Efficient Resource Attribute Retrieval in RDF Triple Stores

Andreas Brodt, Oliver Schiller, Bernhard Mitschang
IPVS, Universität Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany
{brodt, schiller, mitschang}@ipvs.uni-stuttgart.de

ABSTRACT
The W3C Resource Description Framework (RDF) is gaining popularity for its ability to manage semi-structured data without a predefined database schema. So far, most RDF query processors have concentrated on finding complex graph patterns in RDF, which typically involves a high number of joins. This works very well to query resources by the relations between them. Yet, obtaining a record-like view on the attributes of resources, as natively supported by RDBMS, imposes unnecessary performance burdens, as the individual attributes must be joined to assemble the final result records.

We present an approach to retrieve the attributes of resources efficiently. We first determine the resources in question and then retrieve all their attributes efficiently at once, exploiting contiguous storage in RDF indexes. In addition, we present an index structure which is specifically designed for RDF attribute retrieval. Our measurements show that our approach is clearly superior for larger numbers of attributes.

Categories and Subject Descriptors
H.2 [Database Management]: Systems—Query processing; H.3 [Information Storage and Retrieval]: Information Search and Retrieval—Search process; Selection process

General Terms
Performance, Algorithms

Keywords
RDF, SPARQL, triple store

1. INTRODUCTION
The W3C Resource Description Framework (RDF) is a data model for structured information that is gaining popularity in different application domains including life sciences, Web 2.0 platforms, and data integration tasks. RDF provides great flexibility towards any kind of schema and is even usable without a schema at all. RDF allows collecting data in a “pay-as-you-go” fashion, starting with very little schema information and refining the schema later, as required. Thus, RDF allows breaking the traditional data engineering workflow of designing a schema first and populating it later. This suits RDF for scenarios such as community-based data collection, and in fact large knowledge bases from sources including Wikipedia have been built using RDF [2, 11].

RDF models data as a number of (subject, predicate, object)-triples: subject identifies the resource that is being described, predicate denotes a particular relation or attribute, and object is either another resource identifier, or a literal. The triples form a directed labeled graph that represents the resources with their relations and attributes. E.g., the triples in Listing 1 describe four resources: The novel “First Blood” (id1) by David Morrell (id2), on which the first Rambo movie (id3) is based, starring Sylvester Stallone (id4).

SPARQL [9] is a W3C-recommended language to query RDF graphs by graph pattern matching. At its core, a SPARQL query consists of triples containing at least one variable. We refer to such triples as triple patterns. The result is a table of variable bindings, such that the pattern exists in the queried RDF graph using the variable bindings in each result row. For instance, the SPARQL query in

```
SELECT ?first ?last WHERE {
  ?movie <title> "Rambo: First Blood Part 1".
  ?actor <actedIn> ?movie.
  ?actor <firstName> ?first.
  ?actor <lastName> ?last.
}
```

Listing 1: Sample RDF graph

Listing 2: Exemplary SPARQL query
Listing 2 returns the first names and last names of all actors acting in the first Rambo movie. Executed on the RDF graph of Listing 1, the result would be a table with the columns “?first” and “?last”, and two rows holding the values “Michael” and “Stallone”, and “Sylvester” and “Stallone”.

A system that manages RDF data cannot assume a defined schema in the general case. Thus, it possesses little knowledge about the higher-level structure of the data, e.g., entity types like “actor” or “movie”. Hence, it can index and process the data only at triple-level. A common way to manage very large RDF data sets efficiently is to keep the triples in a single three-column table [7, 12, 4] with B+-tree indexes in various (if not all) permutations and aggregations of subject, predicate, and object. Such an index efficiently returns all triples having one or two constant values at a given position, which precisely corresponds to a triple pattern. To combine these triples to the graph pattern defined in the query, joins are required. By carefully choosing indexes on the right triple permutation, index sort orders can frequently be exploited for efficient merge joins. Otherwise, hash joins can be used. We call this approach to generate an execution plan canonical, as it can be applied directly on any SPARQL graph pattern.

Figure 1 shows a canonical execution plan for the query of Listing 2. The query optimizer chose to use the predicate-subject-object (PSO) index, so that the triples corresponding to the triple patterns in lines 2, 3, and 4 are all sorted by their subject, i.e., the actor. This allows using two merge joins. The first triple pattern comes sorted by the movie, so a hash join is required to join it with the rest of the query.

Canonical execution plans are relatively easy to build and able to restrict intermediate results early, through a number of measures [8]. This makes them suitable for complex analytical queries on relationships between many resources; for instance, to identify all actors which acted in a movie which they also directed and which are married to the author of the novel on which the movie is based. However, canonical plans are not necessarily optimal for retrieving attributes of resources. Lines 3 and 4 in Listing 2 illustrate that SPARQL requires one triple pattern per attribute in order to query attributes, such as the first name. I.e., to retrieve n attributes of a resource, a query lists n triple patterns, which a canonical plan evaluates using n index scans and n − 1 joins. Even though most often merge joins will be used to assemble the attributes to larger result tuples, these joins impact query performance severely for larger numbers of attributes.

In a record-based RDBMS, all attributes of a resource are stored contiguously on disk in the same record. Thus, the record can be retrieved in a single fetch operation and does not need to be assembled first. This is not possible for RDF because the schema is unknown. However, the subject-predicate-object (SPO) index is primarily sorted by subject, i.e., it stores all relations and attributes of a resource contiguously. By using the SPO index to retrieve attributes of a resource, a potentially large number of joins can be saved, without compromising the advantages of canonical plans for complex graph patterns.

In this paper we evaluate how the SPO index can be exploited to make resource attribute retrieval in RDF triple stores more efficient. We introduce our attribute retrieval approach (Section 2), discuss its integration into the query optimizer (Section 3), and present a dedicated index for attribute retrieval (Section 4). We evaluate our concepts (Section 5) and conclude with promising future research directions (Section 6).

2. ATTRIBUTE RETRIEVAL APPROACH

Canonical plans use the predicate-subject-object (PSO) index to retrieve one particular attribute, e.g., the first name, for all resources in the data set. Contrarily, the subject-predicate-object (SPO) index supports retrieving all attributes of one given resource without a join on the resource ID for every attribute. This is because the SPO index stores all triples with the same subject (i.e., resource ID) contiguously. On the other hand, it cannot easily identify the relevant triples without knowing the triple subject. The subject is a variable in every triple pattern that retrieves a resource identification attribute retrieval. Alternatively, scanning the SPO index top to bottom and joining the result with the remainder of the query is not an option, as this would scan the entire database.

We propose a processing model that splits SPARQL queries into two conceptual parts: resource identification and attribute retrieval. Resource identification determines the set $R$ of qualifying resources. For each resource $r \in R$, attribute retrieval fetches the values of the attribute set $A$, as defined in the query.

2.1 The Pivot Index Scan Operator

To implement attribute retrieval using the SPO index, we designed the pivot index scan operator. It retrieves the values for a given resource $r$ and attribute set $A$ from the SPO index and pivots them into one or more result tuples in a single operation. The pivot index scan is designed for use under a nested loop join which connects resource identification with attribute retrieval. The nested loop join iterates on all identified resources, invokes the pivot index scan on each resource, and propagates all its result tuples. For the query of Listing 2, this results in an execution plan as shown in Figure 2: Resource identification determines the identifiers of all actors acting in the first Rambo movie. A nested loop join invokes attribute retrieval on each actor: a pivot index scan to look up the first name and the last name of each actor from the SPO index.
# B-tree search for resource \( r \) and smallest attribute \( a_{\min} \)
scan \( \leftarrow \) SPOIndex.openScan\((r, a_{\min})\)
\((s, p, o) \leftarrow \) scan.firstTriple()

foreach \( a \in A \) do # \( A \) is sorted in index order
    while \( s = r \land p < a \) do
        # Skip intermediate triple
        \((s, p, o) \leftarrow \) scan.nextTriple()
        od
    if \((s > r \lor p > a) \land \neg \text{isOptional}(a) \) (*)
        then exit # \( r \) does not have all required attributes fi
    # Buffer all values for \( a \)
    valueBuffer\[a\] \leftarrow \) valueBuffer\[a\] \cup \{o\} (**) 
    \((s, p, o) \leftarrow \) scan.nextTriple()
    while \( s = r \land p = a \) od

Listing 3: Implementation of the pivot operation, which drives a scan on the SPO index (for brevity we assume the scan always returns a triple)

As shown in Figure 2, the pivot index scan takes a variable binding as input, which was produced by resource identification. During one run of the nested loop join, this variable binding is treated as a constant: it is the subject of the triples to look up from the SPO index. The second input for the pivot index scan is the set of attributes \( A \), as defined in the query. Thus, the pivot index scan fetches a set of attributes for a given resource in a single invocation. Figure 2 shows that it possesses a set of output variables for this purpose: one for every attribute to retrieve.

Unlike other operators, the pivot index scan processes sets of triples rather than individual triples. Internally, it consists of two parts: an index scan and a pivot operation. First, the index scan looks up the first triple containing the resource \( r \) as subject, and the smallest attribute \( a_{\min} \in A \) as predicate from the SPO index. This requires the attributes in \( A \) to be sorted in the same order as they appear in the index. Then the pivot operation makes the index scan iterate through the triples of \( r \), until the values for all attributes in \( A \) have been fetched, or all triples about \( r \) have been read. Listing 3 shows how the pivot operation selects all triples of the resource containing a predicate \( p \in A \). The pivot operation projects the objects of these triples and buffers them.

## 2.2 Optional Attributes

SPARQL allows declaring parts of the query *optional*. This corresponds to a one-sided outer join in the relational model, i.e., the optional parts of a query will be evaluated and returned if they are found. Otherwise the projected variables from an optional query part produce NULL values, but do not cause a result tuple to be dismissed. If single triple patterns which retrieve attributes are declared optional, they can be evaluated in a pivot index scan. The pivot index scan operator would normally abort if it cannot find a triple containing a queried attribute for the current resource. To handle optional attributes, the query optimizer only needs to tell the pivot index scan which attributes are mandatory and which are optional. Listing 3 shows how a pivot index scan handles missing optional or mandatory attributes in the line marked with an asterisk (*)

## 2.3 Multi-Attributes

It is not at all safe to assume that a resource carries only one value for a particular attribute. There can be an arbitrary number of triples with the same subject and predicate, but with a different object. In this case, the predicate is a multi-attribute and the semantics of SPARQL require producing the cross product. For instance, the subject \( \text{id4} \) in Listing 1 carries two triples with predicate \( \text{firstName} \) and one triple with predicate \( \text{lastName} \). Consequently, the query of Listing 2 produces two result tuples; one for each first name. It is important to note that the result of a pivot index scan is not one tuple, but a set of tuples.

A canonical plan automatically produces the cross product of multi-attributes when the joins assemble the attributes to larger result tuples. To preserve this behavior, the pivot index scan must buffer all values for each attribute \( a \in A \) when it scans the triples of one resource. It must then calculate the cross product over these buffers in an explicit separate step. Listing 3 shows how a pivot index scan buffers the values indexed by attribute in the line marked with a double-asterisk (**). In a Volcano-style [5] operator implementation providing an open/next/close interface, the \text{open()} \ call has to scan the relevant triples and buffer the attribute values, as shown in Listing 3. It may then close the underlying index scan. The calls to \text{next()} \ then return the next combination of the cross product. The \text{close()} \ call discards the buffer.

## 2.4 Multiply Selected Attributes

SPARQL allows selecting an attribute of a resource more than once. This is particularly interesting in combination with multi-attributes and produces the cross product of all attribute values. For example, Listing 2 could be extended by the triple patterns in Listing 4 to return the cross product on all (ex-) wives of the actor.

This might seem a purely academic case at first, but in fact occurs not infrequently. The cross product on multiply selected attributes is often used to find a particular relation between all values. It is usually combined with a filter to ensure that resulting value pairs contain different values, i.e., \(?\text{spouse1} != ?\text{spouse2}\) in the example. This way, one could determine whether two (ex-) wives of the actor share the same hobby or hair-color, for instance. Naturally, it is perfectly possible to simply query the cross product of the values as (part of) the query result.

A canonical plan will strictly map every triple pattern of the query to an index scan and create joins between them. It will equally apply this principle to triple patterns which retrieve the same attribute and the joins will automatically produce the cross product on the respective values.

A pivot index scan must replicate this behavior explicitly. As discussed above, it must buffer the values of multi-attributes and produce their cross product anyway. To create the cross product on a multiply selected attribute \( a \), the query optimizer must inform the pivot index scan about the number \( k_a \) denoting how often to consider the values for \( a \) in the cross product. The pivot operation can be used as shown in

\begin{verbatim}
?actor <wasOrIsMarriedWith> ?spouse1. 
?actor <wasOrIsMarriedWith> ?spouse2.
\end{verbatim}

Listing 4: Triple patterns extending Listing 2 by a cross product on all values of one attribute
Listing 3. Only the cross product algorithm must consider the value buffer for \( a \) not just once, but \( k \) times.

Formally, this means that the set \( A \) of attributes to retrieve is really a multi-set in the general case. In the interest of a simpler presentation of this paper, we will nevertheless refer to \( A \) as a set. Consequently, by \(|A|\) we do not really mean the number of distinct attributes to retrieve, but the number of triple patterns involved in attribute retrieval. This does not restrict the generality and applicability of our approach.

2.5 Related Work

The idea of a pivot operation transposing rows into columns is not new. Some RDBMS including Microsoft SQL server provide the Pivot and Unpivot data manipulation operators. Pivot transforms a set of rows into a set of fewer rows with additional columns. Unpivot is the reverse operation of Pivot [3]. [13] provides a formal characterization. When the set of columns is unknown in advance, a table can be implemented as a property table that is accessed using the Pivot operator. This is more flexible, as columns can be added as property rows of the form (id, propertyname, propertyvalue). This form is very similar to the RDF data model.

Before Pivot and Unpivot were introduced, application programmers frequently used complex SQL statements with a nested subquery for every pivoted column [3]. These queries were inefficient and hard to optimize, and strongly resembled canonical execution plans for SPARQL queries.

The main difference to our pivot index scan is that Pivot must not produce multi-attributes, as the relational model does not allow them. For this purpose, Pivot takes an aggregation function which collapses multi-attributes into a single value. SPARQL, by contrast, requires returning the cross product on all multi-attribute values.

3. PLAN GENERATION

The classical bottom-up dynamic programming approach of System R [10] is well-suited to create canonical execution plans for SPARQL graph patterns. We first explain bottom-up plan generation for canonical plans, as implemented in [7]. Then we introduce our approach to extend such an optimizer so that it produces execution plans that use pivot index scans, if this is cheaper. We describe our cost model and an approach to cardinality estimation in presence of multi-attributes. Finally, we address selective attributes, which influence the set of selected resources in the query result.

3.1 Generating Canonical Plans

As illustrated in Figure 1, a graph pattern containing \( n \) triple patterns can be solved by a canonical plan which consists of \( n \) index scans and \( n - 1 \) joins. Such a plan can be created using the following recursive rule: A plan which covers one triple pattern is an index scan. A plan covering \( x > 1 \) triple patterns is a join of two sub-plans covering \( a \) and \( b \) distinct triple patterns, respectively, with \( a + b = x \).

A plan covering \( x \leq n \) triple patterns solves the query.

The canonical optimizer uses a table for dynamic programming, which contains all sub-plans produced so far. Thus, the table consists of \( n \) rows, where all plans in row \( x \) cover exactly \( x \) triple patterns. The plans of each row are grouped in sets of plans solving the same problem, i.e., covering the same triple patterns. First, the optimizer seeds the first row of the dynamic programming table with \( n \) sets of index scans. For every triple pattern, it generates an index scan on the index of every possible triple permutation, such that all possible sort orders are considered. E.g., for the second triple pattern of Listing 2, an index scan on the predicate-subject-object (PSO) index produces the triples sorted by actor, whereas using the POS index orders them by movie. Subsequently, the higher rows are populated incrementally. Every plan in row \( x \) is produced by joining one sub-plan from row \( a \) and one sub-plan from row \( b \), with \( a + b = x \). The two sub-plans must share a common variable to join on and their problems must not overlap, i.e., their sets of covered triple patterns must be disjoint. If both sub-plans are sorted on the join variable, a merge join is used; otherwise a hash join is created. As explained in [7], the optimizer should only accept a new plan for a problem, if is cheaper than all other plans of the same problem, or if it introduces a new sort order. Finally, the optimizer chooses the cheapest plan in row \( n \).

3.2 Generating Plans with Pivot Index Scans

Before a pivot index scan can be created, the triple patterns which retrieve attributes have to be identified. For this purpose we sort the triple patterns by subject, predicate, and object. We order variables in the subject before constants, constants in the predicate before variables, and variables in the object before constants. This makes it easy to find set of triple patterns, which have (1) the same subject, (2) a variable subject, (3) a constant predicate, and (4) a variable object. As a side product, this sort order ensures that the set \( A \) of attributes to retrieve is sorted, which is later required by the pivot index scan, as described in Section 2. In addition, we require that the object (5) is listed in the SELECT clause of the query and (6) does not occur in any other triple pattern. Without condition (5), the object value does not need to be retrieved at all. Condition (6) ensures that the triple pattern is not involved in any joins and thus does not belong to the resource identification part of the graph pattern. Finally, we require that (7) every set contains at least two qualifying triple patterns; otherwise a pivot index scan would not save a join and definitely not benefit from storage locality.

Every such set of triple patterns for attribute retrieval can be solved by one pivot index scan. For each set, the predicates of the triple patterns constitute one set \( A \) and the pivot index scan covers \(|A|\) triple patterns. As explained above, a canonical optimizer joins existing sub-plans to larger plans, starting with index scans as atomic plans. A pivot index scan is an atomic plan as well, but it covers more than one triple pattern. Thus, in addition to seeding the first row of the dynamic programming table with index scans, the optimizer must also generate pivot index scans in row \(|A|\) for each set of triple patterns. When the bottom-up algorithm reaches row \( x = |A| + b \), it generates a join between the pivot index scan and all sub-plans in row \( b \). Naturally, these sub-plans must contain the variable which denotes the resource for which the pivot index scan retrieves the attributes. In addition to that, the optimizer must choose a nested loop join instead of a merge or hash join when it joins a pivot index scan. Also, the pivot index scan must become the right child of the join, i.e., it must occur inside the loop.

The nested loop join should occur near the root of the plan. It does, ideally, not restrict the cardinality of the query result (see Section 3.5), but might increase it through cross products quite a bit. Thus, executing the pivot index scan too early in the query plan causes an unnecessarily high number of intermediate result tuples. Moreover, it is in practice not
We experienced about 4000 compressed triples in a single DS which are connected to we will refer to them in the next section, too.

| can be simply estimated as follows:

\[ S = \sum_{i} |A_i| \]

where \( \sum_{i} |A_i| \) denotes the maximal amount of triple patterns covered by all possible pivot index scans.

### 3.3 Cost Model

A pivot index scan executes one B-tree lookup for every resource \( r \) that is produced by resource identification. Even if \( r \) does not carry all mandatory attributes, the pivot index scan only removes \( r \) from the query result after the B-tree lookup has been performed, as illustrated in Listing 3. We do not generally assume that the amount of scanned triples for each resource is significant. A pivot index scan rarely needs to access more than one leaf page of the SPO index for a single resource, as the triples can be compressed very well (we experienced about 4000 compressed triples in a single leave page using the method presented in [7]). The B-tree lookup costs for each resource will by far dominate the costs for sequentially scanning the respective triples. In addition, the cost of the nested loop join operator itself is negligible. Thus, if resource identification is estimated to produce \( |R| \) resources with total costs \( \text{cost}_{\text{res.id.}} \) and one B-tree lookup causes costs of \( \text{cost}_{\text{lookup}} \), then the costs of the nested loop join over resource identification and a pivot index scan \( \text{cost}_{\text{ul}} \) can be simply estimated as follows:

\[
\text{cost}_{\text{ul}} = \text{cost}_{\text{res.id.}} + |R| \times \text{cost}_{\text{lookup}}
\]

This shows that plans using a pivot index scan are practically independent of the number of selected attributes \( |A| \), but in turns do depend on \( |R| \). Thus, it depends on the query and on the RDF data set, whether a pivot index scan should be favored over a canonical plan. Our plan generation approach always creates both possibilities and lets the cost model decide which plan is the cheapest.

### 3.4 Cardinality Estimation

Estimating the cardinality of a plan that uses our attribute retrieval approach is no different from general cardinality estimation, as cardinality depends on the queried data and not on the chosen execution plan. A pivot index scan evaluates a star-shaped graph pattern which comprises many triple patterns. Thus, a method to estimate cardinality of larger subgraphs is desirable, rather than estimating single triple patterns and combining them incrementally.

**Characteristic sets** [6] approximate star-shaped subgraphs in RDF data very well. They describe resources through their predicates, to characterize them in a similar way as entity types would. Thus, characteristic sets can be seen as some sort of “soft entity types”. We introduce them here, as we will refer to them in the next section, too.

A characteristic set \( S_C(s) \) of a triple subject \( s \) occurring in an RDF data set \( DS \) is defined as the set of predicates which are connected to \( s \):

\[
S_C(s) = \{ p \mid \exists o : (s, p, o) \in DS \}
\]

The set of all characteristic sets occurring in \( DS \) is consequently defined as follows:

\[
S_C(DS) = \{ S_C(s) \mid \exists p, o : (s, p, o) \in DS \}
\]

Neumann and Moerkotte observed that the amount of characteristic sets in a real-world RDF data set is surprisingly low. They compute and store all characteristic sets for an RDF data set \( DS \) and count the resources belonging to each characteristic set \( S_C \), denoted by \( \text{count}_{S_C} \). To estimate the number of resources matching a star-shaped graph pattern, all characteristic sets have to be determined that fully include the predicates of the pattern. Summing up \( \text{count}_{S_C} \) for all these characteristic sets, returns the number of resources matching the pattern.

The output cardinality equals this number of resources, conditioned there are no multi-attributes. Otherwise, the cross product over multi-attributes must be taken into account. To estimate multi-attributes, Neumann and Moerkotte compute \( \text{count}_{S_C}(p) \) for each predicate \( p \) as the total sum of how often \( p \) occurred in a resource belonging to \( S_C \). Thus, the average occurrence of \( p \) in a resource belonging to \( S_C \) is

\[ \frac{\text{count}_{S_C}(p)}{\text{count}_{S_C}(\text{Res})} \]

For instance, in Listing 1 two resources (id2 and id4) belong to a characteristic set \( S_C \) that fully includes \{firstName, lastName\}, such that \( \sum_{S_C} \text{count}_{S_C} = 2 \). Within these two resources, the firstName predicate occurs three times, so \( \sum_{S_C} \text{count}_{S_C}(\text{firstName}) = 3 \). Thus, on average, these resources carry \( \frac{3}{2} \) first names.

To estimate the output cardinality of a pivot index scan with subject variable \( ?s \), the set \( P_Q(?s) \) of predicates which are connected to \( ?s \) in the query \( Q \) has to be determined:

\[ P_Q(?s) = \{ p \mid \exists o : (?s, p, o) \in Q \} \]

Is important to note, that \( P_Q(?s) \) must also include predicates which are not retrieved by the pivot index scan, but still connected to its subject variable in the query. Resources which do not possess these predicates will never reach the pivot index scan and are therefore irrelevant for output cardinality. Thus, for the query in Listing 2, \( P_Q(?actor) \) is \{firstName, lastName, actedIn\}, even if the pivot index scan does not retrieve actedIn. Subsequently, all characteristic sets are determined which fully include \( P_Q(?s) \). The occurrence counts of every predicate \( p \in P_Q(?s) \) are summed up to \( \text{count}(p) \) across all these characteristic sets, and their resource counts to \( \text{count}(\text{Res}) \):

\[
\text{count}(p) = \sum_{S_C \supseteq P_Q(?s)} \text{count}_{S_C}(p)
\]

\[
\text{count}(\text{Res}) = \sum_{S_C \supseteq P_Q(?s)} \text{count}_{S_C}(\text{Res})
\]

The output cardinality is \( \text{count}(\text{Res}) \) times the factor by which the cross-product over multi-attributes increases the output. This factor can be estimated by multiplying the average occurrence of each predicate \( p \in P_Q(?s) \), as described above. This, however, does not take multiply selected attributes into account, as discussed in Section 2.4. Thus—and this is a contribution beyond the cardinality estimation approach of [6]—every predicate must be considered in the cross-product estimation as often as the query selects it. I.e., in the product, the average occurrence of each predicate in the characteristic sets considered must be multiplied by its oc-
3.5 Selective Attributes

The basic assumption of our approach is that resource identification, which determines the set $R$ of qualifying resources, is highly selective. Also, attribute retrieval, which fetches the queried attribute values for each resource $r \in R$, is assumed to influence $R$ very little. While this assumption intuitively holds for many cases, it is far from a given. It is not true, if only small fraction of $R$ carries certain attributes of $A$. I.e. the existence of attributes is selective.

If, e.g., the following triple pattern was added to the SPARQL query of Listing 2, it would not return any results for the data set of Listing 1:

\[
\text{?actor} \text{ <wonOscarForBestActorInYear> ?year}. 
\]

Whereas most actors possess at least one first name and a last name, most actors (including Sylvester Stallone, at the time of writing of this paper) never win the academy award for the best actor in any year. Thus, \text{wonOscarForBestActorInYear} is very selective.

In a canonical execution plan, the joins between the index scans retrieving attributes will eliminate resources that lack a mandatory attribute. The optimizer will arrange joins over selective attributes early in the plan. And through sideways information passing \cite{8}, operators higher in the query plan may skip non-qualifying resources early. By contrast, a pivot index scan will only realize that a resource $r$ lacks a mandatory attribute, when it has already found $r$ in the SPO index—and the index lookup is the expensive part. Unlike the cascaded joins of a canonical plan, which are able to influence each other in a zig-zag fashion, a pivot index scan has no chance to influence resource identification.

3.5.1 Execution Plans for Selective Attributes

The only way to avoid failing index lookups in a pivot index scan due to selective attributes is to call it only on resources that possess all mandatory attributes in the first place. This means that resource identification must restrict $R$ to such resources. The obvious solution is to remove a selective attribute $a_{sel}$ from the set $A$ of attributes which the pivot index scan retrieves, and to process it in the manner of a canonical plan: using a separate index scan on the PSO index and one join. The join is able to eliminate resources that lack $a_{sel}$ early, so that they never reach the pivot index scan. We refer to this approach as "reduced" in our evaluation.

The separate index scan will return only resources which possess $a_{sel}$. It will also retrieve the corresponding attribute values. On the other hand, if the optimizer still chooses to use a pivot index scan, the scan will also find the values for $a_{sel}$ practically for free. So instead of removing the entire triple pattern that includes $a_{sel}$ from the pivot index scan, it would be enough to ensure that every resource that enters attribute retrieval really possesses $a_{sel}$—without retrieving the value. For this, an index that only includes the predicate and the subject, but lacks the object value, would fully suffice.

The RDF-3X triple store \cite{7} creates, in addition to the six possible triple permutations, also six \textit{aggregated} indexes (SP*, PS*, OP*, ...). The aggregations project away one column of the triples and store the amount of resulting duplicates. E.g., the PS* index stores the number of triples in the data set with a particular predicate and subject. RDF-3X uses the aggregated indexes to evaluate triple patterns containing variables that are not used anywhere else in the query, i.e. their values are not required. It also uses the aggregated indexes for cardinality estimation of single triple patterns. Naturally, they are smaller than the full triple indexes.

Thus, instead of removing $a_{sel}$ from the pivot index scan completely, as the "reduced" approach would, we can also add an index scan on the aggregated PS* index for $a_{sel}$ to resource identification, as shown in Figure 3. The aggregated index scan returns less intermediate results than the full triple index and thus produces less join costs. Nevertheless, it is still able to restrict $R$ on resources which possess $a_{sel}$. Finally, the pivot index scan will retrieve the values for $a_{sel}$, together with all other attribute values. We call this approach "checked", as the aggregated index scan checks the existence of the selective attribute.

3.5.2 Identifying Selective Attributes

To identify selective attributes, we need to characterize the resources that \textit{enter} attribute retrieval. For this we take a look at the set $P$ of predicates that are connected to this subject variable in the query, but not contained in the attribute list of the pivot index scan; i.e. $P \cap A = \emptyset$. For the query in Listing 2, $P$ would be \{actedIn\}. In order to qualify for resource identification, a resource $r$ must possess all predicates $p \in P$. But differently, if $r$ lacks any of these predicates, it never reaches attribute retrieval. Next, we need a method to determine, which other predicates these resources typically carry, once they possess all predicates $p \in P$. I.e. we need to characterize the resources based on the predicates in $P$. This is exactly what characteristic sets \cite{6} do, as explained in Section 3.4.

First, we need to retrieve all characteristic sets in the RDF data set $DS$ that fully include $P$. By summing up the number of resources belonging to these sets, $\text{count}_{S_C}(\text{Res})$, we obtain an upper bound $\text{card}_{p}$ for the input cardinality $|R|$ of the pivot index scan.

\[
\text{card}_{p} = \sum_{S_C \supseteq P} \text{count}_{S_C}(\text{Res})
\]

To see how each attribute $a \in A$ influences cardinality, we can similarly calculate the amount of resources which possess $a$, as well as all predicates in $P$:

\[
\text{card}_{a} = \sum_{S_C \supseteq \{a\} \cup P} \text{count}_{S_C}(\text{Res})
\]
If \( \text{card}_a \ll \text{card}_P \), then \( a \) is selective with respect to resources possessing all predicates in \( P \). This means, that a pivot index scan likely performs many index lookups in vain, because many resources entering attribute retrieval lack \( a \).

This approach only works if the query contains other triple patterns with constant predicates for the subject variable of the pivot index scan (otherwise \( P = \emptyset \)). If this is not the case, it is difficult to characterize the resources that enter attribute retrieval. Assuming statistical independence, we can still roughly determine selective attributes based on their general occurrence in the entire data set. For this we need to determine how many resources carry a particular attribute.

We can obtain this information using characteristic sets as well. For every retrieved attribute \( a \in A \), we only need to determine every characteristic set \( S_C \) that includes \( a \). By summing up \( \text{count}_{S_C}(\text{Res}) \) for all these sets, we obtain the number of resources in the data set that possess the attribute \( a \). If one attribute returns significantly fewer resources than others, it is selective with respect to the entire data set.

The second approach assumes that resource identification selects resources that behave similarly to all other resources in the data set. In practice, selectivity of an attribute may depend quite heavily on the particular resources in question. E.g., the attribute \( \text{wonOscarForBestActorInYear} \) is utterly selective for actors or people in general, but much less for people honored with a star on the Hollywood Walk of Fame. Thus, we try to identify selective attributes using the characteristic sets that contain the predicates that are not retrieved by a pivot index scan. If this is not possible, we use general occurrence of each retrieved attribute in the data set.

4. Attribute Retrieval Index

Most RDF triple stores map the actual values of the triples to integer IDs, as they can be stored and processed much more efficiently than the strings, of which these values mostly consist [1, 7, 4]. Consequently, a dictionary must map every query result back to its string representation before returning it. This mapping step can cause significant overhead for large query results; we observed a fraction of over 90% of the total execution time for some queries.

Our attribute retrieval approach exploits locality of RDF triples about the same resource. It retrieves the attribute values from the SPO index to produce the final query result. Due to condition (6) in Section 3.2, these values cannot be involved in further operators of the query, i.e. they are simply returned for output. To save the mapping costs caused by the dictionary, we implemented a variant of the SPO index that does not store the integer IDs for the object values. Instead, it directly points to the external representation of the object value. Subject and predicate are still represented as IDs, because they are required for query processing, and because they are not an output of the pivot index scan. As a pivot index scan never searches by object value, the inner nodes of this attribute retrieval index are organized exactly as in the aggregated SP* index. I.e., they omit the triple object, which increases their fan-out by 25%.

The leaf nodes are specifically designed for the access pattern of a pivot index scan. The access pattern starts with the subject representing the resource in question and looks for its list of predicate/object pairs. After the predicate/object pairs have been found, the subject is no longer required. Thus, as shown in Figure 4 we store all subjects contiguously at the beginning of a leaf node. After each subject we store the length (in bytes) of the subject’s predicate/object list. We store the predicate/object list of each subject contiguously at a location further behind in the page. All predicates and objects are stored in the same page for all subjects, except for the last one. The last subject continues on the next page, if the page header indicates this. The page header also contains the in-page byte offset that points to the beginning of the first predicate-object list. At the same time, this offset marks the exclusive end of the subject list.

We use a simple compression technique for the leaf nodes. We use delta encoding for the subject list and for the predicates of each subject. Both are sorted integer IDs, which allows storing only the difference to their predecessor. This results in smaller integer numbers, which we store using 7-bit variable-length encoding. In most cases, this requires only one byte per subject or predicate. To point to the object values, we store the uncompressed page number and the in-page offset that identify the location where the dictionary stores the external value representation.

The storage addresses of the object values are longer than the internal IDs of the objects. Moreover, the addresses are not sorted. This makes the triples harder to compress than in the fully sorted SPO index. In our experiments, the compression rate was worse by a factor of about 2.2, compared to the compression method for the SPO index presented in [7]. Nevertheless, our compression technique for the attribute retrieval index stored the triples in roughly 50% of their uncompressed size. In addition, the attribute retrieval index is only intended for point queries, so the absolute index size is not primarily an issue. For point queries, it is much more important to access the relevant parts of the leaf nodes quickly, at which the attribute retrieval index excels.

When a pivot index scan searches for the attributes of a resource, it first locates the right leaf node through standard B-tree search. It opens the leaf node and reads the offset pointing to the end of the subject list. Then, it reads the (sorted) subject list sequentially until it finds the subject. The compressed subjects do not occupy much space, so that the linear search benefits from a good processor cache locality and can exploit hardware branch prediction well. For each subject, the pivot index scan reads the length of the corresponding predicate/object list. It adds the length to two total lengths; one including and one excluding the current subject. Added to the offset of the first predicate-object list, these lengths mark the offset where the current predicate-object list begins and ends in the page, respectively. The predicate-object list is processed as shown in Listing 3.

5. Evaluation

We carried out an extensive performance evaluation to compare (1) canonical plans with execution plans that utilize
our attribute retrieval approach (2) on the “ordinary” SPO index and (3) on the attribute retrieval index.

5.1 Implementation

We implemented our attribute retrieval approach and integrated it into the RDF-3X [7] triple store, version 0.3.4, which is available as open source.\(^1\) However, our approach is applicable for any triple store that indexes RDF data on a per-triple basis in several permutations (including the subject-predicate-object permutation, of course), which is the current state of the art.

In addition to adding a new logical and physical database operator, we had to modify the query optimizer, as described in Section 3.2. To integrate the attribute retrieval index, we also had to modify data import, and—most notably—large parts of the runtime system and the dictionary. As a consequence of the attribute retrieval index, intermediate query results no longer consist of dictionary IDs only. Instead, they may consist of IDs and addresses of object values. The runtime system must differ between both types, which makes the code quite a bit more complicated.

5.2 Test Setup

We generated artificial data, which allowed us to study different execution plans in a controlled way. Our test databases contained between 59.7 and 149.7 million triples, described 1 million resources, and occupied 6.6 to 15 GB of space.

We ran all test queries ten times consecutively on warm caches and measured the execution times. Our figures report the median of the ten test runs. In addition, we counted the logical page requests to get an impression on the main memory (if not disk) I/O behavior of the execution plans. Moreover, to examine only the physical database operators, we measured execution times and page requests both including and excluding the dictionary. I.e. in the latter case, the raw internal IDs and addresses were returned instead of resolving the external string representation of the values.

We ran all tests on a Dell Optiplex 755 equipped with an Intel Core2 Quad Q9300 CPU running at 2.50 GHz and 4 GB of main memory. We used two striped 250 GB SATA 3.0 GB/s hard drives spinning at 7.200 RPM. The test machine ran a 64 bit 2.6.31 Linux kernel.

5.3 Resources versus Attributes

The most important and most interesting question is, how the different execution plans behave for different amounts of queried resources $|R|$ and retrieved attributes $|A|$. To answer this, we ran a large series of queries on a database containing 1 million resources. Every resource carried exactly 30 attributes with random values, so that they can be assumed distinct. To simulate resource identification, we created 1000 resource groups of different sizes. The group members were chosen randomly. A resource is a member of a particular group, if it is connected to the group URI via the inResourceGroup predicate. This enabled us to select a defined amount of resources using a single triple pattern.

The test queries retrieved between $|A| = 2$ and $|A| = 30$ attributes. Figure 5 shows the results for $|R| = 100$, 2500, and 5000 resources, both retrieving only the raw internal IDs, and the total effort including the dictionary overhead. They lead to the following observations:

1. The raw figures of our approach and the attribute retrieval index are always simply horizontal lines. Thus, for a given number of resources $|R|$, our approach requires constant time and page accesses, independent of $|A|$. This is because it accesses all attributes of a resource from a contiguous memory area, so that it
effectively makes no difference, how many attributes it retrieves. The canonical plans scale linearly with |A|, as they require one join per retrieved attribute.

2. The difference between the raw and the total figures, i.e. the dictionary overhead, is the same for canonical plans and our approach on the SPO index. On the attribute retrieval index, the dictionary overhead is about 20% less for execution times and nearly 50% for page accesses. This is because it does not need to resolve the internal IDs of the attribute values.

3. The attribute retrieval index shows far better execution times than all other approaches for |A| > 3, both with and without the dictionary overhead. However, it accesses the identical amount of pages as our approach on the SPO index. Thus, the leaf page layout must make the difference. The attribute retrieval index only accesses a small part of every leaf page and only uncompresses what it needs to read. The other indexes must uncompress the entire page first.

4. The break-even-point of canonical plans and our approach on the SPO index is (in this evaluation) around

\[
\frac{|R|}{|A| + 1} \approx 300.
\]

Higher values favor canonical plans, below our approach on the SPO index is faster. The attribute retrieval index was always faster than both of them for |A| ≥ 3.

5.4 Multi-Attributes

As pointed out in Section 2.3, SPARQL returns the cross-product over multi-attributes. This results in exponentially growing results, if the number of retrieved attributes |A| is increased. We ran large test series on databases similar to the one described in Section 5.3, except that every resource carried each attribute two, three, or four times, respectively.

For lower amounts of computed cross-products, the results were similar to those of Section 5.3. The more cross-products the multi-attributes caused, the closer the raw execution times of the different plans became (i.e. not counting dictionary overhead). For extremely high numbers of result tuples, the canonical plan showed faster raw execution times, even though it accessed significantly more pages. We presume that this is due to the more sequential memory access pattern of the streamlined merge joins that leads to better CPU cache-hit rates across the triples of different resources when computing the cross-products. A pivot index scan bears excellent memory locality for one resource, but it causes random memory access for different resources.

Figure 6 shows the raw execution times of one test series on a database with two multi-attributes. For 5000 resources, three attributes were retrieved once and a fourth attribute was selected two to eight times. It is observable, that the execution times become similar with high amounts of multiply-selected attributes. For lower amounts, however, our approach is more than an order of magnitude faster on the attribute retrieval index. The results are roughly comparable to a different test series on the same database that retrieved the same amount of distinct attributes, i.e. without multiply selected attributes. Thus, multiply selected attributes are hardly different from distinct selections, as long as the number of cross-products is the same.

5.5 Selective Attributes

As discussed in Section 3.5, naively applying our approach on attributes which only occur at a small fraction of resources will result in many failing index lookups. To measure this effect, we used the database of Section 5.3 and added 100 selective attributes, which occurred only at 0.1% to 10% of all resources. We randomly chose the resources which carried a selective attribute. Yet we ensured that the fraction of resources with selective attributes was the same for all resource groups. Also, for every selective attribute, every group contained at least one resource that carried the attribute.

Our queries retrieved four “ordinary” attributes, as in Section 5.3, and one selective attribute from different amounts of resources. Thus, if all attributes were always present, we would expect our approach to be superior up to about 1800 resources. We compared a canonical execution plan, a naive application of our approach, the “reduced” approach of Section 3.5.1, and the “checked” approach of Section 3.5.1.

Figure 7 shows the measured execution times and page requests for |R| = 100, 2500 and 5000 resources. As expected, the naive application of our approach is by far worse than the canonical plan, both for execution times and page requests. The “reduced” and “checked” plans, on the other hand, are more than twice as fast as the canonical plan and also request significantly fewer pages. The “checked” plan is always slightly better than the “reduced” plan.

6. CONCLUSIONS

The W3C Resource Description Framework (RDF) supports managing semi-structured data without a predefined database schema. RDF models data as triples which may represent a relation between two resources or an attribute of a resource. So far, the focus of most RDF query processors has been on finding complex graph patterns in RDF data based on the relations between resources. This typically involves a high number of joins. By contrast, obtaining a record-like view on the attributes of resources, as natively supported by record-based DBMS, imposes unnecessary performance burdens in these query processing models. They must join the individual attributes to assemble the final result records, as they do not differ between finding resources that match a graph pattern and retrieving resource attributes.

We proposed a processing model that splits queries to RDF data into two conceptual parts: resource identification and attribute retrieval. First, resource identification determines the resources of interest for the query. In a second step, attribute retrieval fetches the queried attributes in a single
step for each identified resource. For this, we exploit an index that stores all attributes of a resource contiguously, which saves a large number of joins. Most RDF stores already possess such an index anyway; they just do not exploit it in our proposed way. In addition to that, we proposed an index structure that is specifically designed for the access pattern of attribute retrieval. Our performance evaluation showed that our processing model is clearly superior for larger numbers of retrieved attributes and moderately large amounts of resources. Moreover, our attribute retrieval index further improves performance by large margins.

The underlying concept of our presented query processing model is not restricted to RDF data management. We believe it is applicable on any decomposed storage model. It is an interesting challenge to apply it on other types of database systems, such as column-oriented databases or object databases, in the future.

Acknowledgments
We thank our students Tim Waizenegger and Bastian Reitschuster for strongly supporting our evaluation. Furthermore, we thank our colleagues Matthias Großmann, Carlos Lübbe and Nazario Cipriani for the fruitful discussions on the paper.

7. REFERENCES