

# Integrating custom index extensions into Virtuoso RDF store for E-Commerce applications

Matthias Wauer, Andreas Both,  
Stephan Schwinger, Martin Nettling  
R & D, Unister GmbH  
Barfußgässchen 11  
Leipzig, Germany  
{firstname.lastname}@unister.de

Orri Erling  
OpenLink Software  
London  
UK  
oerling@openlinksw.com

## ABSTRACT

Triple stores are the backbone of the evolution of the Linked Data Web. However, even standard queries on many attributes can result in high query response times using available triple stores. For use in commercial search applications, particularly for web scale applications, short and predictable query response times are a strict necessity in order to achieve industrial applicability. In this paper, a custom index extension for the Virtuoso triple store is used to improve the runtime of typical queries. The index extension is connected to a lightweight bitset index that covers the properties in question. A case study with real world data in the e-commerce domain shows that more comprehensive queries even on highly materialized data result in consistently improved query response times with a predictable upper bound that meets industrial requirements. Hence, to the best of our knowledge, industrial applicability is achieved for the first time considering the requirements of the e-commerce industry.

## Categories and Subject Descriptors

H.2.4 [Systems]: Query Processing

## General Terms

Semantic Web

## Keywords

Semantic Web; Triple Store; SPARQL; Query processing; Index structure; Bitset

## 1. INTRODUCTION

Using RDF as a data model for search applications has several benefits. In contrast to a fixed schema in relational databases, the schemaless approach is much more flexible in a rapid development process. The basic data model also facilitates the integration of additional datasets when Linked

Data principles are applied. Finally, with the SPARQL Protocol and RDF Query Language an established standard for querying such data is available.

Despite performance improvements of RDF stores in recent years [1], query response times can still be an issue in some use cases. As we will show in Section 5, in a typical e-commerce use case in the tourism domain, searching for resources of a certain application-specific type with a large number of properties, even if they are already materialised, can fail to satisfy performance requirements for such applications. Additionally, such web applications require a predictable and robust performance as customers demand a short response time for any query. From the database point of view, a query such as “Hotels with Wireless LAN” can typically be answered quickly if the required information is readily available, as the query requires a single join. In the given simple use case, the results of the triple pattern `?s rdf:type :Hotel` only have to be joined with those of the triple pattern `?s :hasFeature :WLAN`. Usually RDF stores provide indexes on such information for faster query response times. However, more complex queries on many more graph patterns require many joins. Hence, they increase the difficulty of the query optimisation problem. As a result, for reasonable datasets typical star-like queries on many features such as “Hotels in Europe near a lake with beach, WLAN, restaurants, childcare, 24h reception, suitable for families” exceed acceptable query response times.

In order to make sure performance requirements can be met while providing the flexibility of an RDF data model, we investigate how we can improve query processing with custom index structures. In this paper we present an index extension based on a bitset data structure representing the most commonly used properties of a certain type of resource. We will show that performance requirements can be fulfilled, s.t., the query response time is robust and predictable considering typical queries.

The paper is organized as follows. After discussing related work in Section 2 and outlining our concept (Section 3), we describe an index extension API for the Virtuoso column store in Section 4.1. In Section 4 the implementation of the bitset index plugin is described. Thereafter, an evaluation on production data (Section 5) is presented. This paper will be closed with a discussion and conclusions in Sections 6 and 7.

## 2. RELATED WORK

Most triple store implementations provide general indexing on the stored assertions, such as AllegroGraph [2], Virtuoso [5], or RDF-3X [9] which uses compressed clustered B+ trees. For instance, Hexastore [10] builds an index of all six permutations of triple elements, however, not all of them are strictly required as other stores limit the amount of indexes to those patterns typically used in queries. While this works reasonably well for queries on a single property, for many more complex queries only the combination of triple patterns is selective. Such queries require a more sophisticated join selectivity prediction. One approach is to precompute join cardinalities for literal combinations and improve scans using sideways-information-passing [8]. Still, query response times depend on estimating the correct join ordering.

Another approach is based on replacing the typically used B+ trees with more compact index representations in order to reduce disk access and process queries in memory. BitMat [4] uses a compressed bit matrix structure to represent the entire dataset and introduces a join query processing method on it. Although more generic and fast on "star join" queries, the on-disk size of the transformed data is very large. The general applicability of the approach also results in delays due to loading the required index structures for a query, which can be very large for non-selective patterns. Custom approaches using bitsets could avoid this. While  $k^2$ -trees provide a high compression [3], queries are not processed universally faster for all types of joins. Also, keeping an index that represents the entire dataset in-memory usually reduces vertical scalability options.

## 3. CONCEPT

In order to substantiate the proposed approach, we will first define a use case, including derived requirements, and outline related data structures both of the stored RDF data and our proposed index structure. Finally, we give a summary of the system architecture and required components.

### 3.1 Use Case

The approach proposed in this paper is developed in a tourism e-commerce use case, which aims at improving the user experience when searching for hotels matching several specific user needs. Each hotel has a unique identifier and is characterised by different attributes, including but not limited to:

- its label
- a description
- the regions it is located in
- entities and classes located nearby
- customer ratings
- a comprehensive list of features
- activities and target groups the hotel is suitable for.

Regions are organised hierarchically, i.e., typical hotels are located in a city, which belongs to a region or administrative divisions which themselves are located within a country and continent. Nearby resources usually are distinct entities, such as a certain point of interest like the Stephansdom in Vienna or a certain lake.

Typical queries are logical conjunctions of such attributes. A SPARQL representation for an example query "Hotel with WLAN near a lake in Germany" is shown in Listing 1. A search process usually consists of picking attributes (more or less selective), selecting a region, and sorting the results. Often the result set size is limited.

Listing 1: Example SPARQL query for use case

```
PREFIX uo: <http://example.org/ontology#>
PREFIX uor: <http://example.org/resource/>
PREFIX uorf: <http://example.org/resource/f/>

SELECT DISTINCT ?s FROM uor: {
  ?s a uo:Hotel;
    uo:hasFeature uorf:56;
    uo:nearby uo:Lake;
    uo:locatedIn uor:DE .
}
```

For search applications, it makes sense to materialise transitive relations and generalisations of this information, i.e., to apply materialisation rules. We implemented several such rules in order to, e.g., derive statements for transitive properties such as `uo:locatedIn`. For instance, the statements `uor:hotel1 uo:locatedIn uor:Austria` and `uor:hotel1 uo:locatedIn uor:Europe` could be derived for an existing statement `uor:hotel1 uo:locatedIn uor:Vienna`.

On top of this dataset, we want to implement a search application that is:

- fast with regards to average query response times
- robust with regards to maximum query response times
- scalable, i.e. with low memory consumption overhead

Since we preprocess the data [6], including the generation of the application-specific index structure, and test them intensively in a staging environment before rolling it out in production, update performance is not a requirement here. These processes and requirements are similar to many other use cases to which the proposed approach could be applied.

### 3.2 Data Structures and Bitset Application

As explained above, the properties of hotels are generally stored as an RDF named graph, with materialised statements added for production use. Geographical regions are represented in a directed acyclic graph, as shown in Figure 1. The use of literals is limited to further specifying respective resources for facilities. In other words, instead of statements like `uor:hotel1 uo:hasFeature "WLAN"` the dataset contains `uor:hotel1 uo:hasFeature uorf:1` and `uorf:1 rdfs:label "WLAN"`.

While the query in Listing 1 requires joins to be processed, we can optimize such queries on single resources using bit arrays. As a first step, we determine the triple patterns that are most appropriate. Usually, we can identify respective statements by their (lack of) selectivity and the access frequency. Here, lack of selectivity means that a triple pattern should be represented in a bit array when it applies to many resources of interest. For example, for the used dataset the triple pattern `?s uo:locatedIn uor:Europe` applies to much more subjects than the more selective pattern `?s uo:locatedIn uor:Vienna`. However, if typical users are only

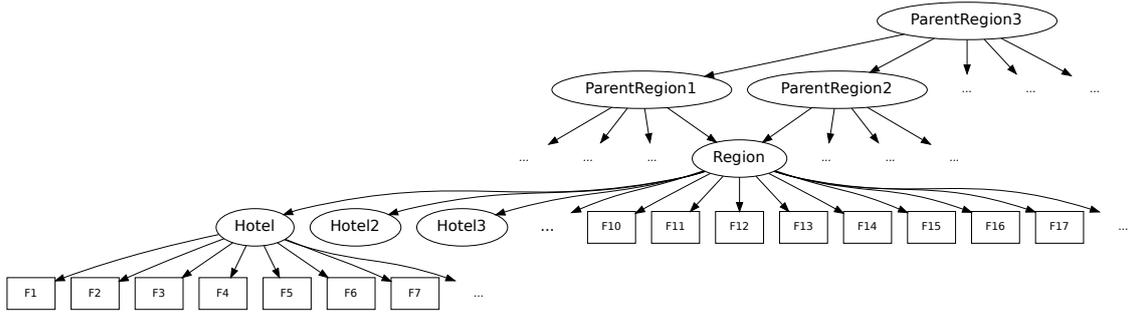


Figure 1: Hierarchical directed graph of regions and hotels

Table 1: Generic bitset structure

Feature $f \in F$	Resource $r \in R$		
	$r_1$	$r_2$	$r_n$
$f_1$	{0,1}	{0,1}	{0,1}
$f_x$	{0,1}	{0,1}	{0,1}

Table 2: Example bitset structure for use case

Feature $f \in F$	Resource $r \in R$		
	hotel1	hotel2	hotel3
$f_1$ : uo:locatedIn uor:Vienna	0	1	1
$f_2$ : uo:hasFeature uor:f56	0	0	1
$f_3$ : uo:nearby uor:Lake	1	0	1

looking for hotels in certain cities, it would be more useful to store the latter triple pattern. The required thresholds have to be set based on a tradeoff between index coverage and index memory consumption. Secondly, we prepare a sorted list of target resources  $r \in R$ , in our use case all hotels ordered by their identifier.

For all respective triple patterns, represented as a *feature*  $f \in F$ , we generate a bitset for each such feature and the respective target resources, as shown in Table 1. Each row represents the matching resources for a certain feature, such as “located in Vienna”. An example bitset for our use case, consisting of 3 hotels and 2 features, is shown in Table 2.

Using such an index structure, processing the joins for basic graph patterns<sup>1</sup> referring to a single resource can be translated into basic bit operations on these bitsets. For example, a query “Hotel in Vienna nearby Lake” processing the intersect of two atomic features can be computed using bitwise AND on the bitsets  $f_1$  (011) and  $f_3$  (101), returning (001) representing that only hotel3 satisfies these conditions.

As an extension to this, a UNION could be computed with bitwise OR. Aggregations could equally be improved because counting bits that are set, i.e., the cardinality of a resulting bitset, is much faster than conventional index lookups. In this paper, we will focus on the performance with regards to basic graph patterns.

### 3.3 System Architecture

<sup>1</sup>Sets of triple patterns, details at <http://www.w3.org/TR/sparql11-query/#BasicGraphPatterns>

The system architecture for using this index extension consists of five primary components:

- Preprocessing stage required for preparing the bitsets and lookup tables for feature and resource identifiers
- Component storing the preprocessed bitsets
- RDF store with index extension capabilities
- Index extension implementation querying the bitset index and computing the bitset operations according to the query
- Search query analyzer, including query parser/rewriter.

In the preprocessing stage, we apply the materialisation rules on the RDF dataset, identify a list of resources we want to index, select the triple patterns that we want to index, and iterate over the triple patterns to build respective bitsets for the indexed resources. These are then stored in a respective component. The search query analyzer translates given free-text queries into basic graph patterns. It uses the lookup tables to identify features stored in the bitset index and generates a respective SPARQL query. Note that this component is work in progress, thus it is not covered in this paper. Finally, the RDF store processes the SPARQL query, using the index extension implementation for handling triple patterns represented in the bitset.

## 4. BITSET INDEX IMPLEMENTATION

We have implemented a prototype of the proposed concept as an extension for the Virtuoso triple store by OpenLink Software. It is one of the fastest RDF stores available according to BSBM<sup>2</sup> and the DBpedia SPARQL benchmark [7] and outperforms other triple stores particularly for larger datasets. In this section we discuss the required index extension API and the implementation of a bitset index extension plugin.

### 4.1 Index Extension API

In order to integrate such a custom index, OpenLink added an Index Extension API to its open-source Virtuoso 7 Fast Track codebase<sup>3</sup>. The API is suitable for using alternate text

<sup>2</sup>Berlin SPARQL Benchmark, latest results available at <http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/results/V7/>

<sup>3</sup><https://github.com/v7fasttrack/virtuoso-opensource/tree/feature/analytics> commit 84d693c

index implementations or specialized search logic or content indices for diverse media types.

An index extension type defines a number of functions for updating and searching data from the index extension. Index extensions can be defined for columns of relational tables or for specific types of RDF literals. We are just interested in the latter in the scope of this paper. The index extension can manage id, string pairs. The id is the internal identifier of an RDF literal, and the string is the string of the literal as returned by the `str(?literal)` SPARQL function.

A registered index extension is used by the `contains` function in SQL or the `bif:contains` predicate in SPARQL. Extra arguments may specify the index extension to use and additional variables to be bound, e.g., for scores.

The index extension defines three data structures:

- The index instance  $I$  (`index_type_t`) – this is a handle on the physical index.
- A cursor  $C$  (`iext_cursor_t`) – this is a handle on a query against the index. A cursor can retrieve a set of matches in multiple batches.
- A transaction  $X^A$  (`iext_txn_t`) – the transaction object represents a set of changes to the content of an index extension instance, given as an argument to functions reading or updating the index extension instance. An index extension transaction will be committed or rolled back together with the containing transaction.

In Virtuoso, an index extension is registered at server startup, typically as part of loading a plugin. A `sys_vt_index` system table associates index extension instances to tables and columns, which is used by the server to call the opening function of an index extension instance. Further extension-specific properties can be configured here.

For using the extension, the `iext_exec` function starts a query against an index extension instance. One query string is provided that can be freely interpreted by the extension, and subsequent calls to the `iext_next` function are expected to produce matches and optional match scores for values found in the index. For query optimisation, an `iext_sample` function should return an estimate of the number of hits and single-threaded CPU time this is expected to take, while the `iext_cost` function gives estimates on the time it takes to return the next match.

## 4.2 Bitset Index Plugin

On top of this API, OpenLink provided a sample implementation. For verifying our proposed concept, we implemented an index extension plugin that processes queries using bitsets. Since we already had created bitsets stored in Memcached<sup>4</sup> using a Spring Batch<sup>5</sup> based preprocessing component, we implemented the following three-layer prototype:

**Plugin (C):** As Virtuoso is implemented in C, we implemented a C plugin based on the sample index extension code.

<sup>4</sup><http://memcached.org/>

<sup>5</sup><http://projects.spring.io/spring-batch/>

**Bitset Wrapper (Java):** In order to use our existing Java implementation for managing bitsets stored in Memcached, we implemented a wrapper around this functionality. The wrapper is invoked by the plugin using Java Native Interface (JNI).

**Bitset Processor (Java):** A basic implementation for serialising and deserialising Java bitsets in Memcached and invoking bitset operations.

The plugin basically consists of the following features. It creates the index instance structure and registers the index with Virtuoso, and it creates a wrapper instance using JNI including its initialisation. During this initialisation, the lookup tables for features and identifier mappings are loaded. Next, it forwards a query string to the Java wrapper. Finally, it looks up the returned result ids and translates them into literal ids used internally by Virtuoso.

The wrapper essentially parses the given query string into a filter structure, which basically is a computing instruction for the actual Bitset Processor implementation. The filter can consist of a combination of intersect, union, and atomic filters. After filtering out deleted resources, the results are returned to the plugin. For instance, the index extension’s query string “`uo:hasFeature uorf:56; uo:locatedIn uor:Vienna`” would be parsed into the structure:

```
IntersectFilter(AtomicFilter( $f_2$ ),AtomicFilter( $f_1$ )).
```

Finally, the Bitset Processor fetches the bitsets for the given features from Memcached and executes the processing instruction, e.g., returning the intersect of the  $f_1$  and  $f_2$  bitsets for the above example. The implementation is proprietary.

## 5. EVALUATION

We will first describe the dataset used for evaluation, the execution environment, selected queries and the benchmarking process before we analyse and discuss the evaluation results.

### 5.1 Dataset

The proposed concept was evaluated using the discussed implementation on a real-world private dataset in the tourism e-commerce domain. The dataset contains 16,106,429 triples in five named graphs for 278,277 resources, which include 71,600 hotels, and 3,022,583 literals.

The bitset created in the preprocessing phase contains 16,507 selected properties, which results in a memory consumption of 63,109,198 bytes according to Memcached “stats” output.

### 5.2 Execution Environment

We used Virtuoso 7.1 sources from the “v7fasttrack” github repository to compile and link the plugin. As a baseline, we tested Virtuoso with the plain dataset, i.e., no special literals and indexes were configured, with 70,000 buffers (824MB memory) on a Virtualbox VM running Debian Squeeze. The virtual machine was configured with 3GB RAM and 2 CPU cores of a quad-core Intel Xeon E31225 at 3.1GHz.

For testing the plugin, we used a separate database copy and Virtuoso configuration. The database contained the additional literals and the index configuration on the system

RDF\_QUAD table. Virtuoso was then configured for using the plugin and having only 60,000 buffers (706MB) to reflect the memory taken by Memcached, which was configured to use at most 256MB of memory.

### 5.3 Evaluation Queries

In order to define test queries, we first examined the dataset w.r.t. selectivity of respective features. An example query for `uo:locatedIn` and a target type `country` would be:

```
SELECT DISTINCT ?o count(?o) FROM uor:
{ ?s uo:locatedIn ?o . ?o a uo:Country }
GROUP BY ?o ORDER BY desc(count(?o))
```

Next, we generated test queries representing a typical search process (Section 3.1) under the following primary assumptions:

- A1** A typical user selects a number of features he would like to have, such as Wireless LAN or a swimming pool.
- A2** A user further narrows the selection by choosing a region and deciding for additional properties, such as the suitability for certain activities like hiking.

Consequently, in order to represent assumption **A1** we generated queries  $q_1$  to  $q_8$  with the schema:

```
SELECT ?s FROM uor: { ?s a uo:Hotel; uo:hasFeature
fs1, fs2, ... fsn }
```

Here,  $f_s$  represents a selected feature. We added additional queries  $q_9$  and  $q_{10}$  extending  $q_8$  to address assumption **A2**:

```
SELECT ?s FROM uor: { ?s a uo:Hotel; uo:hasFeature
fs1, fs2, ... fs8; uo:locatedIn fr }
```

```
SELECT ?s FROM uor: { ?s a uo:Hotel; uo:hasFeature
fs1, fs2, ... fs8; uo:locatedIn fr ; uo:suitableFor fa }
```

Here,  $f_r$  represents a selected region and  $f_a$  a selected activity. In order to evaluate the impact of selectivity of the selected features, we manually compiled two sets of features:

- Selection S1 uses less selective features of similar selectivity, i.e., the queries return many results.
- Selection S2 uses more selective features, i.e., queries return fewer results. For example, query  $Q_{10}$  returns only 4 results with Selection 2 while using the data of the experiment.

In addition to that, we define the following test cases  $T$ :

- A:** Queries  $q_1$  to  $q_{10}$  are processed.
- B:** In addition to test case A, all queries have an additional statement, i.e., an additional join is required which is not executed as a bitset operation. This can indicate whether the approach is flexible with regards to querying features not represented in the bitset.
- C:** In addition to test case A, all queries have an added triple pattern on a decimal literal (popularity) and an "order by" clause on the bound value of this triple pattern. This test indicates whether sorting is handled properly<sup>6</sup>.

<sup>6</sup>Since the implicit ordering of the bitset results is not neces-

- D:** In addition to test case A, the queries end with `LIMIT 107`.
- E:** In contrast to test case A, the projection will be an aggregate, i.e., `SELECT ?s` is replaced with `SELECT count(?s)` in all queries.

For testing the index extension plugin, we rewrote all queries accordingly. For example,  $q_{10}$  was rewritten as follows:

```
SELECT ?s FROM uor: { ?s uo:bitsetFeatures ?f . ?f
bif:contains "uo:hasFeature fs1, ... fs8; uo:locatedIn
fr ; uo:suitableFor fa" option (score ?sc, "index",
"memcache") }
```

### 5.4 Benchmarking Process

In addition to the defined test cases, the benchmarking process further distinguishes between cold and warm database, i.e., whether a warm-up  $W$  has been performed on the database to fill the buffers prior to executing the query. The actual benchmarking is executed as in Algorithm 1.

**Algorithm 1** Benchmarking process

---

```
for  $i = 1, i++,$  while  $i \leq 5$  do
  for all Queries  $q_n \in Q$  do
    for all Test cases  $t \in T$  do
      for all Feature selections  $s \in S$  do
        for all Warm-up states  $w \in W$  do
          for all Execution env.  $e \in E$  do
            benchmark()8
          end for
        end for
      end for
    end for
  end for
end for
```

---

For all queries, test cases, feature selections, warm-up states and execution environments (*test configuration*), each test is executed 5 times (test run). For each test run, the Virtuoso server is started with the respective configuration of the execution environment. A first timestamp is taken before the respective query for the test configuration is executed. After the query has been processed, a second timestamp is taken and the duration is written to a logfile. Finally, the Virtuoso server is stopped again. No other user-level processes are running during benchmarking and the evaluation has been executed at night to avoid falsification of results.

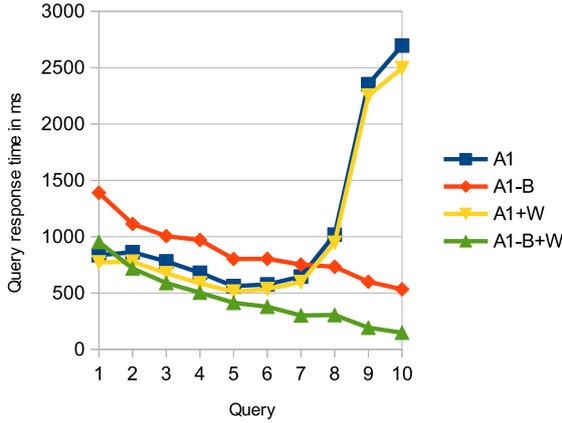
### 5.5 Evaluation Results

For each test configuration, we selected the median of the 5 test run query response times in order to eliminate outliers. For each benchmark X, we compare the following results:

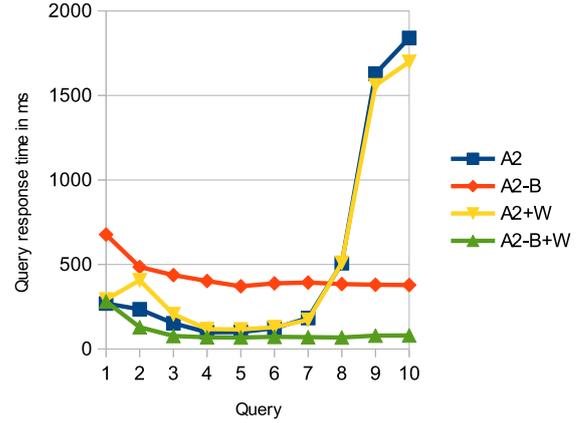
- X are query response times using ordinary SPARQL query (without bitset)
- X-B are query response times using the bitset index extension plugin

sary retained in the SPARQL query result, the current implementation does not allow for performance benefits here since explicit ordering is still required.

<sup>7</sup>Since the bitset index extension is not aware of further selections in the query, the result set limit can not be applied early on, so no performance benefits should be expected.



(a) Benchmark A1 using less selective features.



(b) Benchmark A2 using more selective features.

Figure 2: Benchmarks on test case A

- X+W are query response times using ordinary SPARQL query, with warm-up
- X-B+W are query response times using the bitset index extension plugin, with warm-up<sup>9</sup>

Figure 2 shows query response times for less and more selective features for test case A. The queries that don't use the bitset index (A{1,2} and A{1,2}+W) show acceptable query response times for up to 7 (A1) or 8 (A2) features, but are significantly slower for more complex queries, especially  $q_9$  and  $q_{10}$  which introduce additional predicates. Using the bitset index, the more complex queries can be computed significantly faster. For the current implementation, a warm-up is strictly required since the JNI-based integration appears to add a constant  $\sim 350$ ms latency for the first query. With warm-up, the bitset plugin outperforms a Virtuoso without this index extension except for  $q_1$ . With regards to selectivity, using more selective features (A2) generally reduces response times, and the bitset index plugin can further improve response times because it has to invoke less reverse lookup queries for matching resources.

With regards to the additional test cases, using additional joins (B) for features not represented in the bitset shows a severe relatively constant penalty. Both B1 and B2 show response times of more than 1 second because the added triple pattern `uo:nearby uo:Sea` has a low selectivity. Still, using the bitset index generally results in slightly better response times and helps to avoid the very high query response times for the complex queries  $q_8$ ,  $q_9$  and  $q_{10}$ .

When sorting the results explicitly using `ORDER BY` (test case C), the bitset actually increases the performance benefit, in particular for more selective features in C2, despite the additional popularity triple pattern `?s uo:popularity ?p`.

Finally, when using `LIMIT` in the SPARQL query (test case D) the bitset index introduces some processing overhead

<sup>9</sup>Warm-up for bitset index extension was extended with a generic query for the reverse lookups of URI literals to Virtuoso literal identifiers.

Table 3: Median query response times for different benchmarks (X) in ms

Benchmark	Test configuration			
	X	X-B	X+W	X-B+W
A1	1,100	870	1,014	<b>450</b>
A2	514	430	521	<b>99</b>
B1	2,519	1,712	2,200	<b>1,353</b>
B2	1,642	1,476	1,594	<b>1,109</b>
C1	1,159	805	1,134	<b>403</b>
C2	656	397	658	<b>77</b>
D1	616	536	620	<b>122</b>
D2	521	393	485	<b>66</b>
Median	1,349	981	1,242	<b>582</b>

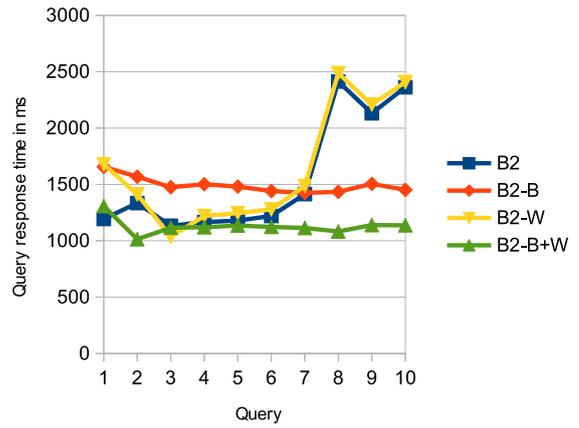
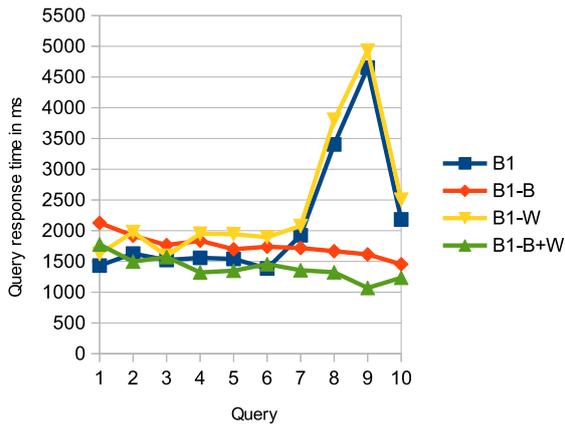
Table 4: Maximum query response times for X in ms

Benchmark	Test configuration			
	X	X-B	X+W	X-B+W
A1	2,696	1,390	2,496	<b>953</b>
A2	1,840	677	1,699	<b>281</b>
B1	4,927	1,917	4,651	<b>1,572</b>
B2	2,487	1,568	2,413	<b>1,139</b>
C1	2,607	1,132	2,634	<b>721</b>
C2	1,848	489	2,043	<b>132</b>
D1	2,361	632	2,311	<b>156</b>
D2	1,902	439	1,801	<b>74</b>
Maximum	3,148	1,268	3,023	<b>851</b>

for less selective features so ordinary SPARQL queries are slightly faster up to query  $q_5$ . Still, for highly selective features and, again, for more complex queries, bitset-enabled queries still perform much better and much more consistent.

## 6. DISCUSSION

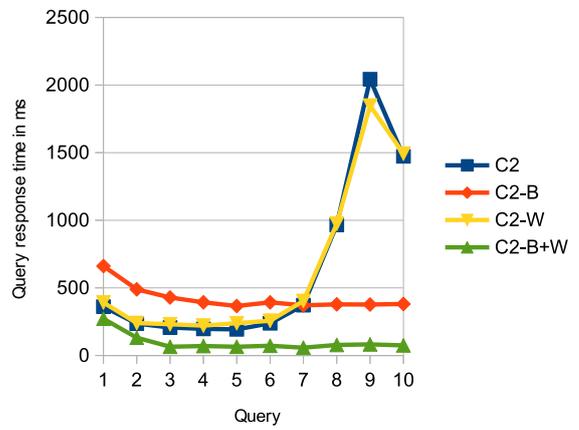
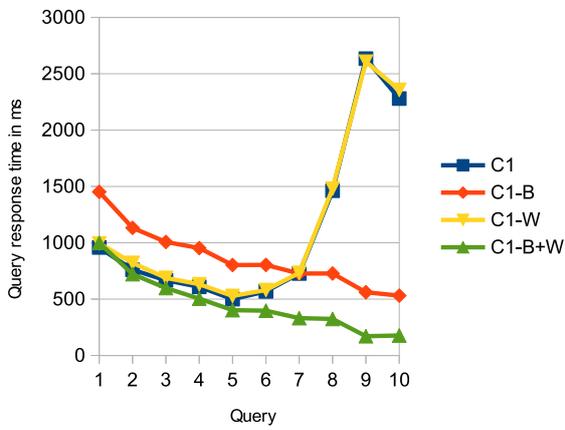
The evaluation shows that a bitset-based index extension can significantly improve SPARQL query response times in a real-world scenario. With regards to the use case specified in Section 3.1, bitsets improve average query response times in all test cases (Table 3). The implementation is robust since there is an upper bound for the query response times, and



(a) Benchmark B1 using less selective features.

(b) Benchmark B2 using more selective features.

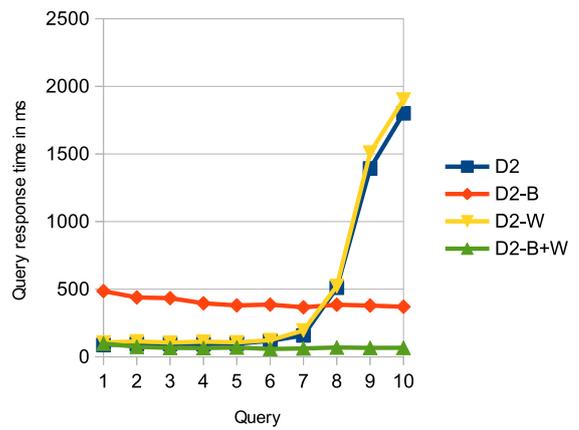
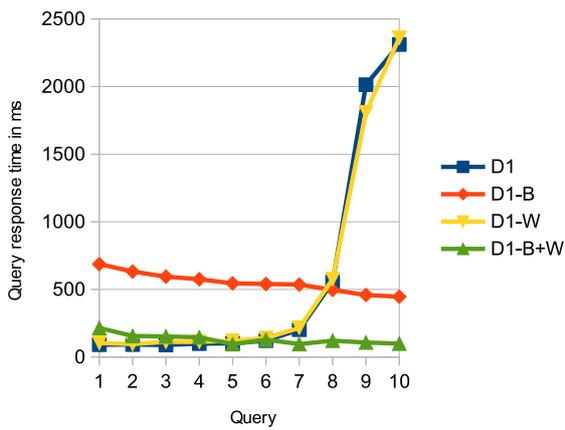
Figure 3: Benchmarks on test case B (additional join)



(a) Benchmark C1 using less selective features.

(b) Benchmark C2 using more selective features.

Figure 4: Benchmarks on test case C (sorting)



(a) Benchmark D1 using less selective features.

(b) Benchmark D2 using more selective features.

Figure 5: Benchmarks on test case D (limit)

more complex queries consistently result in lower response times (Table 4).

With regards to vertical scalability, the memory consumption overhead due to bitsets is relatively low (less than 64MB in the evaluated use case). Less frequently used and highly selective features can flexibly be used via additional triple patterns in the SPARQL query. Also, compressed bitsets can be used for large datasets to store bitsets more efficiently. Horizontal scalability can easily be achieved by deploying the plugin and the bitset store on all nodes.

While the dataset used for evaluation is moderate-sized, we expect the impact of a bitset index to be much more significant on larger datasets. Since we tested with a production dataset used in a real-world application, the approach already shows an impact despite only indexing 71.600 resources (subjects) and using a prototypical implementation with high overhead.

Since the initial bitset data for hotels was based on long values as resource identifiers, in this use case the plugin further transforms the returned long ids into respective literals. In the prototype, we created respective literals representing each indexed URI using the SPARQL query in Listing 2. Thus, the plugin translates the hotel id long values into URI literals by adding a configured prefix. For instance, it will append the id 1 to the prefix `uor:hotel` for `hotel1`, resulting in the URI `http://example.org/resource/hotel1` for the example use case. This prototypical implementation can certainly be optimised.

Listing 2: SPARQL query generating special literals

```
PREFIX uo: <http://example.org/ontology#>
PREFIX uor: <http://example.org/resource/>

INSERT INTO uor:
{ ?u uo:bitsetFeatures ?literal }
WHERE { graph uor: {
  ?u a uo:Hotel .
  BIND(STRDT(str(?u), uo:bits) AS ?literal )
} }
```

Some potential improvements, such as limiting the result set early on in test case D, can provide additional benefits. For example, the additional lookup overhead could be circumvented by letting the plugin know that there are no further triple patterns outside the index query and it can already apply the limit, i.e., put only the given number of results into the cursor.

## 7. CONCLUSIONS

Despite recent advances in query processing on RDF data, we have shown that complex queries can result in relatively long query response times even on a moderately sized real-world dataset. In this paper, we have discussed a concept for using bitsets as an index extension for common properties of certain resources. We have implemented a prototype plugin using an index extension API developed for the Virtuoso triple store and compared its performance against ordinary SPARQL queries in different test cases.

Query processing using the bitset index was consistently faster with regards to average response times. More importantly for enterprise search applications, we were able

to achieve robustness with predictable maximum query response times. The proposed solution has low memory overhead and we expect the approach to be very scalable. Hence, to the best of our knowledge we presented the first solution for scalable and highly performant search solutions w.r.t. the e-commerce vertical.

The presented index extension API in Virtuoso is not limited to bitset indexes. We currently also work on an Elastic-Search plugin for improved and much more flexible full text search fully integrated with SPARQL query processing.

## 8. ACKNOWLEDGMENTS

This work was supported by a grant from the European Union's 7th Framework Programme (2007-2013) provided for the project GeoKnow (GA no. 318159). We would like to thank Mirko Spasic from OpenLink Software for contributing to the implementation.

## 9. REFERENCES

- [1] Berlin SPARQL benchmark results. <http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/results/index.html#tdb>. (visited on 12-June-2015).
- [2] J. Aasman. Allegro graph: RDF triple database. Technical report, Technical report. Franz Incorporated, 2006. url: <http://www.franz.com/agraph/allegrograph/> (visited on 10/14/2013)(cited on pp. 52, 54), 2006.
- [3] S. Álvarez-García, N. R. Brisaboa, J. D. Fernández, and M. A. Martínez-Prieto. Compressed k2-triples for full-in-memory RDF engines. *arXiv preprint arXiv:1105.4004*, 2011.
- [4] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix Bit loaded: a scalable lightweight join query processor for RDF data. In *Proceedings of the 19th WWW*, pages 41–50. ACM, 2010.
- [5] O. Erling and I. Mikhailov. RDF support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media*, pages 7–24. Springer, 2009.
- [6] A. Garcia-Rojas, D. Hladky, M. Wauer, A. Both, R. Isele, C. Stadler, and J. Lehmann. The GeoKnow Generator Workbench: An integration platform for geospatial data. WasABi workshop at ESWC. May 2015.
- [7] M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. DBpedia SPARQL benchmark–performance assessment with real queries on real data. In *The Semantic Web–ISWC 2011*, pages 454–469. Springer, 2011.
- [8] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 627–640. ACM, 2009.
- [9] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.
- [10] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.