Extended Characteristic Sets: Graph Indexing for SPARQL Query Optimization

Marios Meimaris[1,2], George Papastefanatos[2] and Ioannis Anagnostopoulos[1]

[1] University of Thessaly, Greece
[2] ATHENA Research Center, Greece
{m.meimaris, gpapas}@imis.athena-innovation.gr
janag@dib.uth.gr

Abstract. SPARQL query execution in state of the art RDF engines depends on, and is often limited by the underlying storage and indexing schemes. Typically, these systems exhaustively store permutations of the standard three-column triples table. However, even though RDF can give birth to datasets with loosely defined schemas, it is common for an emerging structure to appear in the data. In this paper, we introduce a novel indexing scheme for RDF data, that takes advantage of the inherent structure of triples. To this end, we define the *Extended Characteristic Set* (ECS), a schema abstraction that classifies triples based on the properties of their subjects and objects, and we discuss methods and algorithms for the identification and extraction of ECSs. We show how these can be used to assist query processing, and we implement *blinkDB*, an RDF storage and querying engine based on ECS indexing. We perform an experimental evaluation on real world and synthetic datasets and observe that blinkDB consistently outperforms the competition by a few orders of magnitude.

1 Introduction

The Resource Description Framework[3] (RDF) and SPARQL[4] are W3C recommendations for representing and querying graph data on the web. In recent years, the Web of Data has been established as a vast source of data from diverse domains, such as biology, statistics, finance, and health. As these data become larger and wider in range, complex queries start to emerge, calling for improvements in the performance of RDF storage and querying engines.

In the case of indexing and query processing, traditional approaches often rely on permutating a single table with three columns, representing the subject, predicate and object (SPO) of a triple, in order to store triples with different relative orderings. For example, the high-performance store RDF-3x uses all six permutations of the SPO table, namely SPO, SOP, PSO, POS, OSP, and OPS, and maintains interesting orders on the index attributes in order to allow for as

[3] https://www.w3.org/RDF/
[4] https://www.w3.org/TR/sparql11-overview/

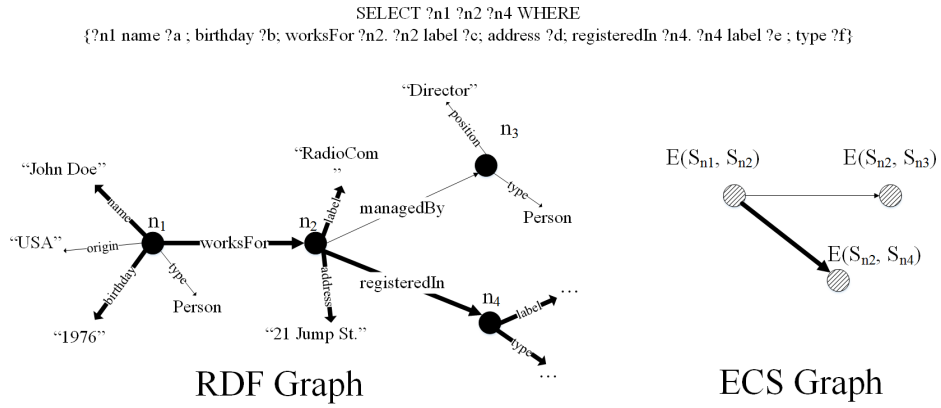


Fig. 1: A set of triples (left) and their ECS graph (right). A multi-chain-star query pattern is shown in the top, with its match in the graphs marked in bold.

many merge joins as possible in a given query plan. In a similar way, the open source version of Virtuoso 7.2 relies on full and partial permutations, also catering for named graphs. Query planning and execution on these systems rely on the *data independence assumption*, which ignores any inherent structure in the data. Thus, optimizers mostly rely on first-level statistics, such as the number of distinct triples with a particular property, and heuristic estimations on join cardinalities. While these techniques are efficient for evaluating queries with small amounts of unbound variables and short paths in the data, their performance degrades when adding complex, multi-join query patterns with potentially low selectivity. Their shortcoming is more evident in Table 1, where we present the running times from the execution of two queries requiring multiple joins in the underlying data[5] on the Reactome dataset and the LUBM100 dataset using three widely used RDF query engines, namely RDF-3x, Virtuoso Opensource 7.2, and Jena TDB 3.0.1. Even though the datasets are relatively small (∼16m triples in Reactome, ∼17m triples in LUBM100 with transitive closure), they fail to produce results fast.

Specifically, these approaches tend to be problematic when answering queries that contain long paths (*chains*) in the data and descriptive star patterns around the chain nodes, i.e., queries with an abundance of subject-object, and subject-subject joins, which we call *multi-chain-star* queries herein. Such an example is shown at the top of Fig. 1, while its evaluation on the RDF graph is marked with bold edges at the left of the figure. In fact, these types of joins are the most frequent in SPARQL queries, making up for more than 90% of all query patterns in empirical studies [2].

SPARQL query optimizers depend on traditional methods for providing good query plans, including the use of data statistics and cardinality estimation for ordering triple patterns. The assumption of data independence imposes a risk of

[5] Query Q9 from the LUBM experiments and Q8 from the Reactome experiments

Table 1: Runtimes in seconds

	blinkDB	RDF-3x	Virtuoso 7.2	Jena TDB
Reactome	0.016	4.7	8.1	15.2
LUBM	0.23	8.2	timeout	timeout

propagating errors in the planning process, especially in joins that reside deeper in the query plan. This can result in the creation of plans with large intermediate results between joins.

To overcome these limitations, we present a novel indexing scheme, the *ECS index*, that aims to accelerate query processing for conjunctive queries with multi-chain-star patterns. The ECS index is based on the notion of *Extended Characteristic Set* (ECS), which is a schema abstraction that carries the benefit of partitioning triples based on the triples' inherent structure. This partitioning requires less storage overhead by not relying on excessive replications, and decreases the effects of bad estimates by quickly filtering triples that collectively participate in multiple joins, in one single scan. In order to define the ECS, we extend the notion of *Characteristic Sets* [8], that typically represents the inherent structure of nodes in an RDF dataset, in order to represent *the structure of triples* in a given dataset. We discuss how ECSs, can be used for the construction of an inverted index that maps ECSs to collections of triples, and we present our approach on evaluating conjunctive SPARQL queries with multi-chain-star patterns based on this index. In short, the contributions of this work are the following:

- We define *Extended Characteristic Sets* (ECS) as a schema abstraction for collections of triples, based on the work in [7],
- we present an algorithm for efficient extraction of ECSs in RDF datasets, as well as extraction of ECS graphs that represent paths in the data,
- we present an algorithm for query processing over the ECS index,
- we implement the approach in *blinkDB*, a reference engine for ECS indexing and query processing that handles conjunctive multi-chain-star queries, and
- we evaluate the performance of blinkDB on one synthetic and two real datasets with respect to storage and querying, and we compare it with three widely used systems. To this respect, we find that blinkDB consistently outperforms the competition by 1-3 orders of magnitude, both in the case of selective and unselective queries.

This paper is organized as follows. Section 1 serves as an introduction. Section 2 provides preliminary definitions for RDF and SPARQL, and defines Extended Characteristic Sets (ECSs) and ECS graphs. Section 3 presents algorithms for extracting characteristic sets and extended characteristic sets, and constructing the index. In section 4, we discuss query processing based on this index, and in section 5 we present an experimental evaluation on synthetic and real-world data. Finally, section 6 presents related work, and section 7 concludes the paper.

2 Preliminaries

In this section, we introduce the main concepts and notation used in the paper.

RDF and SPARQL. RDF models facts about entities in a triple format consisting of a subject s, a predicate p and an object o. A collection of triples is usually represented as a directed labelled graph with subjects and objects being the nodes, and predicates being the edges of the graph. Formally, let I, B, L be infinite, pairwise disjoint sets of IRIs, blank nodes and literals, respectively. Then, an RDF triple t is represented by a triple $(s, p, o) \in (I \cup B) \times (I) \times (I \cup B \cup L)$ and a collection of triples $\{t_1, t_2, \ldots, t_n\}$ is represented by an RDF graph, in which a node $n \in T = (I \cup B \cup L)$ and an edge $e \in I$ [10].

Following this notation, a SPARQL query defines a set of triple patterns of the form $(T \cup V) \times (I \cup V) \times (T \cup V)$, where V is the set of variables that can be bound to T. Triple patterns can be recursively combined via AND, $OPTIONAL$ and $UNION$ operators.

Extended Characteristic Sets (ECS). One of the benefits of RDF is that it is loosely structured; one can extend and modify the schema at will, by adding or deleting new triples for properties and classes. For example, the nodes n_1 and n_3 of Fig.1 are of the same type (e.g., *foaf:Person*); however n_1 has a *name*, an *address* and an employer (with the *worksFor* property), while n_3 does not have an employer, but has a *position*. Thus, it is not easy to consider n_1 and n_3 to be of the same type in a low-level data-oriented approach.

Neumann and Moerkotte [8] introduced the notion of *characteristic sets* as a means to capture the underlying structure of an RDF dataset. A characteristic set CS identifies node types based on the set of properties they emit. Formally, given a collection of triples D, and a node s, the characteristic set $S_c(s)$ of s (or simply S_s) is:

$$S_c(s) = \{p \mid \exists o : (s, p, o) \in D\} \tag{1}$$

and the set of all S_c for a dataset D is:

$$S_c(D) = \{S_c(s) \mid \exists p, o : (s, p, o) \in D\} \tag{2}$$

Characteristic sets provide a node-centric partitioning of an RDF dataset, based on the structure of a node, and they have been used effectively in the characterisation of joins and cardinality estimation [8]. However, they cannot capture the different relationships of nodes in a dataset, i.e., how triples, instead of nodes, can be partitioned based on their characteristics. For this reason, we introduce the *Extended Characteristic Set* (ECS), the *triple-level analogue* of the node-based characteristic set. An ECS captures the inherent schema of triples, based on the properties of their adjacent nodes, i.e., the characteristic sets of the subject and the object. Formally, given a triple $t = (s, p, o)$, the ECS $E_c(t)$ of t, is an ordered set containing the characteristic sets of s and o:

$$E_c(t) = \{S_c(s), S_c(o)\} \tag{3}$$

Table 2: Observed cardinalities of properties, CS and ECS in synthetic and real data.

	LUBM	BSBM	WordNet	Reactome	EFO	GeoNames	DBLP
#properties	18	40	64	65	80	36	26
#CS	14	44	779	112	520	851	95
#ECS	68	374	7250	346	2515	12136	733

which is shortly denoted as $E_{s,o}$. The set of all ECS in D is given by:

$$E_c(D) = \{E_c(t) \mid \exists p : (s, p, o) \in D\} \tag{4}$$

An ECS helps to quickly identify the largest superset of graph patterns that contain a star pattern around s, a star pattern around o, and a link between s and o. An example multi-chain-star graph with three ECSs is shown in Fig. 1, where nodes n_1 and n_2 are present in the same triple t_{n_1,n_2} as subject and object respectively, and descriptive star patterns are present for each of the two nodes. In a similar manner, an ECS is formed between n_2 and n_3, as well as n_2 and n_4. Note that, by definition, if n_1 and n_2 are related with multiple properties, these are part of the same ECS, which is defined by all the properties from n_1 to n_2, along with the rest of the properties emitting from n_1 and n_2.

Each triple $t(s, p, o)$ corresponds to one and only one ECS, specifically $E(S_s, S_o)$. The upper bound for $|S_c(D)|$ is the distinct number of subject nodes, i.e., nodes that emit property edges, however, the existence of an inherent structure in RDF data makes the distinct set of Characteristic Sets that appear in real-world data small [8]. Similarly, the maximum number of ECSs in a given dataset is $|S_c(D)|^2$, that is, one ECS for each pair of characteristic sets. However, in practice, we observe that triples are partitioned in tractable numbers of ECSs, as it can be seen in Table 2 for several real-world and synthetic datasets.

ECS Graphs and ECS Query Graphs. ECSs can be further combined in directed graphs to capture transitive relationships between characteristic sets in an RDF dataset. This is useful for representing paths between types of triples, while at the same time maintaining the star-shaped sub-graphs around the subject nodes. An ECS graph is a directed graph $G_E = (V_E, E_E)$ where $V_E \in E_c(D)$, and $E_E \in (V_E \times V_E)$ are the nodes and edges of the graph, respectively. A node in G_E corresponds to an ECS of the RDF dataset. A directed edge $e = (E_{n_1,n_2}, E_{n_2,n_3})$ exists when there is at least one triple t_a with ECS E_{n_1,n_2}, whose object is the subject of a triple t_b with ECS E_{n_2,n_3}. In other words, an edge between two ECSs represents sets of triples that form subject-object joins in the dataset. An ECS chain c_E is a path formed by consecutive edges between ECSs in an ECS graph. An example of an ECS graph for a given RDF graph is depicted in Fig. 1, where the ECSs as combinations of the characteristic sets S_{n_1}, S_{n_2}, S_{n_3} and S_{n_4} of nodes n_1, n_2, n_3 and n_4 are shown. Consider a multi-chain-star query q listed at the top of the figure. Its evaluation can be seen with bold edges. The query defines a chain from n_1 to n_4 through n_2, along with star patterns around n_1, n_2 and n_4. Notice that in the equivalent ECS graph, the query is reduced to one single edge between $E(S_{n_1}, S_{n_2})$ and $E(S_{n_2}, S_{n_4})$, and thus, one join operation between two ECSs.

An ECS graph provides a suitable abstraction for traversing multi-chain paths in the RDF graph efficiently, without spending computational resources in the execution of subject-subject self-joins that usually have low selectivity and generate large intermediate results [12]. Instead, it relies on treating subject-object joins as first-class citizens. Incoming queries can be evaluated on top of the dataset's ECS graph, by (i) quickly assessing the existence of one or more ECS sub-graphs that are super-sets of the query graph, and (ii) finding a minimal set of triples that contribute to the evaluation of the query. The first point is important for determining whether large, complicated queries have non-empty results, while the second point allows us to access and process a small subset of the data that is sure to contribute to the query processing stage. The latter point is of particular interest when handling complicated queries of long paths with many unbound variables, and helps avoid large intermediate results.

Given the above, it is necessary to derive the ECSs out of a query graph and map them to the dataset's ECS graph space. An incoming query pattern q is mapped to the ECS query graph Q_E based on the identification and extraction of the extended characteristic sets of the triple patterns in q. Formally, a small modification to the ranges in the original definition of characteristic sets [8] is needed in order to allow variable nodes to instantiate characteristic sets as well. Specifically, a characteristic set $S_c(s_q)$ of a node s_q in a query pattern is allowed to be defined over unbound, as well as bound instances of s_q, and unbound or bound instances of predicates and objects in the triple patterns with s_q as subject, i.e., $S_c(s_q) = \{p_q \mid \exists o_q : (s_q, p_q, o_q) \in q\}$ $, (s_q, p_q, o_q) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$.

3 ECS Index

In this section, we provide methods and algorithms for efficient extraction of CS and ECS from datasets, and present how the ECS structure is used for triple storage and indexing. We use our reference implementation, blinkDB, in order to discuss implementation choices and techniques.

Triple Representation. In blinkDB, triples are modelled as three consecutive integers of 0-4 bytes, one for each triple component, namely subject, predicate and object, as is typically done in RDF stores [3,9,15]. The id assignment is performed during initial parsing of the input RDF, and the references are stored in memory during the loading phase until they are flushed to disk in bulk based on our index structures. While in memory, the triples are modelled as integer arrays of size 4. The first three positions of the array are used to hold the subject, predicate, and object ids, while the last position is used to denote each triple based on the CS of its subject.

Notice that we reserve 4 bytes (32-bit integers) for each component during the loading phase, instead of maintaining an encoding of varying size. This verbosity will become useful later, as we use this structure in order to sort the triples both by subject and CS. Furthermore, it is relatively affordable, as even for 1 billion distinct ids, the system needs 4 GB of RAM while loading the data. In any case,

this structure is held off-heap and is backed by a memory mapped file in order to avoid disastrous overflows during data loading.

A dictionary is built and maintained during parsing, that holds values for the node and predicate ids (IRIs), as well as the literals. IRIs are compressed based on their prefixes in order to avoid tedious duplications of strings that occur frequently in the data. The dictionary is then used during query parsing, in order to map bound values from the query to the actual RDF data in the system, as well as when building the result.

Characteristic Set Extraction and CS Index. A characteristic set $S_c(s)$ is a a set of common properties $p_1, \ldots, p_n$ that emit from a set of subject nodes. The set of all CS $S_c(D)$ can be easily retrieved with a linear scan on the triples of a dataset [4],[8]. We sort the triples by subject and construct a new CS each time a new combination of properties is found in a subject. As we are also interested in maintaining a mapping between triples and characteristic sets, we take care to map each triple to a characteristic set during this iteration, based on the characteristic set of the subject node. In the same iteration, we construct the *CS index* by mapping each triple to its corresponding CS. The *CS Index* is a key-value set, in which the key is the set of properties comprising a CS and values are the triples mapped to this CS. The *CS Index* partitions all triples based on their subject's CS and allows us to easily evaluate properties in star patterns. For example, in Fig. 1, the CS S_{n_1} of n_1 is made of properties *name*, *origin*, *birthday*, *type* and *worksFor*. The *CS Index* is also used for extracting ECSs and building the ECS index by combining pairs of CSs, as will be discussed. An example instantiation of the CS index can be seen in Fig. 2, where *Jake* and *Hugh* have the same properties. First, we keep a characteristic-set-to-triples index, namely the *CS index*, on disk for evaluating attributes in star patterns. In order to reduce the number of disk writes during the loading phase, we first retrieve all the triples of each ECS, then write them in the CS index. This is easily achieved by sorting the triples again, this time by their characteristic set, and scanning all triples of each characteristic set. When the scan encounters the next characteristic set, the triples are flushed on disk using the characteristic set's id as key. Second, the upcoming extraction of the unique ECSs relies on joining the retrieved characteristic sets based on their triples' attributes, as will be discussed in the next section The algorithm is presented in Algorithm 1. In blinkDB we implement CS keys as bitmaps, where each bit corresponds to a property in D (e.g., for k properties in D, we construct a bitmap of length k) and a bit of 1 denotes the participation of this property to a CS. This is useful for fast subset checking during the query preprocessing phase, as will be discussed.

Extended Characteristic Set Extraction and ECS Index. The next step is to extract the extended characteristic sets, and build the *ECS index*. A naive way of extracting the ECSs is to perform a subject-object self-join on the whole dataset, scan the resulting rows and create a new ECS for each different combination of the subjects' and objects' CSs. A more efficient way is to take advantage of the previously computed CS Index for calculating ECSs.

Algorithm 1 *extractCharacteristicSets*

Input: *triples*: A $N \times 4$ table of ids, where N is the number of triples in the input. The first three columns are used for subject, predicate and object ids, and the fourth column is used for CS ids.

Output: *csMap*: An inverted index, with CS ids as keys, and sets of triples as values.

```
1: sort(triples) by subject
2: properties ← new Set()
3: previousSubject ← triples[0][0]
4: lastIndex ← 0
5: for each i = 1; i ∈ triples do
6:     subject ← triples[i][0]
7:     if previousSubject ≠ subject then
8:         cs ← newCharacteristicSet(csId, properties)
9:         for each j = lastIndex; j < i; j++ do
10:            triples[j][3] ← csId
11:        csId++
12:        properties.clear()
13:    properties.add(triples[i][1])
14:    previousSubject ← subject
15: sort(triples) by CS
16: triplesToAdd ← newSet()
17: lastCS ← triples[0][3]
18: for each i ∈ triples do
19:    if lastCS ≠ triples[i][3] then
20:        csMap.put(lastCS, triplesToAdd)
21:    triplesToAdd.add(triple[i])
22:    lastCS ← triples[i][3]
    return csMap
```

Specifically, an ECS $E_{1,2}$ comprises the characteristic sets S_1 and S_2 of the subjects and objects of all triples pertaining to this $E_{1,2}$.

Specifically, instead of suffering the cost of a self-join on the whole body of triples, we utilize the CS Index and iterate through all *pairs* of characteristic sets looking for subject-object joins in chunks of triples, remebling a block nested loop. Recall that the CS index maps characteristic sets to sets of triples based on the characteristic set of the subject. Each time we encounter a successful subject-object join between two triples, we construct a new ECS based on the CSs of the triples' subjects. At the same time, we store the mapping between the triples and their ECS, thus building an *ECS Index*. In other words, given two characteristic sets S_1 and S_2, and two sets of triples T_1 and T_2, whose subjects belong to S_1 and S_2 respectively, the subject-object join between T_1 and T_2 will be non-empty when there exist triples that belong in S_2, whose subjects are objects in triples of S_1. We can then store the extended characteristic set $E(S_1, S_2)$ along with references to its subject and object CSs, as well as the triples contained in it.

An *ECS Index* is a key-value set, in which the key is an ordered pair of CSs, and the value is a set of triples pertaining to this ECS, i.e., triples whose

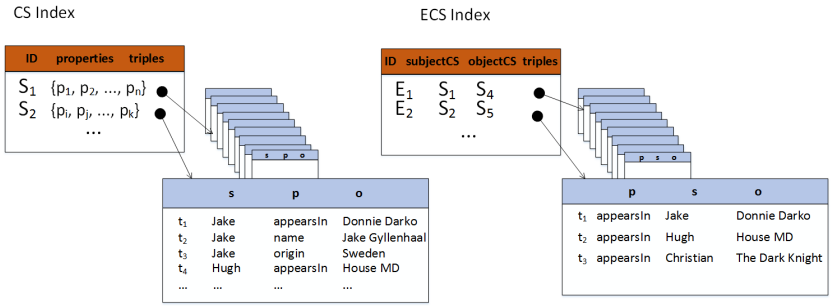


Fig. 2: Example instantiation of the CS (left) and the ECS (right) indexes. The CS contains a set of properties $p_i..p_k$, while the ECS is a composition of a subject CS and an object CS.

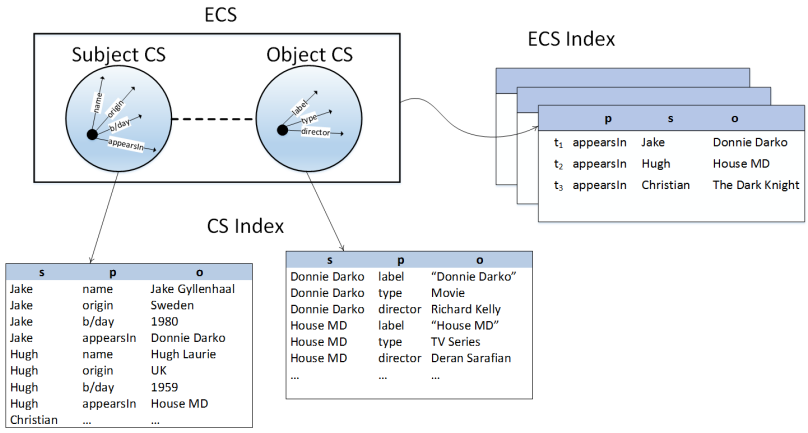


Fig. 3: Our index structures with their metadata and links to triple sets.

subject and object CSs comprise the ECS key. In contrast to the *CS Index* that partitions all triples in a dataset, the *ECS Index* partitions triples pertaining to a valid ECS, i.e., whose subject and object have non-empty CSs. These are triples that are mostly used in multi-chain query patterns with successive subject-object joins. In contrast to the *CS Index* that partitions all triples in a dataset, the *ECS Index* partitions triples pertaining to a valid ECS, i.e., whose subject and object have non-empty CSs. These are triples that are mostly used in multi-chain (e.g., successive subject-object joins) query patterns. An example entry of the ECS Index with its subject and object CSs, as well as its corresponding triples can be seen in Fig. 2 and 3. In the example, the CS index is depicted on the left, where S_2 points to a collection of triples the subjects of which have the properties defined in S_2. Similarly, the triples mapped to an ECS E_2 that has S_2 as its subject CS are shown on the right.

For all characteristic set pairs, keeping the triples of $S_{c,left}$ and mapping them to $E(S_{c,left}, S_{c,right})$ will iteratively result in the ECS index.

Algorithm 2 *extractExtendedCharacteristicSets*

Input: $csMap$ An inverted index that maps characteristic sets to sets of triples based on the characteristic set that is defined by a triple's subject node.
Output: $ecsMap$ An inverted index that maps extended characteristic sets to sets of triples. **Output:** $ecsLinks$ A set of adjacency lists with links between the retrieved ECSs.

```
1: ecsMap, subjectCSMap, objectCSMap ← new Map()
2: for each S_i ∈ csMap do
3:     added ← newSet()
4:     for each S_j ∈ csMap do
5:         triples ← Π_{t_i}(S_i ⋈|_{s−o} S_j), with t_i ∈ csMap.get(S_i)
6:         if triples.size()! = 0 then
7:             ecs ← newECS(S_i, S_j)
8:             ecsMap.put(ecs, sort(triples))
9:             subjectCSMap.get(S_i).add(ecs)
10:            objectCSMap.get(S_j).add(ecs)
11: %Find links between ECSs%
12: for each S_i ∈ Π_{S_i}(objectCSMap ⋈ subjectCSMap) do
13:     for each ecs_left ∈ objectCSMap.get(S_i) do
14:         for each ecs_right ∈ subjectCSMap.get(S_i) do
15:             ecsLinks.get(ecs_left).add(ecs_right)
    return ecsMap, ecsLinks
```

The algorithm for building the ECS index is shown in Algorithm 2. The algorithm takes as input a CS Index and outputs an ECS index and an ECS graph in the form of adjacency lists. It iterates through pairs of S_i, S_j of CSs (lines 2-4), joins their contents (line 5) and if the result is not empty, it creates a new ECS entry and puts the triples sorted in the PSO order (lines 6-8). This ordering is useful for early filtering of triples with properties not in the query pattern. When all pairs with S_i as subject CS have been checked, we take care to put any *orphan* triples, i.e., triples that belong to S_i but were not in the join results with any inner S_j, into a new ECS with $\emptyset$ as its object CS (lines 14-20). For this reason, we maintain the *added* set and drop it after each outer iteration. After retrieving the ECSs, the algorithm finds directed links between ECSs (lines 11-15). To achieve this, we first populate the $subjectCSMap$ and $objectCSMap$ that link CSs to ECSs based on their position in the ECSs (lines 9-10). Then, we look for CSs that are both objects and subjects in different sets of ECSs, and we link these together. The resulting adjacency list represents the *ECS graph*, and is stored as part of the indexing scheme in blinkDB. It becomes essential in the query preprocessing stage, were an incoming query is matched to existing ECS paths in the data. **Auxiliary indexes and statistics.** blinkDB uses two auxiliary indexes in order to assist the pre-processing stage of query evaluation. Specifically, we build a property index map for each ECS, that holds information on the first appearance of a given property in the ECS triples. Recall that we

sort triples by property and subject. Given that the amount of properties in an ECS is generally small (1-4 properties for each ECS in our experiments), having precomputed the first occurrence of a property is useful for avoiding logarithmic searches in large collections of triples during query evaluation. Furthermore, we retrieve edges between ECSs in order to be able to traverse the ECS graph using standard graph traversal algorithms. The algorithm for extracting edges is based on finding ECSs that exhibit subject-object joins on the CS level. This is shown in Algorithm X TODO. Finally, the cardinality of triples in each ECS is computed during the loading phase. Moreover, we store the cardinality of distinct properties in the triples of each ECS. These statistics are useful for the query planning process.

4 Query Processing

In this section, we present how query processing is performed on top of the ECS Index. Our goal is to to employ the derived index structures in order to reduce the size of scans, number of joins, and amount of intermediate results when evaluating SPARQL queries with multi-chain-star graph patterns, even though the approach works for simple query patterns as well. An overview of the query processing stage is shown in Fig. 5. Given an incoming query, we first identify the set of characteristic sets around the chain variables, i.e., S_x, S_y, S_z, and S_w for $?x$, $?y$, $?z$, and $?w$ respectively (Fig. 5a). Then, the query ECSs $Q_{x,y}, Q_{y,z}, Q_{z,w}$ are extracted, and the chain, formed between them, is identified (Fig. 5b). Finally, each query ECS is matched to the ECS index and the triples are joined to output the result (Fig. 5c). Note that Fig. 5 shows a query, for which the ECS graph consists of a single chain pattern. For multiple chain patterns, the processor evaluates them individually and joins them on their common attributes in the last step. In the following, we discuss these steps in detail.

Query parsing and ECS query graph extraction. Incoming queries are first converted to ECS query graphs by the query parser. This is achieved by first exracting the characteristic sets of the query's nodes, then applying Algorithm 2 to find the ECSs on the query pattern, and create adjacency lists between the query ECSs. We then perform depth-first search on the adjacency lists of all extracted query ECSs, in order to identify chains in the ECS query graph. This results in a set of chain patterns $c_1 \dots c_n$ that are joined on one or more chain nodes. After this step, we remove chains that are fully contained in other chains.

Matching of query ECSs to the ECS index. Matching of the query ECSs to the ECS index is performed through a depth-first traversal of the ECS graph. For each ECS in the query chain, we traverse edges of the ECS graph in search of potential matching ECSs. By performing depth-first search on the ECS edges, it is guaranteed that consecutively matched ECSs over the query are actually linked in the data, because each reached ECS will be a child of the preceding one. The condition to be met during traversal is that, for an ECS E_i in the data, and a query ECS Q_j, the properties of E_i must be a superset of

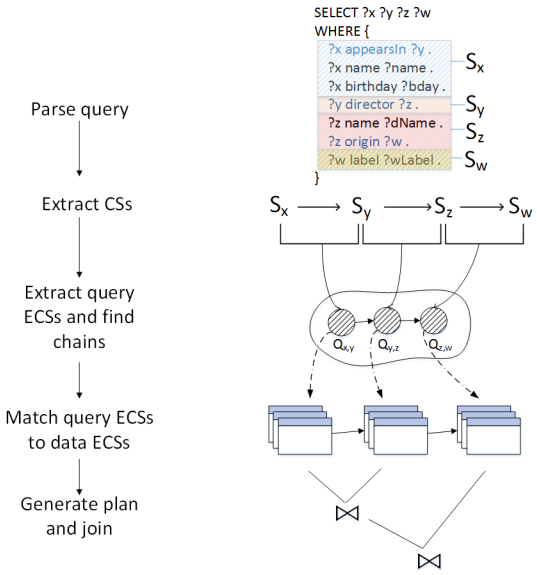


Fig. 4: Query processing for a chain pattern of three query ECSs.

the properties of Q_j, with respect to their subject and object characteristic sets. When this is the case, E_i is added to the matches of Q_j, as it will participate in the joining process. In essence, this subset check ensures that the matched ECSs will contain triples, the nodes of which include all properties in the star patterns around the chain nodes. The output of this process is a set of ECS chains in the ECS graph that match the query's ECS chains. This is shown in Algorithm 3. The algorithm works by initiating a depth-first traversal of the ECS graph from all starting ECS edges (i.e., keys in the adjacency list), for all chain patterns c_i (lines 2-4). As the traversal goes deeper in the graph, a pointer to the query chain is also advanced, for each edge that is traversed in the ECS graph (lines 14-15). When the query chain reaches the end or the traversal finds a cycle, the result is returned.

An example query with one ECS, and its matches in the data, is shown in Figure 6. The query defines an ECS Q that links the characteristic set of $?x$ to the characteristic set of $?y$. In the bottom right of the figure, the ECSs of the two data sub-graphs, namely E_1 and E_2, as well as the query ECS Q are shown. The query's properties are successfully matched to E_1 and E_2 as all properties in Q form a subset of the properties in E_1 and E_2. Notice that E_1 and E_2 have different sets of properties, hence, they define separate sub-graphs (sets of triples) in the data. However we are only interested in the properties in Q, which are indeed contained in both E_1 and E_2, thus making the triples in E_1 and E_2 relevant with respect to the query.

Query planning. The query planner decides the join execution order for the various sets of triples corresponding to the matched ECS chains of the previous

Algorithm 3 *matchQueryChainToECSIndex*

Input: $ecsEdges$**:** The ECS adjacency list
Input: $c(q_0 \dots q_{n-1})$**:** A chain of query ECSs
Output: $ecsMatches$**:** A chain of ECS sets that match the ECSs in c

```
1: ecsMatches ← newMap()
2: for each q ∈ c_i do
3:     for each e ∈ ecsEdges.keySet() do
4:         matchDataPatterns(e, ecsEdges, c_{0...n−1}, ecsMatches)
   return ecsMatches
5: procedure MATCHDATAPATTERNS(ecsEdges, c_{q0...qn−1}, ecsMatches)
6:     if q.subjectCS.properties ⊈ e.subjectCS.properties
7:         OR q.objectCS.properties ⊈ e.objectCS.properties
8:         OR propertyIndex.get(e).contains(q.property) then
9:         return null
10:    if visited(e) OR c.size == 1 then
11:        return ecsMatches
12:    visited.add(e)
13:    ecsMatches.get(q).add(e)
14:    for each e_child ∈ ecsEdges.get(e) do
15:        matchDataPatterns(e_child, ecsEdges, c_{q1...qn−1}, ecsMatches)
```

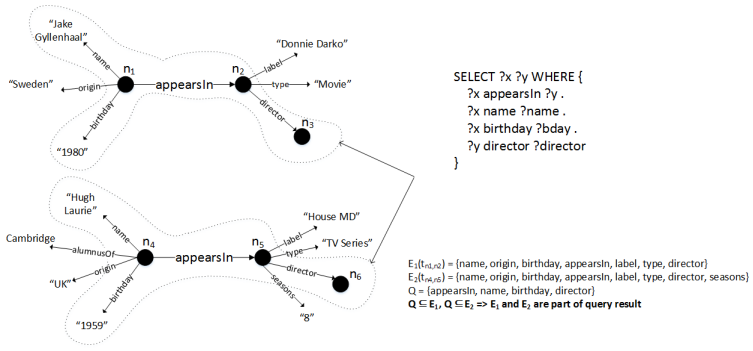


Fig. 5: Matching two ECS in the data to an incoming query.

step. The planner distinguishes between the outer ordering (evaluation of different chains) and the inner ordering (evaluation of a specific chain). The outer ordering is useful for filtering out triples as early as possible based on the common attributes of the different chain patterns. The inner orderings help reduce intermediate results between ECSs, that do not contribute to the final result. blinkDB uses a combination of heuristics and cardinality statistics to compute the outer and inner orders of execution.

The (outer) order of chains is simply derived by the size of the chains. As larger chains tend to be more selective, we evaluate those first. For queries with unbound star patterns, i.e., triple patterns around the chain nodes without bound variables in the query, like the ones in Fig. 5 and Fig. 6, the execution

order does not matter, because all paths defined by the matched ECS chains will eventually be traversed.

For queries with bound nodes, we order ECSs based on the number of bound elements in their triple patterns. The intuition is that the larger the number of bound nodes, the more selective the pattern. Filtering is then performed by looking up the CS index, finding characteristic sets that contain the bound values, and filtering out the matched ECSs that do not contain these characteristic sets at the corresponding position, i.e., as subject or object. If ordering by bound variables fails or is inconclusive, the next employed heuristic is to order ECSs by the cardinality of their mapped triples. In order to take advantage of the matches from the ECS graph, we take care to progressively evaluate ECSs in a chain by expanding on an existing join order, either by subject or object. That is, each ECS is evaluated on the hashed results of the previous ECS.

Query execution. Each query chain pattern is executed individually, by looking up the ECS index and joining the triples of each ECS of the matched chains. Multiple chain patterns are joined in the final step of the execution using hash joins on their common attributes, the join tables of which are created dynamically during the evaluation of individual chains. It is worth noting that the execution of different chain patterns is parallelizable and we can also pipeline information between threads in order to filter out triples during parallel executions. In blinkDB, the engine takes advantage of multiple cores by paralellizing the execution of different chains, however, such an optimization is not possible when evaluating an individual chain. Note that, execution of a chain pattern does not take into account the star pattern variables when joining consecutive ECSs. Retrieval of the attributes in the star pattern of the subject and/or object of an ECS is instead achieved when retrieving the ECS from disk, by performing a merge-join between the ECS's triples and the triples of the subject/object CS from the CS Index. In fact, a merge-join is possible because the CS Index maintains the interesting order of the subject node. However, this will not happen in the case where none of these variables are part of the query projection. Note that, execution of a chain pattern does not take into account the star pattern variables when joining consecutive ECSs. Retrieval of the attributes in the star pattern of the subject and/or object of an ECS is instead achieved when retrieving the ECS from disk, by performing a merge-join between the ECS's triples and the triples of the subject/object CS from the CS Index. In fact, a merge-join is possible because the CS Index maintains the interesting order of the subject node. However, this will not happen in the case where none of these variables are part of the query projection. In this case, as is the case for the queries of Figures 5 and 6, the restriction for the chain nodes to emit the bound properties is already enforced by the ECS definition, thus avoiding a significant number of subject-subject self-joins and large intermediate results in complex queries.

5 Evaluation

We have conducted an extensive experimental evaluation on *blinkDB* with both synthetic and real-world data, and a comparative study with three widely adopted RDF engines, namely *RDF-3x*, *Virtuoso opensource 7.2* and *Jena TDB 3.0.1*. RDF-3x is a high-performance, RISC-style RDF engine that supports efficient storage and querying by using optimal query plans and efficient data compression. It is commonly cited as a highly efficient graph query engine that minimizes disk reads and writes by using optimal query plans and efficient data compression. Virtuoso is an RDF quad store, built on top of a relational engine. As a Java-based competitor, we chose Jena TDB, a native RDF store commonly used in the deployment of semantic web applications. All experiments were performed on a server with Intel i7 3820 3.6GHz, running Debian with kernel version 3.2.0 and allocated memory of 16GB. For Virtuoso, we used the recommended tuning parameters given by Openlink, for Jena TDB we used the default stats-based optimizer. The aim of the experiments is to assess the performance of blinkDB in *data loading*, *query execution* and finally *scalability* with synthetic data of increasing sizes. Thus, our experiment metrics are: *query execution time*,*loading time* and *disk storage size*.

Implementation. BlinkDB is an open-sourced project[6], implemented using Java 1.8 and the mapDB[7] library, a high-performance key-value engine with drop-in replacements for sets, such as hash tables. blinkDB uses mapDB for object serialization/deserialization and disk I/O on native Java objects. This is also the default way of serializing and deserializing ECS and CS objects. We also utilize the readily available caching mechanism of mapDB. In blinkDB, triples are modelled as three integers of 0-4 bytes (for subject, predicate and object), as it is typically employed in RDF stores [3,9,15]. For triple serialization and persistence, blinkDB uses byte arrays and random access files and writes all data in a single binary file, similar to RDF-3x and Virtuoso. The triples mapped to an ECS/CS are serialized as a contiguous array of integers, and can be retrieved in bulk using the ECS/CS id as key from the ECS/CS index. This format carries the benefit of being easily partitioned, while reducing disk reads to the number of matched ECSs per query. A dictionary holds values for the (compressed) node and predicate ids, as well as the literals. The dictionary is then used during query parsing, in order to map bound values from the query to the actual RDF data in the system, as well as when building the result. In this reference implementation, blinkDB supports only conjunctive SPARQL queries.

Datasets. For synthetic data, we use the *Lehigh University Benchmark* (LUBM) data generator to create datasets of increasing sizes, from 15 (LUBM10) to 370 (LUBM2000) million triples. LUBM uses an academic ontology of universities, with entities for departments, courses, members of faculty and so on. Since blinkDB does not support inferencing, we extended the LUBM generator to add all superclasses of an instance's class, in order to generate the transitive

[6] All code and queries are available in `http://github.com/mmeimaris/blinkDB`
[7] www.mapdb.org

Table 3: Size on disk (GB) and loading times (minutes)

	# triples input		blinkDB		RDF-3x		Virtuoso		Jena TDB	
			size	time	size	time	size	time	size	time
LUBM2000	370m	54.2	8.12	68	16.54	58	14.6	45	42.2	195
Reactome	16m	2.8	0.71	3	1.07	2	0.91	2	1.76	7
Geonames	172m	18.8	8.24	81	12.48	34	8.56	27	16.7	41

closure of the subclass relationships, as well as the *memberOf* and *hasAlumnus* properties. For our real-world experiments, we chose the *Reactome*[8] dataset, which contains information about biological pathways, and is rich in long paths with branching components, and *Geonames*[9], an ontology of geographical features that contains a diverse schema of varying properties (i.e., large number of CS/ECS) among the same types of entities, as shown in Table 2. Geonames maintains a rich graph structure as there is a heavy usage of hierarchical area features on a multitude of levels.

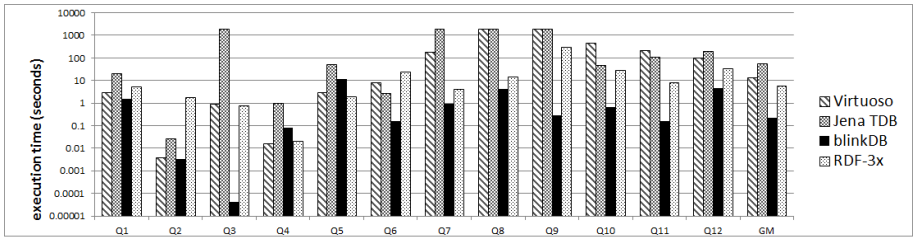
Loading. The size in GB and loading time in minutes for the two real datasets and LUBM2000 can be seen in Table 3. Overall, blinkDB exhibits the lowest space overhead for the input data. This is a derivative of the low degree of data replication imposed by ECS indexing. However, blinkDB suffers from longer loading times than RDF-3x and Virtuoso, because of the added complexity of retrieving the inherent schema of nodes (i.e., CS index), and triples (i.e., ECS index). Especially for Geonames, the loading time is significantly longer, because of the large number of ECSs present in the dataset.

Query performance - LUBM. We use 7 of the original 14 queries (2, 3, 4, 8, 10, 11 and 12), slightly modified with more unbound variables and larger characteristic sets, and we define 5 additional queries with chain-star patterns. The queries are ordered by complexity[10], and Q1-8 are highly selective, while Q9-12 are low in selectivity. Their runtimes can be seen in Fig. 7(a) along with their geometric mean (GM). The actual GM for Virtuoso and Jena is equal to, or greater than the maximum depicted in the figure, as we do not show running times above 30 minutes, or timed-out queries. As shown, blinkDB consistently outperforms the rest, with its geometric mean improving the competition by 1 and 2 orders of magnitude. Especially in the case of queries with complex patterns (Q7-12), blinkDB is better by several orders of magnitude, while Virtuoso and Jena suffer several timeouts after 30 minutes. For Q3, which does not yield any results, the preprocessor cannot match the query graph to any ECS chains in the data, and thus does not perform any joins, giving blinkDB an advantage of up to 4 orders of magnitude compared with RDF-3x and Virtuoso (Jena timed out). In the more selective queries Q4 and Q5, blinkDB is outmatched by the rest of the systems (except Jena), because it does not have permutated indexes that can help to quickly filter out triples that do not contribute to the solution.

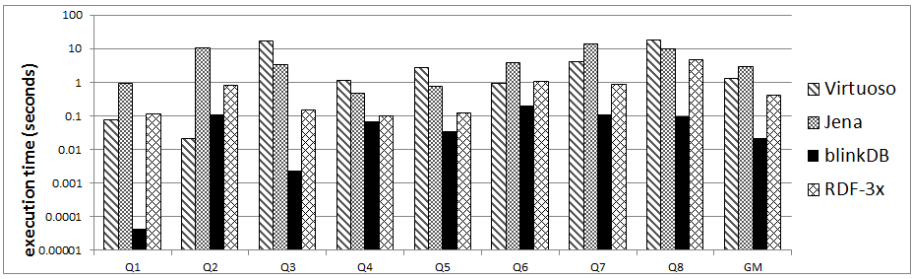
[8] http://www.ebi.ac.uk/rdf/services/reactome

[9] http://www.geonames.org/ontology/documentation.html

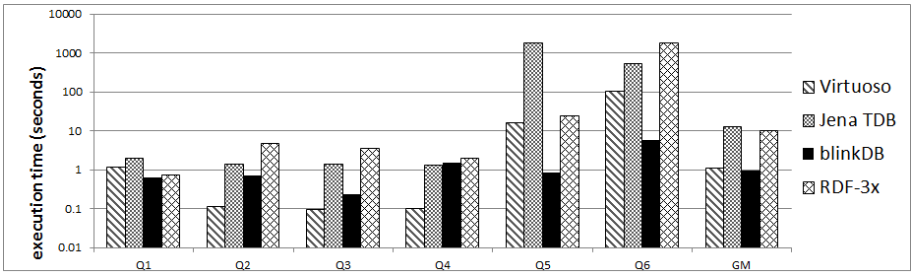
[10] Calculated as the product of (#triple patterns)×(#chains)



(a) Query runtimes for the LUBM dataset



(b) Query runtimes for the Reactome dataset

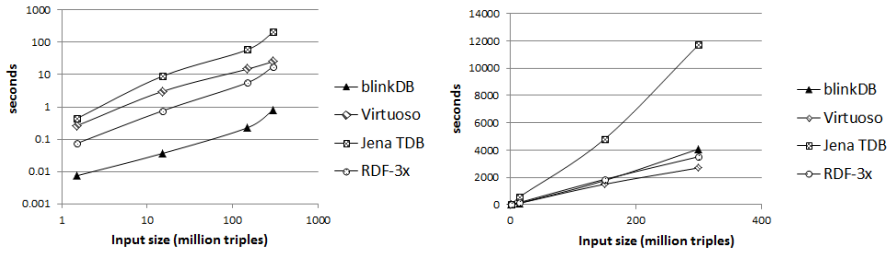


(c) Query runtimes for the Geonames dataset

Fig. 6: Query runtimes in seconds

Instead, it has to scan all of the triples of the matched ECSs, and thus spend more time to produce less results.

Reactome. We use 8 queries for the Reactome dataset, with varying lengths and amounts of chain patterns, with 1-3 chains and 3-6 query ECSs. The results can be seen in Fig. 7(b). Again, blinkDB consistently outperforms the rest for all types of queries. Even though the dataset is relatively small, the complexity of the data can lead to queries with non-trivial patterns. This is evident by the relatively large number of ECSs (346). For the queries with the lowest selectivity (Q6, Q7 and Q8), blinkDB improves the competition by at least one order of magnitude. This provides an intuitive insight that the nature of ECS indexing facilitates the evaluation of complex query patterns by isolating smaller subsets of the data that contribute to the result, and thus decreasing the intermediate results that would be present in traditional indexing paradigms. Instead, an ECS



(a) (b)

Fig. 7: Query execution (a) and dataset loading (b) for increasing sizes of LUBM

graph matches a query to smaller and more relevant subsets of the data, and reduces the number of self-joins and the cardinality of intermediate results.

Geonames. We define 6 queries with increasing selectivity and pattern complexity. The results can be seen in 7(c). While blinkDB outperforms the rest in all queries but Q4, the improvements are not at the same scale with the previous datasets. Genoames has over 10,000 different ECSs, thus invoking costly disk reads even for ECSs with small cardinalities. In fact, this reflects a drawback in the ECS indexing approach, where partitioning of the triples by their associated ECS can become a bottleneck when the partitioning is volatile with respect to the triple cardinality of each ECS. In any case, blinkDB improves the competition by one order of magnitude overall, based on the observed GM.

Scalability. We have experimented with increasing input sizes of LUBM, starting from 15M triples, up to 370M triples. In Fig. 8, we report the GM of Q1-Q12 (a), and the loading time (b) for all four systems, in log-log scales. The query performance of blinkDB scales *linearly* and retains its relative difference by 1-3 orders of magnitude with the rest of the systems for all input sizes. Loading also appears to scale linearly with respect to input size, however, due to the ECS extraction of the loading phase, blinkDB is outperformed by Virtuoso and RDF-3x as the input increases. In any case, our experiments indicate that the methods presented herein are indeed scalable for larger input sizes.

6 Related Work

Related work in RDF stores places systems in three generalized directions, namely *triples tables*, *property tables*, and *vertical partitioning*. A triples table has three columns, representing the subject, predicate and object (SPO) of a triple. This technique usually replicates data in different orderings of SPO in order to facilitate sort-merge joins. For example, RDF-3X [9] and Hexastore [13] build tables on all six permutations of SPO, while RDF-3x also employs indexes for binary and unary projections of the original SPO data. Similarly, Virtuoso [5] uses a large 4-column table for quads, and a combination of full and partial indexes, while Jena TDB relies on three permutations. These methods have

been established and in fact work well for selective queries with small numbers of joins, however, they tend to degrade with increasing dataset sizes, large numbers of unbound variables and decreasing selectivity, as the required index scans become larger. Furthermore, the storage overhead can become a limiting factor when scaling for very large datasets.

Property Tables [14,1] is a technique that places data in one or multiple tables, the columns of which correspond to the properties of the dataset. Each row identifies a subject node and holds the value of each property in the corresponding cells. However, this causes extra space overhead for null values in cases of sparse properties for a given class[1]. Also, it raises performance issues when handling complex queries with many self-joins, as the amounts of intermediate results tend to be significant, especially for increasing sizes of datasets [6].

Vertical partitioning is a technique that partitions the data in tables with two columns. Each table corresponds to a property in the data, and each row to a subject node [1]. This approach provides great performance when evaluating queries with bound objects, but tends to suffer when the sizes of the tables have large variations in size [11]. TripleBit [15] is an RDF store that broadly falls under the vertical partitioning type, but uses bitmaps to store the occurrence or absence of predicate-object pairs in a table where rows represent subject nodes. In TripleBit, the data is vertically partitioned in chunks per predicate. While this approach is efficient for reducing the amount of replication in the data, it suffers from the same problems as property tables. Furthermore, it does not take into account the inherent schema of the triples in order to simplify the evaluation of complex query patterns, as is the case for blinkDB.

Characteristic Sets have been introduced as an abstraction of node types, and they have been used for provision of better estimates when computing join cardinalities [8], and implemented in the RDF-3X high performance triple store. In this regard, Brodt et al [4] present their approach on how the SPO index can be used to identify Characteristic Sets and assist query processing by decreasing the number of subject-subject joins that are common in star patterns. We follow this approach in blinkDB with the use of the CS index, that is essentially an SPO permutation that is partitioned among all CSs of a dataset. Our notion of Extended Characteristic Set is in fact inspired by the Characteristic Set, but focuses on triples, rather than nodes. To the best of our knowledge, this is the first work to use such a structure for RDF indexing and query processing.

7 Conclusions and Future Work

In this paper, we have presented ECS indexing, a novel indexing scheme for RDF data, and discussed its implications on SPARQL query processing. To this end, the notions of *extended characteristic sets*, and *ECS graphs*, were introduced, along with methods and algorithms for ECS retrieval and querying. The above functionality has been implemented in blinkDB, a reference implementation that was used to perform experiments against three widely used RDF storage and querying engines. The experimental evaluation conducted herein has shown that

blinkDB outperforms the state of the art approaches, especially for answering complex query patterns with low selectivity. As future work, we will extend our approach in order to cater for hierarchies of ECSs, as well as query optimizations, and we will study efficient update mechanisms under the ECS context.

References

1. D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422. VLDB Endowment, 2007.
2. M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An empirical study of real-world sparql queries. *arXiv preprint arXiv:1103.5043*, 2011.
3. M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix bit loaded: a scalable lightweight join query processor for rdf data. In *WWW*, pages 41–50. ACM, 2010.
4. A. Brodt, O. Schiller, and B. Mitschang. Efficient resource attribute retrieval in rdf triple stores. In *CIKM*, pages 1445–1454. ACM, 2011.
5. O. Erling and I. Mikhailov. *Virtuoso: RDF support in a native RDBMS*. Springer, 2010.
6. M. Janik and K. Kochut. Brahms: a workbench rdf store and high performance memory system for semantic association discovery. In *ISWC*, pages 431–445. Springer, 2005.
7. M. Meimaris and G. Papastefanatos. Double chain-star: an rdf indexing scheme for fast processing of sparql joins. In *EDBT*, pages 668–669. ACM, 2016.
8. T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *ICDE*, pages 984–994. IEEE, 2011.
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. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In *ISWC*, volume 4273, pages 30–43. Springer, 2006.
11. L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for rdf data management: not all swans are white. *VLDB*, 1(2):1553–1563, 2008.
12. P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. Boncz. Heuristics-based query optimisation for sparql. In *EDBT*, pages 324–335. ACM, 2012.
13. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *VLDB*, 1(1):1008–1019, 2008.
14. K. Wilkinson and K. Wilkinson. Jena property table implementation, 2006.
15. P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. Triplebit: a fast and compact system for large scale rdf data. *VLDB*, 6(7):517–528, 2013.