Exploiting Emergent Schemas to make RDF systems more efficient

Minh-Duc Pham and Peter Boncz

CWI, Amsterdam, The Netherlands,
{duc,boncz}@cwi.nl

Abstract. We build on our earlier finding that more than 95% of the triples in actual RDF triple graphs have a remarkably tabular structure, whose schema does not necessarily follow from explicit metadata such as ontologies, but for which an RDF store can automatically derive by looking at the data using so-called "emergent schema" detection techniques. In this paper we investigate how computers and in particular RDF stores can take advantage from this emergent schema to more compactly store RDF data and more efficiently optimize and execute SPARQL queries. To this end, we contribute techniques for efficient emergent schema aware RDF storage and new query operator algorithms for emergent schema aware scans and joins. In all, these techniques allow RDF schema processors fully catch up with relational database techniques in terms of rich physical database design options and efficiency, without requiring a rigid upfront schema structure definition.

1 Emergent Schema Introduction

In previous work [15], we introduced *emergent schemas*: finding that >95% of triples in all LOD datasets we tested, including noisy data such as WebData Commons and DBpedia, conform to a small relational tabular schema. We provided techniques to automatically and at little computational cost find this "emergent" schema, and also to give the found columns, tables, and "foreign key" relationships between them short *human-readable labels*. This label-finding, and in fact the whole process of emergent schema detection, exploits not only value distributions and connection patterns between the triples, but also additional clues provided by RDF ontologies and vocabularies.

A significant insight from that paper is that relational and semantic practitioners give different meanings to the word "schema". It is thus a misfortune that these two communities are often distinguished from each other by their different attitude to this ambiguous concept of "schema" – the semantic approach supposedly requiring no upfront schema ("schema-last") as opposed to relational databases only working with a rigid upfront schema ("schema-first").

Semantic schemas, primarily ontologies and vocabularies, aim at modeling a knowledge universe in order to allow diverse current and future users to denote these concepts in a universally understood way in many different contexts. Relational database schemas, on the other hand, model the structure of one particular dataset (i.e., a database), and are not designed with a purpose of re-use in different contexts. Both purposes are useful: relational database systems would be easier to integrate with each other if the semantics of a table, a column and even individual primary key values (URIs) would be well-defined and exchangeable. Semantic data applications would benefit from knowledge of the actual patterns of co-occurring triples in the LOD dataset one tries to query, e.g. allowing users to more easily formulate SPARQL queries with a non-empty result (this often results from using a non-occurring property in a triple pattern).

In [15], we observed *partial* and *mixed* usage of ontology classes across LOD datasets: even if there is an ontology closely related to the data, only a small part of its class attributes actually occur as triple properties (partial use), and typically many of the occurring attributes come from different ontologies (mixed use). DBpedia on average populates <30% of the class attributes it defines [15], and each actually occurring class contains attributes imported from no less than 7 other ontologies on average. This is not necessarily bad design, rather good re-use (e.g. foaf), but it underlines the point that any single ontology class is a poor descriptor of the actual structure of the data (i.e., a "relational" schema). Emergent schemas are helpful for human RDF users, but in this paper, we investigate how RDF stores can exploit emergent schemas for efficiency.

We address three important problems faced by RDF stores. The first and foremost problem is the high execution cost resulting from the large amount of self-joins that the typical SPARQL processor (based on some form of triple table storage) must perform: one join per additional triple pattern in the query. It has been noted [7] that SPARQL queries very often contain star-patterns (triple patterns that share a common subject variable), and if the properties of the patterns in these stars reference attributes from the same "table", the equivalent relational query can be solved with a table scan, not requiring *any* join. Our work achieves the same reduction of the amount of joins for SPARQL.

The second problem we solve is the low quality of SPARQL query optimization. Query optimization complexity is *exponential* in the amount of joins [17]. In queries with more than 12 joins or so, optimizers cannot analyze the full search space anymore, potentially missing the best plan. Note that SPARQL query plans typically have F times more joins than equivalent SQL plans. Here F is the average size of a star pattern[1]. This leads to a 3^F times larger search space. Additionally, query optimizers depend on cost models for comparing the quality of query plan candidates, and these cost models assume independence of (join) predicates. In case of star patterns on "tables", however, the selectivity of the predicates is heavily correlated (e.g. subjects that have an `ISBN` property, typically instances of the class `Book`, have a much higher join hit ratio with `AuthoredBy` triples than the *independence assumption* would lead to predict) which means that the cost model is often wrong. Taken together, this causes the quality of SPARQL query optimization to be significantly lower than in SQL. Our work eliminates many joins, making query optimization exponentially eas-

[1] A query of X stars has $X \times F$ triple patterns, so needs $P_1 = X \times F$-1 joins. When each star is collapsed into one tablescan, just $P_2 = (X$-1) joins remain: $\frac{P_1}{P_2} \geq F$ times.

ier, and eliminates the biggest source of correlations that disturb cost modeling (joins between attributes from the same table).

The third problem we address is that mission-critical applications that depend on database performance can be optimized by database administrators using a plethora of physical design options in relational systems, yet RDF system administrators lack all of this. A simple example are *clustered indexes* that store a table with many attributes in the value order of one or more sort key attributes. For instance, in a data warehouse one may store sales records ordered by Region first and ProductType second – since this accelerates queries that select on a particular product or region. Please note that not only the Region and ProductType properties are stored in this order, but *all* attributes of the sales table, which are typically retrieved together in queries (i.e. via a star pattern). A similar relational physical design optimization is table-partitioning or even database cracking [9]. Up until this paper, one cannot even think of the RDF equivalent of these, as table clustering and partitioning implies an understanding of the structure of an RDF graph. Emergent schemas allow to leave the "pile of triples" quagmire, so one can enter structured data management territory where advanced physical design techniques become applicable.

In all, we believe our work brings RDF datastores on par with SQL stores in terms of performance, without losing any of the flexibility offered by the RDF model, thus without introducing a need to create upfront or enforce subsequently any explicit relational schema.

2 Emergent Schema Aware RDF Storage

The original emergent schema work allows to store and query RDF data with SQL systems, but in that case the SQL query answers account for only those "regular" triples that fit in the relational tables. In this work, our target is to answer SPARQL queries over 100% of the triples correctly, but still improve the efficiency of SPARQL systems by exploiting the emergent schema.

RDF systems store triple tables T in multiple orders of Subject (S), Property (P) and Object (O), among which typically T_{PSO} ("column-wise"), T_{SPO} ("row-wise") and either T_{OSP} or T_{OPS} ("value-indexed") – or even all permutations.[2]

In our proposal, RDF systems storage should become emergent schema aware by only changing the T_{PSO} representation. Instead of having a single T_{PSO} triple table, it gets stored as a set of wide relational tables in a *column*-store – we use MonetDB here. These tables represent only the regular triples, the remaining $< 5\%$ of "exception" triples that do not fit the schema (or were updated recently) remain in a smaller PSO table T_{pso}. Thus, T_{PSO} is replaced by the union of a smaller T_{pso} table and a set of relational tables.

Relational storage of triple data has been proposed before (e.g. property tables [20]), though these prior approaches advocated an explicit and human-controlled mapping to a relational schema, rather than a transparent, adaptive and automatic approach, as we do. While such relational RDF approaches have

[2] To support named RDF graphs, the triples are usually extended to quads. Our approach trivially extends to that but we discuss triple storage here for brevity.

performance advantages, they remained vulnerable in case SPARQL queries *do not* consist mainly of star patterns and in particular when they have triple patterns where the P is a variable. This would mean that many, if not all, relational tables could contribute to a query result, leading to huge generated SQL queries which bring the underlying SQL technology to its knees.

Our proposal hides relational storage behind T_{PSO}, and has as advantage that SPARQL query execution can always fall back on existing mechanisms – typically MergeJoins between scans of T_{SPO}, T_{PSO} and T_{OPS}. Our approach at no loss of flexibility, just makes T_{PSO} storage more compact as we will discuss here, and creates opportunities for better handling of star patterns, both in query optimization and query execution, as discussed in the following sections.

Formal definition. Given the RDF triple dataset $\Delta = \{t | t = (t_S, t_P, t_O)\}$, an emergent schema $(\Delta, \mathcal{E}, \mu)$ specifies the set $\mathcal{E}$ of emergent tables T_k, and mapping μ from triples in Δ to emergent tables in $\mathcal{E}$. A common idea we apply is rather than storing URIs as some kind of string, to represent them as an OID (object identifier) – in practice as a large 64-bit integer. The RDF system maintains a dictionary $\mathcal{D} : OID \rightarrow URI$ elsewhere. We use this $\mathcal{D}$ dictionary creatively, adapting it to the emergent schema.

Definition 1. *Emergent tables ($\mathcal{E} = \{T_1, ..\}$): Let $s, p_1, p_2, ..., p_n$ be subject and properties with associated data types OID and $D_1, D_2, ..., D_n$, then $T_k = (T_k.s{:}OID, T_k.p_1{:}D_1, T_k.p_2{:}D_2, ..., T_k.p_n{:}D_n)$ is an emergent table where $T_k.p_j$ is a column corresponding to the property p_j and $T_k.s$ is the subject column.*

Definition 2. *Dense subject columns: $T_k.s$ consists of densely ascending numeric values β_k, .. $\beta_k + |T_k| - 1$, so s is something like an array index, and we denote $T_k[s].p$ as the cell of row s and column p. For each T_k its base OID $\beta_k = k * 2^{40}$. By choosing β_k to be sufficiently apart, in practice the values of column $T_i.s$ and $T_j.s$ never overlap when $i \neq j$.*[3]

Definition 3. *Triple-Table mapping ($\mu : \Delta \rightarrow \mathcal{E}$): For each table cell $T_k[s].p_j$ with non-NULL value o, $\exists(s, p_j, o) \in \Delta$ and $\mu(s, p_j, o) = T_k$. These triples we call "regular" triples. All other triples $t \in \Delta$ are called "exception" triples and $\mu(t) = T_{pso}$. In fact T_{pso} is exactly the collection of these exception triples.*

The emergent schema detection algorithm [15] assigns each subject to at most 1 emergent table – our storage exploits this by manipulating the URI dictionary $\mathcal{D}$ so that it gives dense numbers to all subjects s assigned to the same T_k.

Columnar relational storage. On the physical level of bytes stored on disk, columnar databases can be thought of as storing all data of one column consecutively. Column-wise data generally compresses better than row-wise data because data from the same distribution appears consecutively, and column-stores exploit

[3] In our current implementation with 64-bit OIDs we thus can support up to 2^{16} emergent tables with each up to 2^{40}=1 trillion subjects, still leaving the highest 8 bits free, which are used for type information – see footnote 4.

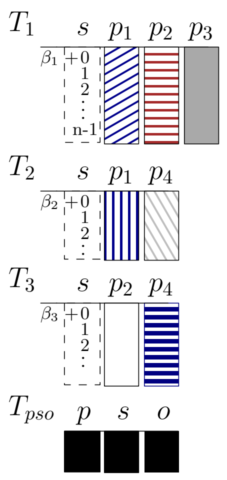


Fig. 1: Columnar Storage of Emergent Tables T_k and exception table T_{pso}

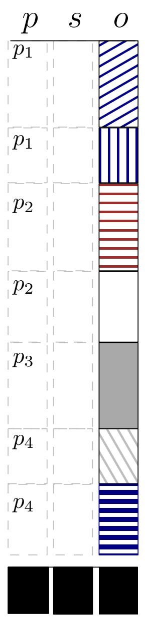


Fig. 2: PSO as view $P_{PSO} \cup T_{pso}$

	Except%	Null%	Compr
Synthetic RDF datasets			
LUBM	0.0%	6.0%	1.8x
BSBM	0.0%	4.2%	2.5x
SP2Bench	0.4%	5.2%	2.0x
LDBC SNB	0.0%	12.2%	2.0x
RDF datasets with Relational Roots			
MusicBrz	0.4%	3.9%	2.2x
Eurostat	0.5%	3.8%	1.4x
DBLP	0.4%	12.6%	1.7x
PubMed	0.3%	15.3%	1.9x
Native RDF datasets			
WebData	7.5%	42.7%	1.4x
DBpedia	3.8%	32.2%	1.4x

Fig. 3: Exception percentage, NULL percentage and Compression Factor achieved by Emergent Table-aware PSO storage, over normal PSO storage.

this by having advanced data compression methods built-in in their storage and query execution infrastructure. In particular, the dense property of the columns $T_k.s$ will cause column-stores to compress it down to virtually nothing, using a combination of delta encoding (the difference between subsequent values is always 1) and run-length encoding (RLE), encoding these subsequent 1's in just a single run. Our evaluation platform MonetDB supports densely ascending OIDs natively with its VOID (virtual OID) type, that requires no storage.

Figure 1 shows an example of representing RDF triples using the emergent tables $\{T_1, T_2, T_3\}$ and the triple table of exception data T_{pso} (in black, below). We have drawn the subject columns $T_k.s$ transparent and with dotted lines to indicate that there is no physical storage needed for them.

For each individual property column $T_k.p_j$, we can define a triple table view $P_{j,k}$=(p_j,$T_k.s$,$T_k.o$), the first column being a constant value (p_j) which thanks to RLE compression requires negligible storage and the other two reusing storage from emergent table T_k. If we concatenate these views $P_{j,k}$ ordered by j and k, we obtain table $P_{PSO} = \cup_{j,k} P_{j,k}$. This P_{PSO} is shown in Figure 2. Note that P_{PSO} is simply a re-arrangement of the columns $T_k.p_j$. Thus, with emergent schema aware storage, one can always access the data P_{PSO} as if it were a PSO table at no additional cost.[4] In the following, we show this cost is actually less.

Space Usage Analysis. P_{PSO} storage is more efficient than PSO storage in an efficient columnar RDF store such as Virtuoso would be. Normally in a PSO table, the P is highly repetitive and will be compressed away. The S column is

[4] SQL-based SPARQL systems (MonetDB, Virtuoso) still allow SQL on T_k tables.

ascending, so delta-compression will apply. However, it would not be dense and it will take some storage ($\log_2(W)$ bits per triple, where W is the average gap width between successive s values[5]) – while a dense S column takes no storage.

Compressing-away the S column is only possible for the regular part P_{PSO}, whereas the exception triples in T_{pso} must fall back to normal PSO triple storage. However, the left table column of Figure 3 shows that the amount of exception triples is negligible anyway – it is almost 0 in synthetic RDF data (stemming from the LUBM, BSBM, SP2Bench and LDBC Social Network Benchmark), as well as in RDF data with relational roots (EuroStat, PubMed, DBLP, MusicBrainz), and is limited to $< 10\%$ in more noisy "native" RDF data (WebData Commons and DBpedia). A more serious threat to storage efficiency could be the NULL values that emergent tables introduce, which are table cells for which no triple exists. In the middle column we see that the first-generation RDF benchmarks (LUBM, BSBM, SP2Bench) ignore the issue of missing values. The more recent LDBC Social Network benchmark better models data with relational roots where this percentage is roughly 15%. Webdata Commons, which consists of crawled RDFa, has most NULL values (42 percent) and DBpedia roughly one third. We note that the percentage of NULLs is a consequence of the emergent table algorithm trying to create a compact schema that consists of relatively few tables. This process makes it merge initial tables of property-combinations into tables that store the union of those properties: less, wider, tables means more NULLs. If human understandability were not a goal of emergent table detection, parameters could be changed to let it generate more tables with less NULLs. Still, space saving is not really an argument for doing so, as the rightmost table column of Figure 3 shows that emergent table storage is overall at least a factor 1.4 more compact than default PSO storage.

Query Processing Microbenchmark. While the emergent schema can be physically viewed as a compressed PSO representation, we now will argue that every use a RDF store will give to a PSO table can be supported at least as efficiently on emergent table aware storage.

Typically, the PSO table is used for three access patterns during SPARQL processing: i) Scanning all the triples of a particular property p (i.e., p is known), ii) Scanning with a particular property p and a range of object value (i.e., p is known + condition on o), and iii) Having a subset of S as the input for the scan on a certain p value (i.e., typically s is sorted, and the system performs a filtering MergeJoin). The first and the second access patterns can be processed on the emergent schema in the similar way as with the original PSO representation by using a UNION operator: $\sigma(pso, p, o) = \sigma(P_{PSO}, p, o) \cup \sigma(T_{pso}, p, o)$

The third access pattern, which is a JOIN with s candidate OIDs is very common in SPARQL queries with star patterns. We test two different cases: with and without of exceptions (i.e. T_{pso}).

Without T_{pso}. In this case, the JOIN can be pushed through the P_{PSO} view and is simply the UNION of JOINs between the s candidates and dense $T_k.s$

[5] $W = \frac{1}{n-1}\sum_1^{n-1}(s_{i+1} - s_i)$ where s_i is the subject OID at row i (table with n rows)

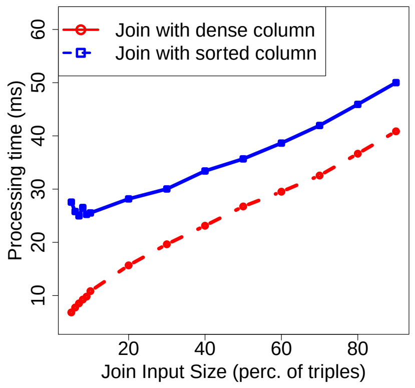


Fig. 4: PSO join performance vs input size (no exceptions)

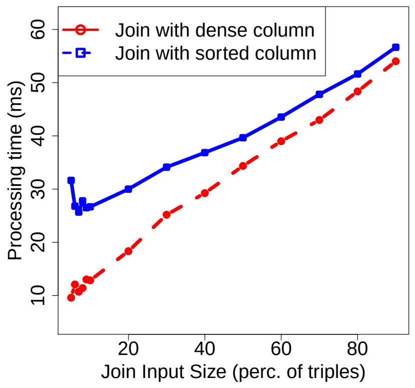


Fig. 5: PSO join performance vs input size (with exceptions)

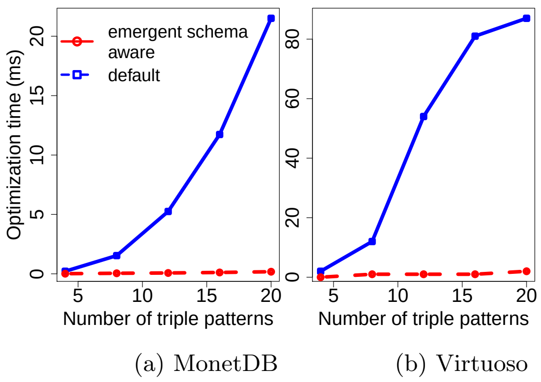
columns in each emergent table T_k. MonetDB supports joins into VOID columns very efficiently, essentially this is sequential array lookup.

We conducted a micro-benchmark to compare the emergent schema aware performance with normal PSO access. It executes the JOIN between a set of $I.s$ input OIDs with two different $T_k.s$ columns: a dense column and a sorted (but non-dense) column; in both cases retrieving the $T_k.o$ object values. The benchmark data is extracted from the subjects corresponding to the `Offer` entities in BSBM benchmark, containing ≈5.7 million triples. Each JOIN is executed 10 times and the minimum running time is recorded. Figure 4 shows that dense OID joins are 3 times faster on small inputs: array lookup is faster than MergeJoin.

With T_{pso}. Handling exception data requires merging the result produced by the JOIN between input ($I.s$) and the dense S column of emergent table $T_k.s$ with the result produced by the JOIN between $I.s$ and the exception table $T_{pso}.s$ – the latter requires an actual MergeJoin. We implemented an algorithm that performs both tasks simultaneously. In order to form the JOIN result between $I.s$ with both $T_k.s$ and $T_{pso}.s$ simultaneously, we modify the original MergeJoin algorithm by checking for each new index of $I.s$, whether the current element from $I.s$ belongs to the dense range of $T_k.s$.

We conducted another micro-benchmark using the same 5.7 million triples. The exception data is created by uniformly sampling 3% of the regular data (BSBM itself is perfectly tabular and has no exceptions). We note that 3% is already more than the average percentage of exception data in all our tested datasets. The list of input $I.s$ candidates is also generated by sampling from 5% to 90% of the regular data. Figure 5 shows that the performance of the JOIN operator on the emergent schema still outperforms that on the original PSO representation even though it needs to handle exception data.

The conclusion of this section is that emergent schema aware storage reduces space by 1.4 times, provides faster PSO access, and importantly hides the relational table storage from the SPARQL processor – such that query patterns that would be troublesome for property tables (e.g. unbound property variables) can still be executed without complication. We take further advantage of the emergent schema in many common query plans, as described next.



(a) MonetDB (b) Virtuoso

Fig. 6: Optimization time as a function of query size (#triple patterns)

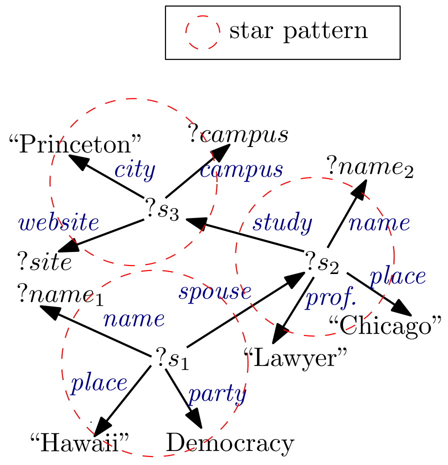


Fig. 7: Example SPARQL graph with three star patterns

3 Emergent Schema Aware SPARQL Optimization

The core of each SPARQL query is a set of (s,p,o) triple patterns, in which s, p, o are either literal values or *variables*. Viewing each pattern as a property-labeled edge between a subject and object, these triples form a *SPARQL graph*. We *group* these triple patterns, where originally each triple pattern is a group of one.

Definition 4. *Star Pattern (ρ=($\$s, p_1, o_1$), ($\$s, p_2, o_2$), ...): A star pattern is a collection of* more than one *triple patterns from the query, that each have a* constant *property p_i and an* identical subject variable *$\$s$.*

To exploit the emergent schema, we identify star patterns in the query and at the query optimization, group query's triple patterns by each star. Joins are needed only between these triple pattern groups. Each group will be handled by one table scan subplan that uses a new "*RDFscan*" operator described further on. SPARQL query optimization then largely becomes a *join reordering* problem. The complexity of join reordering is exponential in the number of joins.

To show the effects on query optimization performance, we created a micro-benchmark that forms queries consisting of (small) stars of size F=4. The smallest query is a single star, followed by one with two stars that are connected by sharing the same variable for an object in the first star and the subject of the star star, etc (hence queries have 4, 8, 12, 16 and 20 triple patterns). Our optimization identifies these stars, hence after grouping star patterns their join graph reduces to 0, 1, 2 and 3 joins respectively. We ran the resulting queries through MonetDB and Virtuoso and measured *only* query optimization time.

Figure 6 shows that emergent schema aware SPARQL query optimization becomes orders of magnitude faster thanks to its simplification of the join ordering problem. The flattening Virtuoso default line beyond 15 patterns suggests that with large amount of joins, it stops to fully traverse the search space using cut-offs, introducing the risk of choosing a sub-optimal plan.

4 Emergent Schema Aware SPARQL Execution

The basic idea of emergent schema aware query execution is to handle a complete star pattern ρ with one relational table scan(T_i,[p_1,p_2,..]) on the emergent table T_i with whose properties p_i from ρ. Assuming a SQL-based SPARQL engine, as is the case in Virtuoso and MonetDB, it is crucial to rely on the existing relational table scan infrastructure, so that advanced relational access paths (clustered indexes, partitioned tables, cracking [9]) get seamlessly re-used.

In case of multiple emergent tables matching star pattern ρ, the *scan plan* (denoted ϑ_ρ) we generate consists of the UNION of such table scans. In ϑ_ρ we also push-down certain relational operators (at least simple filters) below these UNIONs – a standard relational optimization. This push-down means that selections are executed before the UNIONs and optimized relational access methods can be used to e.g. perform IndexScans. For space reasons we cannot go into all details, although we should mention that OPTIONAL triple patterns in ρ are marked and can be ignored in the generated scans (because missing property values are already represented as NULL in the relational tables). Another detail is that on top of ϑ_ρ, we must introduce a Project operator to cast SQL literal types to a special SPARQL value type, that allows multiple literal types as well as URIs to be present in one binding column.[6] Executing (pushed-down) filter operations while values are still SQL literals allows to avoid most casting effort, since after selections much fewer tuples remain.

This whole approach will still only create bindings for the "regular" triples. To generate the 100% correct SPARQL result, we introduce an operator called **RDFscan**, that produces *only* the missing bindings. The basic idea is to put another UNION on top of the scan plan ϑ_ρ that adds the RDFscan(ρ) bindings to the output stream, as shown in Figure 9. Unlike normal scans, we cannot push down filters below the RDFscan - hence these selections remain placed above it, at least until optimization 1 (see later).

Generating Exception Bindings. Correctly generating all result bindings that SPARQL semantics expect is non-trivial, since the exception triples in T_{pso} when combined with *any* emergent table T_k (not only those covering ρ) could produce valid bindings. Consider the example SPARQL query, consisting of a single star pattern and two selections ($o_1 > 10$, $o_3 = 5$):

```
SELECT ?s ?o1 ?o2 WHERE { ?s  p1  ?o1 .
                          ?s  p2  ?o2 .
                          ?s  p3    5 . FILTER (?o1 > 10) }
```

[6] In our MonetDB implementation, the 64-bits OID that encodes (subject) URIs, also encodes literals by using other patterns in its highest 8 bits.

T_1

s	p_1	p_2	p_3
100	11	2	5
101	13	4	6
102	14		5
103	9	6	
104	15	8	5

T_2

s	p_1	p_3	p_4
200	11	7	1
201		5	2
202	13	9	3

T_{pso}

p	s	o
p_1	0	20
p_1	1	9
p_1	201	15
p_2	0	8
p_2	102	6
p_2	201	4
p_3	0	5
p_6	6	7

Result

s	o_1	o_2
100	11	2
104	15	8
0	20	8
102	14	6
201	15	4

Fig. 8: Example RDF data and expected query result.

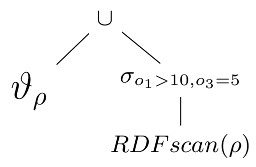


Fig. 9: Query plan for handing exception

Figure 8 shows the expected result of this query on an example data. (For a better view of the example, we assume s base OID of T_1, T_2 are 100, 200, respectively). In this result, the first two tuples come from the regular triples while the last three tuples is the combination of triples stored in T_{pso} table (i.e., in red color) with those stored in tables T_1 and T_2.

Basic approach. RDFscan returns all the bindings for a star pattern, in which each binding is generated by at least one irregular triple (the *missing* bindings). Formally, given a star pattern $\rho = \{(s,\ p_i,\ o_i),\ i = 1,..,k\}$, the RDF dataset $\varDelta$, the output of the RDFscan operator for this star pattern is defined as:

$$RDFscan(\rho) = \{(s, o_1, ..., o_k)\} | (s, p_i, o_i) \in \varDelta \wedge (\exists i : (s, p_i, o_i) \in T_{pso}) \qquad (1)$$

RDFscan generates the "exception" bindings in 2 steps:
Step 1: Get all possible bindings (s, o_1, ..., o_k) where each o_i stems from triple (s,p_i,o_i) $\in T_{pso}$ (for those p_i from ρ), or o_i=NULL if such a triple does not exist, with the constraint that at least one of the object values o_i is non-NULL.
Step 2: Merge each binding (s, o_1, ...,o_k) with the emergent table T_k corresponding to s ($\beta_k \leq s < \beta_k + |T_k|$) to produce output bindings for $RDFscan$.

Step 1 is implemented by first extracting the set E_i of all $\{(s,\ o_i)\}$ corresponding to each property p_i from the T_{pso}: $E_i = \sigma_{p=p_i}(T_{pso})$. Then, it returns the output, S_1, by performing a relational OuterJoin on s between all E_i. We note that, as T_{pso} table is sorted by p, extracting E_i from T_{pso} can be done with no cost by reading a slice of T_{pso} from the starting row of p_i and the ending row of p_i (the information on starting, ending rows of each p in T_{pso} table is pre-loaded before any query processing). Furthermore, as for each p in T_{pso}, $\{(s,o)\}$ are sorted according to s, E_i are also sorted by s. Thus, the full OuterJoin of all E_i can be efficiently done by using a multi-way sort merging algorithm. Figure 10 demonstrates Step 1 for the example query.

Step 2 merges each tuple in S_1 with a tuple of the same s in the regular table in order to form the final output of $RDFscan$. For example, the 4th tuple of S_1 (201, 15, 4, *null*) merged with the 2nd tuple of T_2 (201, *null*, 5, 2) returns a valid binding (201, 15, 4) for the (s, o_1, o_2) of the example query. Figure 11 shows the detailed algorithm of Step 2. For each tuple t in S_1, it first extracts the corresponding regular table and row Id of the current $t.s$ from encoded information inside each s OID (Line 2). Then, for each property p_i, the algorithm

E_1

s	o_1
0	20
1	9
201	15

E_3

s	o_3
0	5

E_2

s	o_2
0	8
102	6
201	4

Output(S_1)

s	o_1	o_2	o_3
0	20	8	5
1	9		
102		6	
201	15	4	

Fig. 10: Step 1

E_1

s	o_1
0	20
201	15

Output(S_1)

s	o_1	o_2	o_3
0	20	8	5
102		6	
201	15	4	

Fig. 12: Step 1 output with pushing down Selection predicates

```
Require: $S_1$: Step 1 output
         $lstP$: List of required properties
         $\mathcal{E}$: Emergent tables
Ensure:  $T_{fin}$: Merging results
 1: for each tuple $t$=($s$, $o_1$,...,$o_k$) in $S_1$ do
 2:   $id$, $r$ = getT_row($t.s$)   # table & row id
 3:   $accept$ = true
 4:   for each $p_i$ in $lstP$ do
 5:     if $t.o_i$ = null & $\mathcal{E}[id]\,[r].p_i$ = null then
 6:       $accept$ = false
 7:       Continue next tuple
 8:     else
 9:       store_cand($bind$, $t.o_i$, $\mathcal{E}[id]\,[r].p_i$)
10:     end if
11:   end for
12:   if $accept$ = true then
13:     append($T_{fin}$, $bind$)
14:   end if
15: end for
```

Fig. 11: Merge-exception-regular algorithm

will check whether there is any non-NULL object value appearing in either t (i.e., $t.o_i$) or the regular column p_i (i.e., $\mathcal{E}[id]\,[r].p_i$) (Line 5). If yes, the non-NULL value will be placed in the binding for p_i (Line 9). Otherwise, if both of the values are NULL, there will be no valid binding for the current checking tuple t. Finally, the binding that has non-NULL object values for all non-optional properties will be appended to the output table T_{fin}.

Optimization 1: Selection push-down. Pushing selection predicates down in the query plan is an important query optimization technique to apply filters as early as possible. This technique can be applied to RDFscan when there is any selection predicate on the object values of the input star pattern (e.g., $o_1 > 10$, o_3=5 in the example query). Specifically, we push the selection predicates down in Step 1 of the RDFscan operator to reduce the size of each set E_i (i.e., $\sigma_{p=p_i}(T_{pso})$), accordingly returning a smaller output S_1 of this step. Formally, given λ_i being a selection predicate on the object o_i, the set E_i of $\{(s,o_i)\}$ from T_{pso}) is computed as: $E_i = \sigma_{p=p_i,\lambda_i}(T_{pso})$. In the example query, $E_1 = \sigma_{p=p_1,o_1>10}(T_{pso})$. Figure 12 shows that the size of E_1 and the output S_1 are reduced after applying the selection pushdown optimization, which thus improves the processing time of RDFscan operator.

Optimization 2: Early check for missing property. If a regular table T_k does not have p_i in its list of columns, to produce a valid binding by merging a tuple t of S_1 (i.e., output of Step 1) and T, the exception object value $t.o_i$ must be non-NULL. Thus, we can quickly check whether t is an invalid candidate without looking into the tuple from T_k by verifying whether t contains non-NULL object values for all missing columns of T_k. We implement this by modifying the algorithm for Step 2. Before considering the object values of all properties from both exception and regular data (Line 4), we first check exception object value $t.o_i$ of each missing property to prune the tuple if any $t.o_i$ is NULL. Then, we continue the original algorithm with the remaining properties.

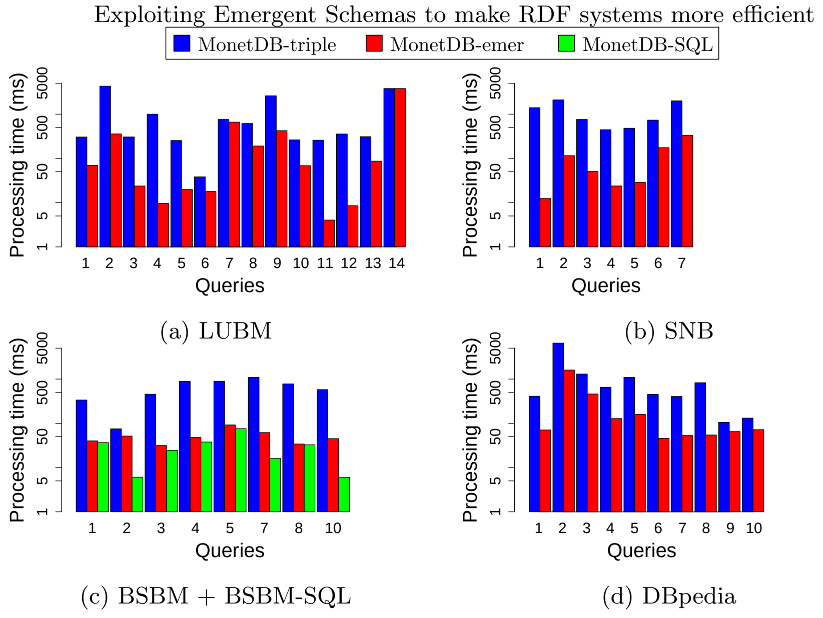
Optimization 3: Prune non-matching tables. The exception table T_{pso} mostly contains triples whose subject was mapped to some emergent table. For example, the triple (201, p_2, 4) refers to the emergent table T_2 because $s \geq 200{=}\beta_2$. During the emergent schema exploration process [15] this triple was temporarily stored in the *initial* emergent table T_2', but was then moved to T_{pso} during the so-called "schema and instance filtering" step. This filtering moves not only triples but also whole columns from initial emergent tables to T_{pso}, in order to derive a compact and precise emergent schema. Assume column p_2 was removed from T_2 during schema filtering. We observe that before filtering, *all* triples (*regular + exception triples*) of subject s were part of the initial emergent table which means that had a particular set of properties. Accordingly, if C is the set of columns of an initial emergent table T' and if C does not contain the set of properties in ρ, there cannot be a matching subject with all properties of ρ stemming from T' even with the help of T_{pso}. This observation can be exploited to prune all subject ranges corresponding to (initial) emergent tables that cannot have any matching for ρ from the pass over T_{pso}.

Specifically, we pre-store, for each emergent table, its set of columns C before schema and instance filtering was applied during emergent schema detection. Then, given the input star pattern ρ, the possible matching tables for ρ are those tables whose set of columns C contain all properties in ρ. Finally, Step 1 is optimized by removing from E_i all the triples that the subject does not refer to any of the matching tables.

5 Performance Evaluation

We tested with both synthetic and real RDF datasets BSBM [1], LUBM [8], LDBC-SNB[6] and DBpedia (DBPSB) [12]; and their respective query workloads. For BSBM, we also include its relational version, namely BSBM-SQL, in order to compare the performance of the RDF store against a SQL system (i.e., *MonetDB-SQL*). We used datasets of 100 million triples for LUBM and BSBM, and scale factor 3 ($\approx$200 million triples) for LDBC-SNB. The experiments were conducted on a Linux 4.3 machine with Intel Core i7 3.4Ghz CPU and 16 GBytes RAM. All approaches are implemented in the RDF experimental branch of MonetDB.

Query workload. For BSBM, we use the SELECT queries from Explore workload (ignoring the queries with DESCRIBE and CONSTRUCT). For LUBM, we use its published queries and rewrite some queries (i.e., Q4, Q7, Q8, Q9, Q10, Q13) that requires certain ontology reasoning capabilities in order to account for the ontology rules and implicit relationships. For LDBC-SNB, we use its short read queries workload. DBPSB exploits the actual query logs of the DBpedia SPARQL endpoints to build a set of templates for the query workload. Using these templates, we create 10 non-empty result queries w.r.t DBpedia 3.9



(a) LUBM (b) SNB

(c) BSBM + BSBM-SQL (d) DBpedia

Fig. 13: Query processing time: Emergent schema-based vs triple-based

Queries	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
Operators: **O**PTIONAL, **F**ILTER, **U**NION	-	-	O	-	U	F	-	F,U	O,F,U	O
Modifiers: **D**istinct, **L**imit, **O**RDER	D	D	D,L,O	D	D,L	D	D,L	-	-	D
# of triple pattern	4	5	5	3	10	3	6	4	6	7
# constraints on O?	1	0	1	1	2	2	1	4	2	0
Has multi-valued prop.?	√	√	√	√	√	√	√	-	√	-

Table 1: Properties of DBpedia queries

dataset[7]. Table 1 show the features of tested DBpedia queries. In Figures 13, 14 and 15, X-axis holds query-numbers: 1 means Q1. For each benchmark query we run three times and record the last query execution (i.e., Hot run).

Emergent schema aware vs Triple-based RDF stores. We perform the benchmarks against two different approaches of MonetDB RDF store: the original triple-based store (*MonetDB-triple*) and the emergent schema-based store (*MonetDB-emer*).

Figure 13 shows the query processing time using two approaches over four benchmarks. For BSBM and LDBC-SNB, the emergent schema aware approach significantly outperforms the triple-based approach in all the queries, by up to two orders of magnitude faster (i.e., Q1 SNB). In a real workload such as DBpedia where there is significant amount of exception triples, our approach is still much faster (note: logscale) by up to more than an order of magnitude (Q8). We also note that multi-valued properties appear in most of DBpedia queries, and this is costly for the emergent schema aware approach as it requires additional MergeJoins to retrieve the object values. In Figure 13d, the best-performing query Q8 is the one having no multi-valued property.

[7] The detailed DBpedia queries can be found at goo.gl/RxzOmy

For LUBM, a few queries (i.e. 7, 14) show comparable processing times for triple-table based and emergent schema aware query processing. The underlying reason is that each subject variable in these queries only contains one or two common properties (e.g., Q14 only contains one triple pattern with the properties `rdf:type`). Thus, the emergent schema aware approach will not improve the query execution time – however as the optimization does not trigger then it also does not degrade performance in absence of fruitful star patterns. For the queries having *discriminative* properties [15] in a star pattern (e.g., Q4, 11, 12), the emergent schema aware approach significantly outperforms the original triple-based version, by up to two order of magnitude (i.e, Q4).

Emergent schema-based RDF store vs RDBMS As shown in Figure 13c, the emergent schema aware SPARQL processing (*MonetDB-emer*) provides comparable performance on most queries (i.e., Q1,Q3,Q4,Q5,Q8) compared to *MonetDB-SQL*. In other queries (Queries 7,10), the emergent schema aware approach also significantly reduces the performance gap between SPARQL and SQL, from almost two orders of magnitude slower (*MonetDB-triple* vs *MonetDB-SQL*) to a factor of 3.8 (*MonetDB-emer* vs *MonetDB-SQL*).

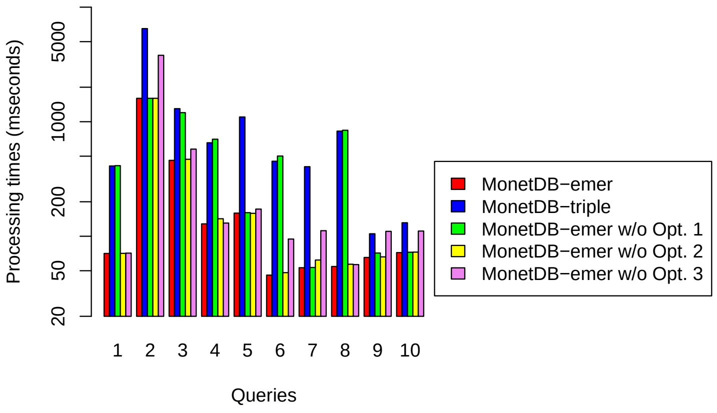
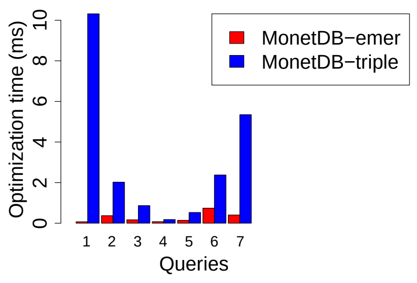


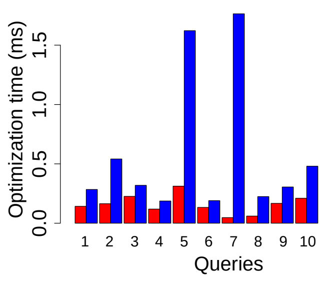
Fig. 14: Query processing with/with-out optimizations

RDFscan optimizations. Figure 14 shows the effects of each of the three described RDFscan optimization by running the DBpedia benchmark without using with each of them. All optimizations have positive effects, though in different queries, and the longer running queries show stronger effects. Selection push-down (Opt. 1) has most influence, while the early check in T_{pso} to see if it delivers missing properties has the least influence. Obviously, selection push-down does not give any performance boost when there is no constraint on the object variables in the queries (e.g., Query 2). For queries having constraints on the object variables, which are quite common in any query workload, it does speed up query processing by up to a factor of 24 (i.e., Q8).

Query optimization time. Figure 15 shows query optimization time on LDBC-SNB and DBPSB (due to lack of space, we omit similar results for BSBM and LUBM). For all queries, the emergent schema aware approach significantly lowers



(a) SNB



(b) DBpedia

Fig. 15: Optimization time: Emergent schema-based vs triple-based

optimization time, by even up to two orders of magnitude (Q1 SNB) or a factor of 37 (Q7 DBPSB). Note also that due to the smaller plan space and strong reduction of join correlations, query optimization also qualitatively improves, a claim supported by its performance improvements across the board.

6 Related Work

Most state-of-the-art RDF systems store their data in triple- or quad-tables creating indexes on multiple orders of S,P,O[19, 5, 14, 16]. However, according to [7, 15], these approaches have several RDF data management problems including unpredictably bad query plans and low storage locality.

Structure-aware storage was first exploited in RDF stores with the"property tables" approach[20, 4, 10, 18]. However, early systems using this approach [20, 4] do not support automatic structure recognition, but rely on a database administrator doing the table modeling manually. Automatic recognition is introduced in some newer systems[10, 18, 11], however unlike emergent schemas these structures are not apt for human usage, nor did these papers research in depth integration with relational systems in terms of storage, access methods or query optimization. Recently, Bornea et al.[2] built an RDF store, DB2RDF, on top of a relational system using hash functions to shred RDF data into multiple multi-column tables. This approach (nor any of the others) allows both SQL and SPARQL access to the same data, as emergent schemas do. Gubichev et al. [7] and Neumman et al.[13] use structure recognition to improve join ordering in SPARQL queries alone. Brodt et al. [3] proposed a new operator, called *Pivot Index Scan*, to efficient deliver attribute values for a resource (i.e., subject) with less joins using something similar to a SPO index – as such it does not recognize structure in RDF to leverage it on the physical level.

7 Conclusion

Emergent Schema detection is a recent technique that automatically analyzes the actual structure an RDF graph, and creates a compact relational schema that fits most of the data. We investigate here how these Emergent Schemas, beyond helping humans to understand a RDF dataset, can be used to make RDF stores more efficient. The basic idea is to store the majority of data, the "regular" triples (typically >95% of all data) in relational tables under the hood, and the remaining “exception” triples in a reduced PSO triple table. This storage still allows to see the relational data as if it were a PSO table, but is in fact > 1.4x more compact and faster to access than a normal PSO table. Furthermore, we provide a simple optimization heuristic that groups triple patterns by star-shape. This reduces the complexity of query optimization by often more than a magnitude, since the size of the join graph is reduced thanks to only joining these groups. Finally, we contribute the RDFscan algorithm with three important optimizations. It is designed to work in conjunction with relational scans, which perform most of the heavy-lifting, and can benefit from existing physical storage optimizations such as table clustering and partitioning. RDFscan keeps the overhead of generating additional binding results for “exception” triples low, yielding overall speed improvements of 3-10x on a wide variety of datasets and benchmarks, closing the performance gap between SQL and SPARQL.

References

1. Bizer, C., Schultz, A.: The berlin sparql benchmark. IJSWIS (2009)
2. Bornea, M., et al.: Building an efficient rdf store over a relational database. In: SIGMOD (2013)
3. Brodt, A., et al.: Efficient resource attribute retrieval in rdf triple stores. In: CIKM (2011)
4. Chong, E., et al.: An efficient sql-based rdf querying scheme. In: VLDB (2005)
5. Erling, O.: Virtuoso, a hybrid RDBMS/graph column store. DEBull (2012)
6. Erling, O., et al.: The ldbc social network benchmark. In: SIGMOD (2015)
7. Gubichev, A., et al.: Exploiting the query structure for efficient join ordering in sparql queries. In: EDBT. pp. 439–450 (2014)
8. Guo, Y., et al.: Lubm: A benchmark for owl knowledge base systems. Web Semantics 3, 158–182 (2005)
9. Idreos, S., et al.: Database cracking. In: CIDR. Asilomar, California (2007)
10. Levandoski, J., et al.: Rdf data-centric storage. In: ICWS (2009)
11. Matono, A., et al.: Paragraph tables: A storage scheme based on rdf document structure. In: DEXA (2012)
12. Morsey, M., et al.: Dbpedia sparql benchmark–performance assessment with real queries on real data. ISWC 2011 (2011)
13. Neumann, T., et al.: Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In: ICDE (2011)
14. Neumann, T., et al.: Rdf-3x: a risc-style engine for rdf. VLDB (2008)
15. Pham, M.D., et al.: Deriving an emergent relational schema from rdf data. In: WWW (2015)
16. Tsialiamanis, P., et al.: Heuristics-based query optimisation for sparql. In: EDBT (2012)
17. Ullman, J., Widom, J.: Database systems: The complete book (2008)
18. Wang, Y., et al.: Flextable: using a dynamic relation model to store rdf data. In: DASFAA (2010)
19. Weiss, C., et al.: Hexastore: sextuple indexing for semantic web data management. VLDB (2008)
20. Wilkinson, K.: Jena property table implementation (2006)