)$,
- $b_r.max_v = min(b_1.max_v, b_2.max_v)$,
- $b_r.bitset_v = b_1.bitset_v$ AND $b_2.bitset_v$,
- $b_r.type_v = b_1.type_v$.

The bitset $b_r.dispatches$ is set to ($b_1.dispatches$ OR $b_2.dispatches$) in order to keep track of the sources contributing to b_r. Properties for variable dimensions not contained in V_j are simply inherited from either b_1 or b_2. Note that it is not possible that the underlying triple patterns contain a common variable that is not in V_j due to the implicit definition of joins. When joining two sets of MBBs we first partition each operand into sets of AVBs with equal data types in the join dimensions. We then solve the rectangle intersection problem for each combination of compatible partitions by applying a standard plane-sweeping algorithm. Afterwards the bitsets of each pair of intersecting AVBs are checked for compatibility.

3.2.3 Join-order Optimization

It is obvious that the performance of joining sets of AVBs strongly depends on a good join-order for many queries. We therefore implemented a cost-based optimizer utilizing a top-down enumeration strategy with memoization [12]. We assume independence between the individual triple patterns and estimate selectivities based on the higher-level inner nodes of the PARTrees. For a given triple pattern we construct a set of MBBs by traversing the PARTrees of each valid partition up to a predefined depth (e.g., three). We estimate the selectivity for a join between two triple patterns by simply computing the cardinality of the spatial join between the sets of MBBs returned by this scans. As LSBs are only stored in the leaf nodes we can in this process only consider the MBBs' and the data types defined by the partitions. This simple procedure has some drawbacks. Firstly, as the PARTrees' higher-level nodes approximate a potentially large number of triples from which only a tiny fraction might be a valid solution the resulting selectivity estimation can be very imprecise. Secondly, some triple patterns have a large number of valid higher-level nodes which leads to a non-negligible running time overhead. Thirdly, the assumption of independence of the selectivities of joins in a SPARQL query does not hold as has been shown in [22]. Nevertheless the described method performs reasonably well for most queries of medium complexity as can be seen in section 5.

3.3 Postprocessing

In the postprocessing phase the resulting operator tree is transformed and subqueries are derived. If the set of resulting AVBs is empty the query can not return any results and query execution can therefore be omitted. Otherwise there is a potential to prune redundant dispatch-operators that have not contributed to the result. To this end we monitor

the resulting AVBs and keep only those operators who have contributed to at least one AVB. The others are pruned from the query execution plan.

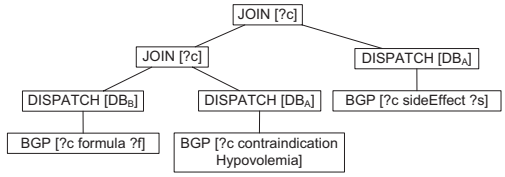

Figure 9: Simplified query execution plan

For our example query from Figure 2 we assume that the results for the triple pattern `(?compound, formula, ?formula)` from source DB_C did not contribute to the overall result. Figure 9 shows the query execution plan after pruning the redundant operator.

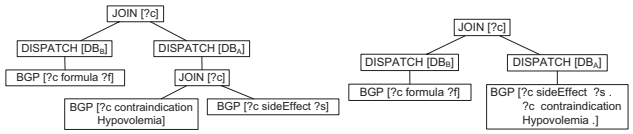

Figure 10: Reorganization and pushdown

Afterwards the tree is reorganized in order to push down joins between triple patterns behind one common dispatch operator whenever possible. The triple patterns of these operators can then be unified to form a more complex basic graph pattern. As in [16] we rely on a heuristic assuming that the minimal number of subqueries yields optimal performance. The Figures 10 and 11 show the transformed execution plan as well as the derived subqueries for our example query.

DB_A

```
SELECT ?compound ?sideeffect
WHERE {
  ?compound   sideEffect  ?sideeffect .
  ?compound   contraindication Hypovolemia
}
```

DB_B

```
SELECT ?compound ?formula
WHERE {
  ?compound   formula     ?formula
}
```

Figure 11: Subqueries for the example query

4. QUERY EXECUTION

In order to evaluate the pruning power of the presented optimizations we have developed a query engine based on the mediator/wrapper architecture shown in Figure 3. It implements a simple query execution model which consists of four steps:

1. Parse and optimize the query.
2. Execute subqueries at the remote systems.
3. Load local results into a global database.
4. Execute the query on the global database.

This is basically equivalent to executing the query on a relevant subgraph of the global graph. Therefore an RDF store can again be used as a global temporary database.

An important side-effect of the presented optimization technique is that the AVBs resulting from query optimization can also be used to prune redundant variable bindings during query execution. As AVBs describe restrictions for hash values of variable bindings they can be used to filter a stream of bindings, leaving only those who are potentially relevant for answering a query. A set of n-dimensional AVBs can be derived from a set of bindings B for n variables by iteratively hashing the values of each binding and approximating them in the same way as during index generation, i.e., by applying the Sort-Tile-Recursive algorithm. When we denote this by a function $AVB(B)$ and returning bindings for a query execution plan P by a function $Q(P)$ the following holds for a query consisting of a join between two triple patterns T_1 and T_2:

$$AVB(Q(T_1 \bowtie T_2)) \approx AVB(Q(T_1)) \bowtie AVB(Q(T_2))$$

As $AVB(Q(T_1)) \bowtie AVB(Q(T_2))$ is exactly what is computed by the optimizer during query simplification, the AVBs returned by the optimizer effectively approximate the variable bindings that would be returned when executing the query. This is true for any other query and can be used to implement a concept related to semi-join reducers (see, e.g., [17,26]). In this approach irrelevant join candidates for a join between two relations from remote systems are pruned by matching them against the results of a semi-join operation. Although there are different ways to implement semi-join reducers (e.g., with bloom-filters), they have in common that the reducers are computed at run-time whereas in our case they are generated at compile-time.

The optimizer returns a set of n-dimensional AVBs (one dimension for each variable in the query) but individual subqueries normally only cover parts of these variables. We therefore split each resulting AVB into multiple (potentially overlapping) AVBs, one for each set of variables contained in a subquery. As the set of AVBs for an individual subquery might now contain duplicates, we further apply a distinct operator. In order to enable efficient checking of variable bindings against the reducers, we again organize AVBs in PARTrees by applying the Sort-Tile-Recursive algorithm. In contrast to indexing where we build two-dimensional trees, the number of dimensions now depends on the number of variables in the subquery. Furthermore we do not partition the resulting AVBs by data types, in order to keep the space overhead low. Instead we organize them in one single PARTree in which nodes also store type information. For each dimension, the data type of a node is defined as the result of a bitwise OR-operation on the according type identifiers of its children. This is inspired by S-Trees, a variant of which has also been used for indexing RDF datasets in [28]. These trees are then transferred to the remote systems together with the subqueries. During the execution of a subquery, members of the stream of resulting variable bindings are pruned if there is no leaf node in the reducer matching the binding's types and hash values. This does not affect the completeness of the overall result of a query because the generated reducers only allow for false positives.

Our prototype implements all concepts described in this paper. It supports distributed indexing as well as query optimization and execution. Its system architecture is shown in Figure 12. Wrappers export standardized interfaces to the remote systems which are provided by instances of the RDF-3X database system [23]. As the wrappers are loosely coupled to the underlying databases and only require a SPARQL interface, we are able to integrate nearly any RDF store. We chose RDF-3X because it is one of the most efficient, open-source RDF database systems available and offers excellent performance. The components communicate via plain sockets and are multithreaded which enables them to process multiple queries in parallel.

The temporary global database is also implemented as an instance of the RDF-3X system. Because the developed optimizations generally cut down heavily on the number of intermediate results (see section 5) it is also possible to replace it by an in-memory database. Unfortunately the number of intermediate results is much higher without optimizations and they do thus often not fit into main memory. We therefore chose RDF-3X to allow for a fair comparison of the unoptimized and optimized case in the following section.

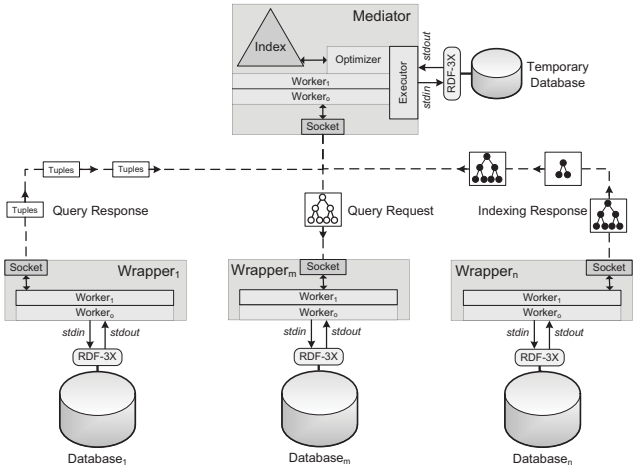

Figure 12: System architecture

5. EVALUATION

5.1 Datasets and Workload

Because there is no established benchmark for querying distributed RDF repositories, we generated a synthetic workload based on several biomedical knowledge bases.

5.1.1 Datasets

Our evaluation datasets contain roughly 100 million triples from different biomedical sources that are available in RDF format. We have built a scenario which models a knowledge base for drug developers and researchers in the area of medicine, incorporating information about proteins, diseases, genes, metabolic pathways, drugs, clinical trials and more. Diseasome [13] publishes a network of disorders and disease genes that have been obtained from Online Mendelian Inheritance in Man (OMIM). OMIM[1] is a compilation of human disease genes and phenotypes. Detailed chemical, pharmacological and pharmaceutical data is provided by DrugBank[2]. Dailymed[3] contains information about marketed drugs. Adverse effects are covered by SIDER[4]. LinkedCT[5] provides data about clinical trials that have been obtained from a public trial registry. DBpedia[6] consists of structured information derived from Wikipedia. The Bio2RDF [10] project publishes different datasets, containing a derivative of Entrez Gene[7] which is a database for gene-specific information from different projects[8]. An overview over the different datasets, the number of contained triples and the number of unique subjects, predicates and objects is shown in Table 1.

System	# Triples	# Subjects	# Predicates	# Objects
Infobox Properties[6]	34.2 M	1.816.862	38.563	8.107.107
Other Properties[6]	31.3 M	9.490.850	8	13.590.111
GeneID[7]	20.1 M	462.855	31	10.750.501
Linked CT[5]	9.8 M	981.880	90	3.808.369
HGNC [10]	1.1 M	125.256	37	655.833
OMIM[1]	0.9 M	20.280	43	379.099
Drugbank[2]	0.5 M	19.693	119	275.336
Dailymed[3]	0.2 M	10.015	28	67.778
Sider[4]	0.1 M	2.674	11	29.410
Diseasome [13]	0.1 M	8.152	19	27.704
Global	98.4 M	11.121.647	38.905	35.837.311

Table 1: Properties of the evaluation dataset

We evaluated three different scenarios by partitioning the dataset in different ways. In the *naturally-partitioned* scenario the datasets have been preserved as is. For the *horizontally-partitioned* scenario we merged all datasets into one single dataset and partitioned it horizontally into n equally sized datasets. In the *randomly-partitioned* scenario all triples have been distributed randomly among n equally sized datasets. These three scenarios model different ways in which we expect data to be collected. The rationale for natural data distribution is straightforward as this models the case in which several subject-specific data collections have been established. Horizontal data distribution resembles a scenario in which different knowledge bases have been built by merging subsets of other datasets. Finally the randomly-distributed scenario models an extreme for an RDF-specific type of data collection. We assume that users take advantage of their ability to uniquely reference entities in other datasets and further annotate them. Therefore the information regarding individual entities is spread over different data sources. A real-world data management solution for our biomedical use cases would probably have to deal with a data distribution lying somewhere in between these three scenarios.

5.1.2 Workload

The workload used in our evaluation has been generated from different patterns. A query pattern is defined by a number of stars (s), a number of constants (c) and a number of variables (v). Each star defines a basic graph pattern that consists of (c+v) triple patterns. Subjects of these triple patterns are always variables, predicates are always bound and c of the objects are constants, whereas v of the objects are variables. If a pattern consists of more than one star, the individual stars are connected via additional triple patterns with unbound subject and object and bound predicate. Therefore a query pattern with parameters s, c and v consists of $n = (c+v+1)*s-1$ triple patterns. Such queries are considered to be good representatives for many real world SPARQL queries [8] and similar query patterns have been used in other evaluations [7,16,23]. In the following sections we denote a pattern consisting of s stars with v variables and c constants as $S_sV_vC_c$. Figure 13 shows an overview over the set of query patterns used. It comprises patterns consisting of one, two and three stars, each of which consists of one, two or three constants and one or two variables. The most complex query pattern ($S_3V_1C_2$) consists of $n = 11$ triple patterns.

We created roughly 100 instances of each pattern. To this end we associated the stars to the data sources (naturally-partitioned). A random instance of a star was generated by

[1]http://www.ncbi.nlm.nih.gov/omim
[2]http://www.drugbank.ca/
[3]http://dailymed.nlm.nih.gov/
[4]http://sideeffects.embl.de/
[5]http://linkedct.org/about/
[6]http://dbpedia.org
[7]http://www.ncbi.nlm.nih.gov/gene
[8]We have extracted a subset of about 20M triples

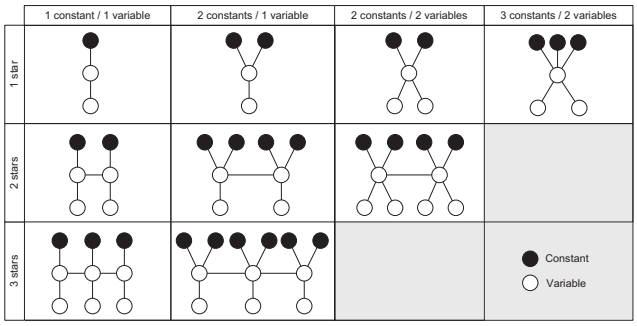

Figure 13: Query patterns

selecting all possible outgoing triples for a random subject, replacing the subject with a variable, randomly selecting v triples and replacing its objects by variables and c triples that were kept as is. In order to generate a workload which can be executed in a reasonable amount of time without applying any optimizations we ensured that each star does not yield more than 10.000 results. We further matched links between stars with links between datasets which allowed us to control how the load is spread among the data sources. For each possible link between two (or three) datasets we created an equal number of queries in a way that resulted in about 100 instances. In the case of $s = 1$ this was trivial. As there are 10 data sources we just created 10 random instances for each dataset. For patterns consisting of $s = 2$ stars we were able to discover 25 possible links between two data sources. Therefore we created four random instances per query pattern and link, which resulted in exactly 100 queries. For $s = 3$ this was more complicated as links between data sources are not always transitive. We found that there are 32 different possibilities for links between three datasets and created three instances for each of these links, resulting in 96 queries per pattern.

Query Class	$\leq 10^1$	$\leq 10^2$	$\leq 10^3$	$\leq 10^4$	$\leq 10^5$
$S_1V_1C_1$	71	5	6	11	7
$S_1V_1C_2$	88	8	1	2	1
$S_1V_2C_2$	82	7	7	3	1
$S_1V_2C_3$	87	9	3	1	0
$S_2V_1C_1$	87	6	4	3	0
$S_2V_1C_2$	98	1	1	0	0
$S_2V_2C_2$	91	6	2	1	0
$S_3V_1C_1$	73	20	3	0	0
$S_3V_1C_2$	88	5	3	0	0
Total	765	67	30	21	9

Table 2: Result cardinalities

Although there are no guarantees that this process generates queries that spread the load equally (i.e., query each dataset with the same frequency), it works well in practice. Regarding the naturally-partitioned scenario, the load is spread equally for $s = 1$. In case of $s = 2$ OMIM and GeneID are slightly underrepresented whereas Diseasome and Drugbank are overrepresented. For $s = 3$ OMIM, GeneID and HGNC are slightly underrepresented whereas Diseasome, Drugbank and Dailymed are overrepresented. All other datasets are queried with the same frequency. Table 2 shows the result cardinalities of the queries in our workload. Due to limiting the results of the individual stars to a cardinality of $\leq$10.000 most queries ($\approx$85%) return $\leq$10 results regardless of the underlying query pattern. As can be seen from the following experiments the overall time needed to execute a query is usually not strongly correlated with the cardinality of its result but with the cardinalities and number of its subqueries which are again influenced by data distribution.

5.2 Setup

Our experiments were performed on three Dell desktops which hosted the data sources and wrappers. Each of these machines has a 4-core 3.1 GHz Intel Core i5 CPU with 6 MB cache and 8 GB of memory running a 64-bit Linux kernel in version 2.6.35. The mediator was deployed on a Dell laptop with a 4-core 1.6 GHz Intel Core i7 CPU with 6 MB cache and 4 GB of memory also running a 64-bit Linux 2.6.35 kernel. All systems are able to perform sequential reads on their local hard disks with about 100 MB/s and were connected via Fast Ethernet. Mediator and wrappers are implemented in Java and all machines were running a 64-bit Sun JVM in version 1.6.0. The JVM heap size was restricted to 512 MB for each wrapper and 2 GB for the mediator. When partitioned naturally, we distributed the datasets among the three machines as follows:

- Machine A: Infobox Properties, OMIM, Drugbank ($\approx$36 M triples total)
- Machine B: Other Properties, HGNC, Sider ($\approx$32.6 M triples total)
- Machine C: GeneID, LinkedCT, Dailymed, Diseasome ($\approx$30.2 M triples total)

In case of horizontal or random partitioning we derived nine equally sized datasets ($\approx$10.9 M triples) and distributed them uniformly among the three machines (three datasets each). As it is more robust against outliers, we report the geometric mean of the running times for the different sets of queries.

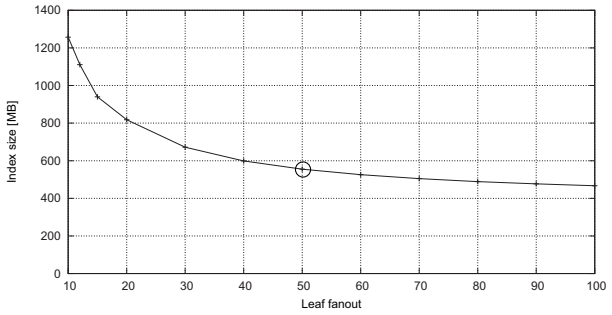

Figure 14: Index size in case of natural partitioning

5.3 Indexing

The size and accuracy of the index can basically be controlled by adjusting the PARTrees' leaf fanout (f_l) as it defines the number of triples that are packed into one leaf node. Figure 14 shows the size of the index for different values of f_l in the case of natural data distribution. Space consumption converges to a lower boundary when increasing f_l. This is due to the fixed four bytes allocated for each indexed triple. The additional overhead for nodes in the PARTrees decreases with f_l. For our evaluation we chose f_l to be 50 which seems to be a good trade-off between space consumption and performance according to our experiments. In this case the index consumes about 5.6 bytes per triple. This totals to about 555 MB of main memory which corresponds to a compression ratio of about 5% compared to the original data in RDF turtle format (12 GB). In case of horizontal and random data partitioning the space consumption increases slightly to 560 MB and 571 MB respectively. Building the index from scratch takes about 18 minutes for natural data distribution and 10 minutes for the other scenarios. The latter benefit from the fact that each dataset contains an equal

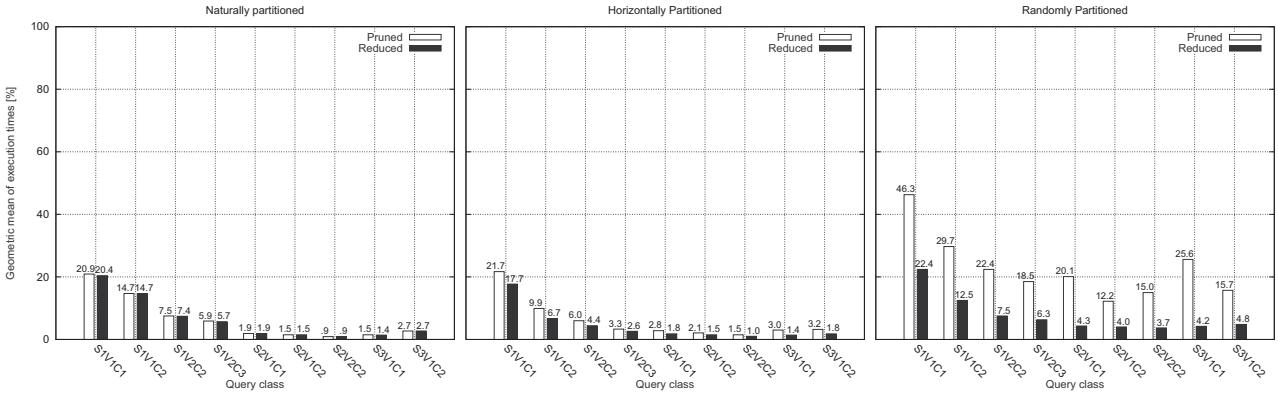

Figure 15: Average query execution times for different optimization levels

number of triples, which increases parallelization. About 30% of the time is spent on building the prefix tree.

5.4 Query Optimization

The average time needed to optimize the queries is shown in Figure 16. The measured running times include all optimization steps, ranging from parsing the query and initial plan generation, to plan simplification and the generation of reducers. As can be seen the running time increases with the complexity of the underlying query pattern. The patterns $S_2V_2C_2$ and $S_3V_1C_2$ are the most difficult, as they contain the most variables and joins. Furthermore the complexity increases slightly when optimizing queries in case of horizontally- and randomly-partitioned data. Analogously to the increasing size of the index this is due to the fact that these partitionings lead to a larger number of unique predicates per dataset and therefore increase the number of partitions. Triples that are summarized in one leaf node in case of natural data distribution are then potentially placed in different nodes. This results in more leaf nodes and the need to join more AVBs during query optimization.

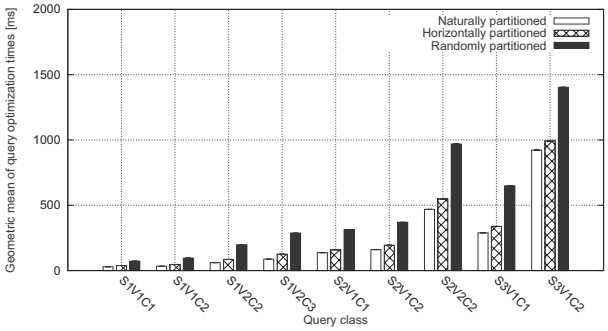

Figure 16: Time needed for query optimization

We also compared our approach to the work presented in [14]. For this purpose we ignored the type information and LSBs stored in our index. Despite the partitioning of triples by data source and predicate, this closely resembles the original index structure. In this case the optimizer was not able to handle about 51% of the workload's queries because it produced a huge number of intermediate results during join processing and ran out of memory. Increasing the heap size limit did not effectively solve this problem. The error rate correlated with the complexity of the underlying query patterns ranging from 0% for the query class $S_1V_1C_1$ to 98% for $S_2V_2C_2$, $S_3V_1C_1$ and $S_3V_1C_2$ respectively. The running times for optimizing the remaining queries increased by an average factor of about 50, ranging from 1.6 for the query class $S_1V_1C_2$ up to 288 for $S_3V_1C_1$.

5.5 Query Execution

Figure 18 shows the average time needed to execute the workload including query optimization. With a workload-average of about one second in case of natural and horizontal partitioning the prototype performs very well considering its simple design. The average query execution times increase by a factor of up to five when the data is partitioned randomly. In contrary to the other scenarios, where most triple patterns can be answered by only one single data source, there are often much more potential sources in this scenario. As a result, less triple patterns can be pushed down and grouped into more complex subqueries which significantly increases the number of subqueries and resulting variable bindings. Although many of these bindings can be pruned, this increases local query execution times because the reducers are only applied to the result sets of the subqueries and are not directly involved into local query processing.

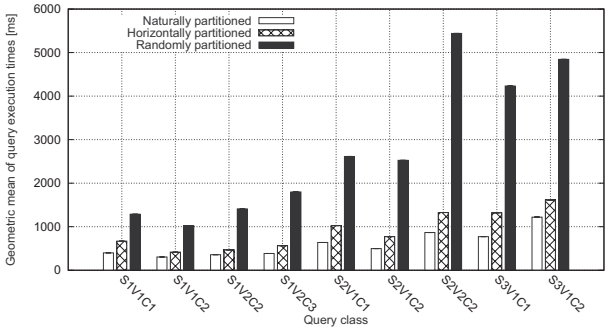

Figure 18: Time needed for query execution

The impact of the different optimizations is shown in more detail in Figure 15. We denote pruning parts of the operator tree with *Pruned*, and further reducing the number of returned variable bindings with *Reduced*. The baseline of 100% is defined by the unoptimized case, in which the step of query simplification is omitted (see section 3.1) and initial execution plans are postprocessed and executed directly without applying any optimizations. In case of natural data distribution, pruning operators reduces the average query execution time by up to two orders of magnitude ($S_2V_2C_2$). Further reduction of the resulting variable bindings yields only an additional speedup of up to 10%. In total, the average query execution time is decreased by a factor of five ($S_1V_1C_1$) to 110 ($S_2V_2C_2$). In this case applying reducers has only little impact, as subqueries do not yield many irrelevant results. When the data is distributed horizontally,

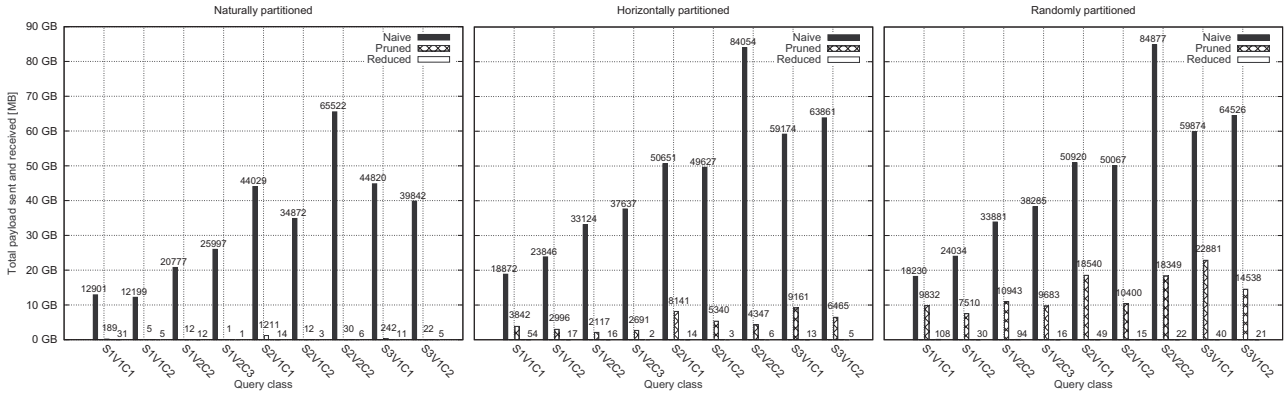

Figure 17: Transferred data volumes for different optimization levels

plan simplification yields a speedup of a factor of five to 65 ($S_2V_2C_2$). Reducers also have a significant impact, further decreasing query execution times by a factor of up to two. Applying all optimizations the average query execution time is decreased by a factor of five ($S_1V_1C_1$) to 100 ($S_2V_2C_2$). In case of random data distribution much more data sources yield relevant results, limiting the impact of plan simplification. This results in a speedup of a factor of two to eight ($S_2V_1C_2$). On the other hand the potential for pruning irrelevant results is significantly increased, leading to an additional speedup of a factor of up to five ($S_3V_1C_1$). This adds up to a total speedup of a factor of four ($S_1V_1C_1$) to 25 ($S_2V_2C_2$).

When compared to natural data distribution, the overall query execution times increase slightly in case of horizontal and significantly in case of random data distribution. For naturally or horizontally distributed data only a single data source returns relevant bindings for most triple patterns. Therefore these scenarios benefit the most from pruning operators from the query execution plan. Horizontal distribution does also offer some potential for reducing the resulting variable bindings, as the unnatural distribution slightly increases the number of irrelevant variable bindings returned by the subqueries. In case of random partitioning pruning parts of the operator tree is more difficult because relevant results for triple patterns can often be returned by more than one data source. On the other hand this significantly increases the potential to reduce the amount of resulting variable bindings. Independent from data distribution the resulting relative speedup increases with the complexity of the underlying query pattern.

Details on the total data volume (over all queries in the respective class) transferred while executing the workload are shown in Figure 17. Here *Naïve* denotes the unoptimized case. Plan simplification reduces the transferred data volume by several orders of magnitude. Further pruning irrelevant variable bindings yields an additional reduction by up to another few orders of magnitude, especially in the case of random data distribution. Due to plan simplification the generated subqueries often correspond to the stars contained in a query when data is distributed naturally or horizontally. The number of bindings for these stars are limited to $\leq$10.000 and therefore the number of bindings returned by all subqueries is often $\leq 10.000 * \#Stars$. Loading a dataset into RDF-3X is very fast for smaller datasets. The total query execution times are thus often dominated by local query processing in this cases because the generated reducers do not affect local query processing. If data is distributed randomly the number of results returned by the subqueries increases significantly. As reducers are able to prune a large amount of these bindings they do have a strong impact on the overall running times in this scenario.

6. DISCUSSION AND FUTURE WORK

We have presented a scalable system for optimizing and executing SPARQL queries over large distributed RDF graphs. Because of the wide-spread use of some vocabularies and the schema-relaxed nature of SPARQL many RDF databases are potentially able to answer a single triple pattern. But if the same triple pattern is part of a more complex SPARQL query many of these answers are irrelevant due to a lack of join partners. The rich RDF-specific synopsis described in this paper enables efficient compile-time and runtime techniques that address this problem. At compile-time, sources that would return only irrelevant results for an individual triple pattern can be pruned from the query execution plan. Especially this technique is also of high relevance for querying the Semantic Web [1]. Because this scenario is characterized by a large number of small datasets that are only accessible via a high latency, low bandwidth network, it is important to minimize the number of subqueries.

Our index structure implements vertical partitioning [7]. It has been shown that this storage scheme does not perform very well for datasets with a large number of distinct RDF predicates when implemented on top of relational database systems [25]. Similar problems arise in our solution for very heterogeneous datasets in which a large number of predicates occur only rarely. As this leads to many small partitions the compression ratio can drop significantly which also has a negative impact on running times. This problem can be overcome by merging small partitions into one common partition. A related approach has been proposed in [22] for other schema-level information.

To improve the performance and results of join-order optimization we are planning to adopt the concept presented in [22]. The basic idea is to provide very accurate cardinality estimations based on a lightweight RDF-specific approach for mining parts of a datasets schema. We feel that this can be implemented for distributed RDF databases and integrated into our system based on the triples' subject, predicate and object types and hash values. There are also several ways to further develop our proof-of-concept query engine. For example, recent work has shown that distributed RDF stores can be queried very efficiently by processing the results of subqueries with MapReduce [16]. Although

the system presented therein also offers query processing over distributed instances of the RDF-3X system, its aim is to use a cluster of machines in order to manage one large RDF dataset. As it needs to control how data is distributed among the local databases its concepts are not directly applicable to our requirements though. Another technique for improving query execution is bind-joins [17]. This concept, which has already been utilized for querying distributed RDF databases in [19] and [24], is also related to semi-join reducers. The basic idea is to compute a join between two subqueries by binding a variable in one query to all according values obtained by executing the other query. In order to make effective use of this optimization, the local databases need to be able to process queries containing initial variable bindings. This is currently not supported in most RDF stores but has been included in the W3C's working draft for the upcoming SPARQL version [5].

In this paper we limited ourselves to a read-only scenario. It is generally possible to propagate updates to our index but the subject, predicate and object hash values and types of triples that have been added, updated or deleted have to be made available. RDF-3X offers a natural way to implement this as it handles updates via a so-called differential index which is periodically merged with the main index. Our index will degrade over time when updates are performed because the MBBs of a PARTree's leaf nodes can not be split in a reasonable way without knowing the original data items. It is therefore recommended (and also very fast) to rebuild the index from scratch at certain intervals.

7. ACKNOWLEDGEMENTS

This work has been supported by the Graduate School of Information Science in Health (GSISH) and the TUM Graduate School.

8. REFERENCES

[1] Linked data - design issues. http://www.w3.org/DesignIssues/LinkedData.html.

[2] OWL web ontology language overview. http://www.w3.org/TR/owl-features/.

[3] RDF primer. http://www.w3.org/TR/rdf-primer/.

[4] RDF vocabulary description language 1.0: RDF schema. http://www.w3.org/TR/rdf-schema/.

[5] SPARQL 1.1 federation extensions. http://www.w3.org/TR/sparql11-federated-query/.

[6] SPARQL query language for RDF. http://www.w3.org/TR/rdf-sparql-query/.

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

[8] M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An empirical study of real-world SPARQL queries. *CoRR*, abs/1103.5043, 2011.

[9] C. Basca and A. Bernstein. Avalanche: putting the spirit of the web back into semantic web querying. In *SSWS*, pages 64–79, 2010.

[10] F. Belleau, M.-A. Nolin, N. Tourigny, P. Rigault, and J. Morissette. Bio2RDF: Towards a mashup to build bioinformatics knowledge systems. *Journal of Biomedical Informatics*, 41(5):706–716, 2008.

[11] K.-H. Cheung, H. R. Frost, M. S. Marshall, E. Prud'hommeaux, M. Samwald, J. Zhao, and A. Paschke. A journey to semantic web query federation in the life sciences. *BMC Bioinformatics*, 10(S-10):10, 2009.

[12] P. Fender and G. Moerkotte. A new, highly efficient, and easy to implement top-down join enumeration algorithm. In *ICDE*, pages 864–875, 2011.

[13] K. Goh, M. E. Cusick, D. Valle, B. Childs, M. Vidal, and A.-L. Barabási. The human disease network. *PNAS*, 104(21):8685–8690, May 2007.

[14] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, and J. Umbrich. Data summaries for on-demand queries over linked data. In *WWW*, pages 411–420. ACM, 2010.

[15] O. Hartig, C. Bizer, and J. C. Freytag. Executing sparql queries over the web of linked data. In *ISWC*, pages 293–309. Springer, 2009.

[16] J. Huang, D. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. In *VLDB*, pages 1123–1134. ACM, 2011.

[17] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[18] G. Ladwig and T. Tran. Linked data query processing strategies. In *ISWC*, pages 453–469. Springer, 2010.

[19] A. Langegger, W. Wöß, and M. Blöchl. A semantic web middleware for virtual data integration on the web. In *ESWC*, pages 493–507. Springer, 2008.

[20] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A simple and efficient algorithm for R-Tree packing. In *ICDE*, pages 497–506. IEEE Computer Society, 1997.

[21] Y. Li and J. Heflin. Using reformulation trees to optimize queries over distributed heterogeneous sources. In *ISWC*, pages 502–517. Springer, 2010.

[22] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, pages 984–994. IEEE Computer Society, 2011.

[23] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.

[24] B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. In *ESWC*, pages 524–538. Springer, 2008.

[25] L. Sidirourgos, R. Goncalves, M. L. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.

[26] K. Stocker, D. Kossmann, R. Braumandl, and A. Kemper. Integrating semi-join-reducers into state of the art query processors. In *ICDE*, pages 575–584. IEEE Computer Society, 2001.

[27] H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, and J. Broekstra. Index structures and algorithms for querying distributed RDF repositories. In *WWW*, pages 631–639. ACM, 2004.

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