Technical Report

Graph Indexing for SPARQL Query Optimization with Extended Characteristic Sets

Marios Meimaris

m.meimaris@imis.athena-innovation.gr
(1) Institute for the Management of Information Systems
Research Center Athena,
Greece

George Papastefanatos

gpapas@imis.athena-innovation.gr
Institute for the Management of Information Systems
Research Center Athena,
Greece

Nikos Mamoulis

nikos@cs.uoi.gr
University of Ioannina,
Greece

Ioannis Anagnostopoulos

janag@ucg.gr
Department of Computer Science and Biomedical Informatics
University of Thessaly,
Greece

Extended Characteristic Sets: Graph Indexing for SPARQL Query Optimization

Marios Meimaris
ATHENA Research Center
m.meimaris@imis.athena-innovation.gr

George Papastefanatos
ATHENA Research Center
gpapas@imis.athena-innovation.gr

Nikos Mamoulis
University of Ioannina
nikos@cs.uoi.gr

Ioannis Anagnostopoulos
University of Thessaly
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 *axonDB*, an RDF storage and querying engine based on ECS indexing. We perform an experimental evaluation on real world and synthetic datasets and observe that axonDB outperforms the competition by a few orders of magnitude.

I. Introduction

The Resource Description Framework[1] (RDF) and SPARQL[2] 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 permuting a single table with three columns, representing the subject, predicate and object (SPO) of a triple, in order to store the triples with different relative orderings. For example, the high-performance store RDF-3x [11] 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 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 numbers of unbound variables and short paths, their performance degrades when adding complex, multi-join query patterns with potentially low selectivity and large intermediate results between joins. This shortcoming is more evident in

TABLE I: Runtimes in seconds

	axonDB	RDF-3x	Virtuoso 7.2	TripleBit
Reactome	0.016	4.7	8.1	2.6
LUBM	0.23	8.2	timeout	timeout

Table I, where we present the running times from the execution of two queries requiring multiple joins[3] on the Reactome and the LUBM100 datasets using three state of the art RDF query engines, namely RDF-3x, Virtuoso Opensource 7.2[6], and TripleBit[20]. Even though the datasets are relatively small (~16m triples in Reactome, ~17m triples in LUBM100 with transitive closure), the engines 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; its evaluation on the RDF graph is marked with bold edges at the left of the figure. In fact, these types of joins are very frequent in real world data, making up for 35% of all joined patterns[4] in empirical studies [2]. Recent approaches [12], [14], [19], [21] attempt to speed up the query performance over very large datasets by distributing data and scaling out joins into multiple nodes; still, their query processing, although distributed, relies on the data independence assumption, thus moving the aforementioned limitations to a distributed setting.

In this paper, we focus on the limitations of the core modules (i.e., indexing schemes) of RDF systems to answer complex queries even in relatively small datasets. We present a novel indexing scheme, the *ECS index*, that aims at accelerating query processing for conjunctive queries with multichain-star patterns. The ECS index is based on the notion of *Extended Characteristic Set* (ECS), which captures the inherent structure of subject-object relationships in an RDF graph. An ECS corresponds to a different type of subject-object relationship by comprising the different types of triples (i.e., properties) of the adjacent nodes. We construct an index that maps a triple to an ECS, and we present an efficient approach that evaluates conjunctive SPARQL queries with multi-chain-star patterns based on this index. An example RDF graph with four ECSs is shown in Fig. 1. ECS E_1 corresponds to the type of relationship between the nodes
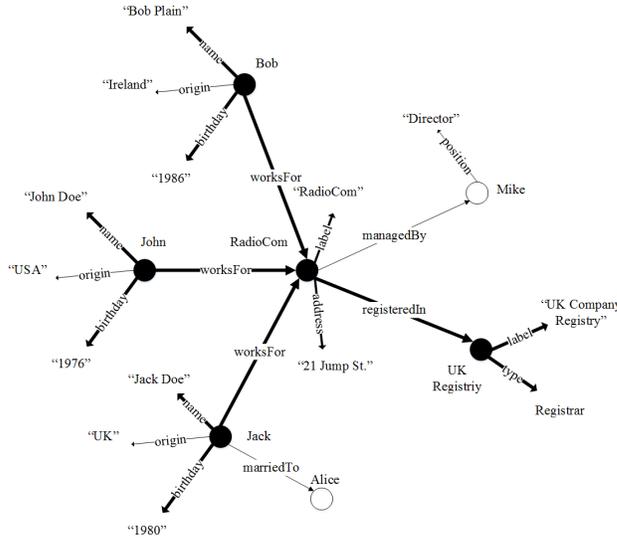
---

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

[4]In the same study, 60% of the join types are subject-subject joins, thus subject-subject and object-subject joins make up for 95% of all join types.

SELECT ?n1 ?n2 ?n4 WHERE
{?n1 name ?a ; birthday ?b; worksFor ?n2. ?n2 label ?c; address ?d; registeredIn ?n4. ?n4 label ?e ; type ?f}

Resource	Characteristic Set
John	S1 = {name, origin, birthday, worksFor}
Bob	S1 = {name, origin, birthday, worksFor}
Jack	S2 = {name, origin, birthday, worksFor, marriedTo}
RadioCom	S3 = {address, label, managedBy, registeredIn}
Mike	S4 = {position}
UK Registry	S5 = {label, type}

Triple	Extended Characteristic Set
John worksFor RadioCom	E1 = {S1, S3}
Bob worksFor RadioCom	E1 = {S1, S3}
Jack worksFor RadioCom	E2 = {S2, S3}
RadioCom managedBy Mike	E3 = {S3, S4}
RadioCom registeredIn UK	E4 = {S3, S5}

Fig. 1: An RDF graph (left), its Characteristic Sets (top right), and Extended Characteristic Sets (bottom right). The evaluation of the query shown on the top of the figure is marked with bold nodes and edges on the graph.

John and *RadioCom*, as well as *Bob* and *RadioCom*; it comprises the properties (*name,origin,birthday,worksFor*) and (*address,label,managedBy, registeredIn*), respectively. In the same way, all ECSs of Fig. 1 are constructed and all triples in the RDF graph are partitioned based on the ECS they belong to. ECS are defined as an extension of *Characteristic Sets* [10] to represent *the structure of triples*, as opposed to nodes, in the data.

This partitioning requires less storage overhead, compared to the permutation approaches, by not relying on excessive replications, and decreases the effects of bad estimates by quickly accessing triples that collectively participate in multiple joins. 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 [9],
- 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 on top of an ECS index,
- we implement the approach in *axonDB*, a reference engine for ECS indexing and query processing that handles conjunctive multi-chain-star queries, and
- we evaluate its performance on one synthetic and two real datasets with respect to storage and querying, and we compare it with three widely used systems; our tool outperforms the competition by 1-3 orders of magnitude, both in the case of selective and unselective queries.

The rest of this paper is organized as follows. 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 for 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.

II. PRELIMINARIES

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 every node $n \in T = (I \cup B \cup L)$ and every edge $e \in I$ [13].

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. Neumann and Moerkotte [10] 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:

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

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

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

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 [10]. 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 (s, p, o), the ECS $E_c(s, o)$ is an ordered set containing the characteristic sets of s and o:

$$E_c(s,o) = \{S_c(s), S_c(o) \mid \exists p : (s, p, o) \in D\} \quad (3)$$

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

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

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 an edge from s towards o. In the example of Fig. 1, where nodes $John$ and $RadioCom$ are present in the same triple $\langle John, worksFor, RadioCom \rangle$ 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 Bob and $RadioCom$, $Jack$ and $RadioCom$, $RadioCom$ and $Mike$, as well as $RadioCom$ and $UK_Registry$. Note that, by definition, if two nodes n_1 and n_2 are linked 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 (s, p, o) corresponds to one and only one ECS, i.e., $E(s, o)$. 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 [10]. 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 II for several real-world and synthetic datasets.

ECS Graphs and ECS Query Graphs. ECSs can be combined to form a directed graph that captures transitive relationships between characteristic sets in an RDF dataset. This is useful for representing paths between types of s, o pairs. 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 two sets of triples whose schemas 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 on the right of Fig. 1, where we show the object-subject joins between ECSs of the RDF graph. Consider the query q listed at the top of the figure. 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. Its evaluation can be seen with bold edges in both the RDF and the ECS graph.

An ECS graph provides a suitable abstraction for traversing long 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 [16]. Instead, it treats subject-object joins as first-class citizens. With the help of ECS-based preprocessing and indexing, 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, we propose to extract the ECSs out of a query graph and map them to the dataset's ECS graph space. A query pattern q is mapped to the ECS query graph Q_E based on the identification and extraction of the ECSs of the triple patterns in q. Formally, a small modification to the ranges in the original definition of characteristic sets [10] 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., $S_c(s_q) = \{p_q \mid \exists o_q : (s_q, p_q, o_q) \in q\}$, $(s_q, p_q, o_q) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$.

We have implemented all aforementioned concepts in a native RDF engine, called *axonDB*. Its overall architecture is shown in Fig. 2. There are three core modules, responsible for a) loading a new RDF dataset and extracting the CS and ECS, b) constructing and storing the CS and ECS indexes and c) processing a SPARQL query and fetching the results. Next sections present the technical details for each module.

III. LOADING AND INDEXING

In this section, we provide methods and algorithms for efficient extraction of CS and ECS from datasets, and show how the ECS structure is used for triple storage and indexing.

A. Data Loading.

In axonDB, triples are stored on disk as three consecutive integers of 4 bytes, one for each triple component, namely subject, predicate and object, as is typically done in RDF stores [3], [11], [20]. The id assignment is performed during initial parsing of the input, and the references are stored in memory during the loading phase until they are flushed to disk in bulk. During the loading phase, each triple is modelled as a vector of size 4. The first three positions hold the subject, predicate, and object ids, and the last position points to 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. The usefulness of this verbosity will become clear later as we use this structure in order to sort the triples both by subject and CS. Furthermore, it is relatively

TABLE II: Observed cardinalities of properties, CS and ECS in synthetic and real data.

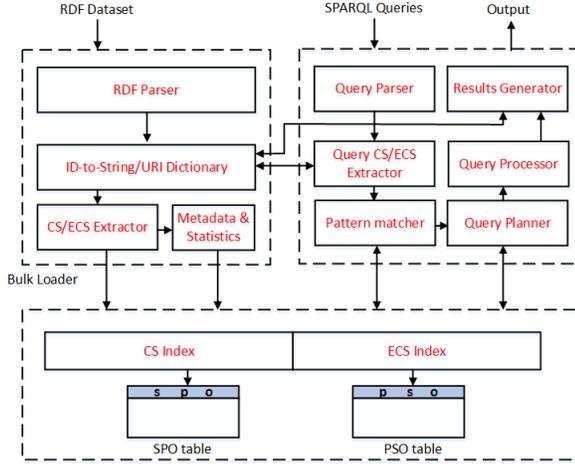|	LUBM	BSBM	WordNet	Reactome	EFO	GeoNames	DBLP
#properties	18	40	64	65	80	36	26
#CS	14	44	779	112	520	851	95
#ECS	68	374	7250	346	2515	12136	733

Fig. 2: Overview of system architecture.

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 overflows during data loading.

A dictionary is built during parsing, that holds values for the node and predicate ids (IRIs), as well as for the literals. IRIs are compressed based on their prefixes in order to avoid tedious duplications of strings that occur frequently in RDF datasets. 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 to generate the human readable results.

B. 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 are emitted 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 [5],[10]. The algorithm is presented in Algorithm 1. We sort the triples by subject and construct a new CS each time a new combination of properties is found in a subject, i.e., while we iterate through triples with the same subject, we aggregate the properties of these triples, and when the iteration moves on to the next subject, we hash the bitmap of the aggregated properties and check if it already exists. If not, we create a new CS with these properties. Each CS is assigned a unique integer identifier, and holds a bitmap of the properties that define it, where each bit corresponds to the presence of a property[5] in D (e.g., for k properties in D, we construct a bitmap of length k; in typical datasets - see *properties* row of Table II - k is small enough for the bitmap to fit in a few bytes). This is useful for fast subset checking during the query preprocessing phase, as will be discussed. During this iteration, we associate a triple to a CS, based on

[5]The properties are ordered as they appear in the first iteration of the input triples. We use this predicate ordering as a reference for all other stuctures and indexes that use it.

the CS of the *subject node*, by setting the fourth element of the triple's vector to the integer identifier assigned to the CS.

We then sort the triples by their CS, maintaining the subject as the secondary sort key, and iterate to construct a big triples table in the SPO ordering for persistent storage. Algorithm 1 does not show this step, as it returns a mapping from CSs to sets of triples. It is trivial to iterate through the keys of this mapping (*csMap*) and flush each CS's triples to the persistent storage sequentially. The *CS index* is constructed on top of this table as a B$^+$-tree, where the keys are defined by the id of the CSs. We can use this index to get the triples associated with a specific CS, by maintaining the start and end indexes of each CS in the SPO table. This way, the *CS Index* partitions all triples based on their subject's CS and allows us to easily evaluate properties in star patterns around a given node or variable, with simple range scans. For our running example, the SPO table and its CS index can be seen in Fig. 3. The CS id's refer to the characteristic sets that were shown in Fig. 1.

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

Analysis. Identification and extractions of CSs comprises sorting the triples once by subject, and scanning. This costs $O(nlogn + n)$ for n triples. Furthermore, we sort the triples a second time, and iterate over them once more in order to

store them on disk at the order of the CS's appearance, i.e., $O(n\log n + n)$. Not counting the cost of disk I/O, the time complexity of this step is $O(2n\log n + 2n)$.

C. Extended Characteristic Set Extraction and ECS Index.

The next step is to extract the ECSs, and build the *ECS index*. A naive way of extracting the ECSs is to perform an object-subject 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.

Specifically, we utilize the CS Index and iterate through all *pairs* of CSs looking for subject-object joins in their chunks of triples that are held in *csMap* (see Algorithm 1). When the result of the join between the triples of S_1 and S_2 is non-empty, we construct a new ECS $E(S_1, S_2)$, based on the CSs of the triples' subjects. This join process enables us to identify an ECS and retrieve all triples associated with that ECS at the same step. 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 object-subject 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 ECS $E(S_1, S_2)$ along with references to the identifiers of its subject and object CSs, as well as the triples contained in it. As with CSs, each ECS is assigned a unique integer identifier. In contrast to the *CS Index* that partitions all of the triples in a dataset, the *ECS Index* partitions only the triples that pertain to a valid ECS, i.e., whose subject and object have non-empty CSs. These are triples that describe paths between resources. We store these triples as a PSO table, and build the ECS Index as a B$^+$-tree on top of this table, where each ECS defines a range of consecutive triples that belong to it. This way, fetching the triples of a specific ECS requires a simple range scan over the PSO table. In our running example, the PSO table and ECS Index can be seen at the bottom of Fig. 3. Note that the size of the PSO table is smaller than the SPO table, which contains all triples of the input data. This is due to the fact that many triples do not belong to a valid ECS. These are either triples with literal objects, or triples with objects that do not have any emitting edges, and are thus described by an empty CS.

Algorithm 2 shows how ECSs are extracted and mapped to triples. The algorithm takes as input the *csMap* and results in two outputs: (i) a mapping of ECSs to sets of triples, and (ii) and a graph in the form of adjacency lists that represents the links between joinable ECSs. It iterates through all pairs S_i, S_j of CSs (lines 2-4), performs an object-subject hash-join of their triples T_i, T_j, (line 5) and if the result is not empty, it creates a new ECS and maps the triples to it, sorted in the PSO order (lines 6-8). This ordering is useful for early filtering of triples with properties not in the query pattern. After retrieving the ECSs, the algorithm finds directed links between ECSs (lines 11-15) to construct the *ECS graph*. It first populates the $subjectCSMap$ and $objectCSMap$ that link CSs to ECSs based on their position in the ECSs (lines 9-10). Then, it looks for CSs that are both objects and subjects in different sets of ECSs, and links these together. The resulting adjacency list represents the *ECS graph*, and is stored as part of the indexing scheme in axonDB. This is essential in the query preprocessing stage, in which an incoming query is matched to existing ECS paths in the data.

Analysis. For p CSs, the step of extracting ECSs contains

SPO table

ID	s	p	o
t1	Bob	name	"Bob Plain"
t2	Bob	origin	"Ireland"
t3	Bob	birthday	"1986"
t4	Bob	worksFor	RadioCom
t5	John	name	"John Doe"
t6	John	origin	"USA"
t7	John	birthday	"1976"
t8	John	worksFor	RadioCom
t9	Jack	name	"Jack Doe"
t10	Jack	origin	"UK"
t11	Jack	birthday	"1980"
t12	Jack	marriedTo	Alice
t13	Jack	worksFor	RadioCom
t14	RadioCom	label	"Radio Com"
t15	RadioCom	address	"21 Jump St."
t16	RadioCom	managedBy	Mike
t17	RadioCom	registeredIn	UK Registry
t18	Mike	position	"Director"
t19	UK Registry	label	"UK Company Registry"
t20	UK Registry	type	Registrar

CS Index

ID	properties	start	end
S_1	{1,1,1,1,0,0,...}	t_1	t_8
S_2	{1,1,1,1,1,0,...}	t_9	t_{13}
S_3	{0,0,0,0,0,1,...}	t_{13}	t_{17}
S_4	{0,0,0,0,0,0,...}	t_{18}	t_{18}
S_5	{0,0,0,0,0,0,...}	t_{19}	t_{20}

PSO table

ID	p	s	o
t4	worksFor	Bob	RadioCom
t8	worksFor	John	RadioCom
t13	worksFor	Jack	RadioCom
t16	managedBy	RadioCom	Mike
t17	registeredIn	RadioCom	UK Registry

ECS Index

ID	subjectCS	objectCS	start	end
E_1	S_1	S_3	t_4	t_8
E_2	S_2	S_3	t_{13}	t_{13}
E_3	S_3	S_4	t_{16}	t_{16}
E_4	S_3	S_5	t_{17}	t_{17}

Fig. 3: Example instantiation of the CS (top) and the ECS (bottom) indexes. The CS contains the bitmap of a set of properties $p_i..p_k$, while the ECS is a composition of a subject CS and an object CS.

	p_1 = name	p_2 = origin	p_3 = birthday	p_4 = worksFor	p_5 = marriedTo	p_6 = address	p_7 = label	p_8 = managedBy	p_9 = registeredIn	p_{10} = position	p_{11} = type
S_1	1	1	1	1	0	0	0	0	0	0	0
S_2	1	1	1	1	1	0	0	0	0	0	0
S_3	0	0	0	0	0	1	1	1	1	0	0
S_4	0	0	0	0	0	0	0	0	0	1	0
S_5	0	0	0	0	0	0	1	0	0	0	1

Fig. 4: Property bitmaps of CSs $S_1 \ldots S_5$

iterating p^2 pairs of CSs. For each pair S_i, S_j, the asymptotic cost of a hash join over the triples T_i, T_j of S_i, S_j respectively, is $O(|T_i| + |T_j|)$. Assuming an even distribution of $|D|/p$ triples per CS, where D is the input dataset, the total cost of ECS extraction is $O(p^2 2|D|/p)$, or $O(p2|D|)$. The computation of $ecsLinks$ entails building the $subjectCSMap$ and $objectCSMap$, which in the worst case when all p^2 pairs are valid, costs $O(p^2)$ insertions. Then, we have to find object CSs that are both in $subjectCSMap$ and $objectCSMap$, in order to ensure that an object CS of an ECS is also a subject CS of another ECS. This can be performed by iterating over $objectCSMap$, and for each CS iterating over pairs of ECSs that have the current CS as key in both $objectCSMap$ and $subjectCSMap$. Thus, the total cost of this step is $O(p^2 + p2|D|)$.

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, $ecsLinks$: A set of adjacency lists with links between the retrieved ECSs.

1: $ecsMap, subjectCSMap, objectCSMap \leftarrow$ new Map()
2: **for each** $S_i \in csMap$ **do**
3:   **for each** $S_j \in csMap$ **do**
4:    %Perform an object-subject join of triples in S_i, S_j%
5:    $triples \quad \leftarrow \quad (csMap.get(S_i) \quad \bowtie |_{o-s}csMap.get(S_j))$
6:    **if** $triples.size()! = 0$ **then**
7:     $ecs \leftarrow newECS(S_i, S_j)$
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: $ecsLinks \leftarrow newMap()$
13: **for each** $S_i \in objectCSMap.keys()$ **do**
14:   **if** $S_i \notin subjectCSMap.keys()$ **then**
15:    continue;
16:   **for each** $ecs_{left} \in objectCSMap.get(S_i)$ **do**
17:    **for each** $ecs_{right} \in subjectCSMap.get(S_i)$ **do**
18:     $ecsLinks.get(ecs_{left}).add(ecs_{right})$
  return $ecsMap, ecsLinks$

D. ECS Hierarchy.

ECSs pertain to a hierarchical structure that defines ancestral, parent-child relationships between them. We consider that an ECS E_1 is a specialization of another ECS E_2 if it contains all properties of E_2, i.e., $E_1 \succ E_2$. As will be discussed in the next section, a query is broken down to query ECSs, derived from the query's graph pattern. In order to match a query ECS with an ECS from our index, we perform subset checking between the query ECS and the ECSs in the index. As a derivative, a query ECS is evaluated by all ECSs that contain all properties of the query ECS, as they appear in the respective subject and object CSs. As many of these matched ECSs in the index are hierarchically related, because they contain *at least* the same subset of properties, we can use this ECS hierarchy to improve disk I/O and naturally group together triples that belong to the same ECS families. In our running example, Fig. 4 shows the bitmaps of the CSs, where each cell denotes the presence (1) or absence (0) of a property p_i. The two ECSs E_1 and E_2 of Fig.3 are hierarchically related, namely $E_1 \succ E_2$, because $S_1 \subset S_2$, and S_3 is the same for both. To take advantage of this observation, we implement an optimization that sorts ECSs based on the pre-order traversal of this hierarchy, and stores triples in the order defined by this traversal. This means that the ECSs of consecutive chunks of triples on the disk will often be hierarchically related, in an attempt to minimize reading redundant pages from persistent storage. For computing the hierarchy from $ecsLinks$, we sort the ECSs based on the number of the properties they contain, because the less properties it contains, the more generic and higher in the hierarchy the ECS is. Then, we iterate through the sorted ECSs, traverse their links from $ecsLinks$ and build paths for each one, taking care to add only one level of children to each previous level of ancestors. This process results in a graph lattice, where the root nodes are the most generic ECSs (containing fewer properties), and the leaves are the most specialized ECSs (containing more properties).

Metadata and statistics. During the loading phase, axonDB computes and stores along with the ECS index, auxiliary metadata and statistics in order to assist the pre-processing stage of query evaluation. First, each ECS maintains pointers to the first occurrences of each property in the indexed PSO table. This helps us avoiding logarithmic searches in large triple collections during query evaluation. Then, it extracts all 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 object-subject joins on the CS level. This is shown in lines 10-17 of Algorithm 2. Finally, it computes the cardinality of distinct properties in the triples of each ECS, as well as the cardinalities of distinct subjects and objects per ECS. These statistics are used by the query planner.

IV. QUERY PROCESSING

In this section, we discuss how query processing is performed on top of the ECS Index. Our goal is to employ the derived index structures in order to reduce the number of scans over the triples, number of joins, and the amount of intermediate results when evaluating SPARQL queries with multi-chain-star graph patterns. Still, our approach is efficient for simple query patterns as well. An overview of the query processing steps is shown in Fig. 5, from top to bottom. Given a query over our example dataset, we first parse the query statement and identify the characteristic sets around the chain variables, i.e., S_x, S_y, S_z, and S_w for $?x$, $?y$, $?z$, and $?w$ respectively. Then, we extract the query ECSs $Q_{x,y}, Q_{y,z}, Q_{y,w}$, and identify the chains between them as well as the type of joins to be performed; OS correspond to object-subject joins where the triples of an object's CS are joined with the triples of the subject's CS and SS denotes a subject-subject join. Finally, we match each query ECS to ECSs in the data and we generate the plan that retrieves and joins triples to output the result. Note that we consider the union of triples from E_1 and E_2 as $Q_{x,y}$ is matched to both ECSs. For simplicity reasons, we have omitted the step of processing the restriction of the bound *"Director"* node. This is performed when retrieving E_3, by doing a semi-join with the triples of its object CS and filtering out by the bound object.

A. 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, to create adjacency lists between the query ECSs. This procedure is identical to the ECS extraction when loading the data, but this time it is performed on the triple patterns of the query. During this step, the dictionary is used for id resolution of predicates and any other bound nodes in the patterns. Having identified the query ECSs and the links between them, we traverse these links in the order of their occurrence, in order to identify chains (i.e., series of object-subject joins) in the ECS query graph. This results in a set of *chain patterns* $c_1 \ldots c_n$. Finally, we remove chains that are fully contained in other chains. Given that their
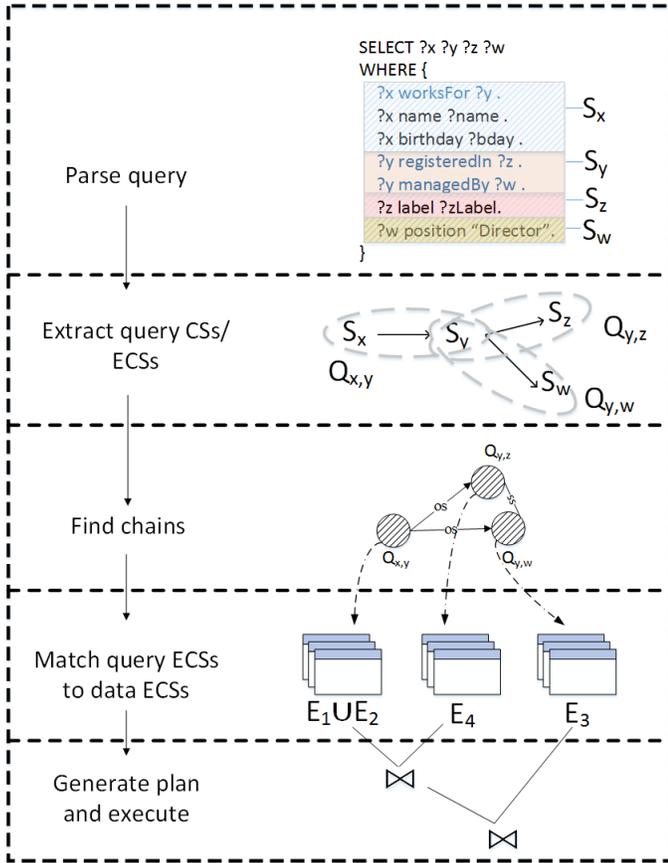
Fig. 5: Query processing for two chain patterns of three query ECSs. Notice that $Q_{x,y}$ matches both E_1 and E_1.

number is rather small, this step is efficiently performed with a single nested loop over the set of chains.

B. Matching of query ECSs to the ECS index.

Each query ECS Q_i is matched to zero or more ECSs in the ECS index. We say that there exists a match between a query ECS $Q_i = \{S_{q,left}, S_{q,right}\}$ and an indexed ECS $E_j = \{S_{j,left}, S_{j,right}\}$, when the following are true:

$$S_{q,left} \subseteq S_{j,left} \quad (5)$$
$$S_{q,right} \subseteq S_{j,right} \quad (6)$$
$$p(Q_i) \in p(E_j) \quad (7)$$

where $p(Q_i), p(E_j))$ are properties of the triple patterns whose subject and object CSs form ECS Q_i and E_j, respectively. In our running example, this is the set of properties that appear in the P column of the PSO table for the same ECS. If we denote the set of all ECSs that match Q_i as $matches(Q_i)$, then when (5)-(7) are true, it holds that $E_j \in matches(Q_i)$. Aggregating the triples of all ECSs in $matches(Q_i)$, gives the *evaluation eval* of Q_i, which is given by the following:

$$eval(Q_i) = \bigcup_{n=1}^{k} T(E_n), E_n \in matches(Q_i) \quad (8)$$

$T(E_n)$ corresponds to the triples associated with E_n.

Matching of the query ECSs to the ECS index is performed through a depth-first traversal on the ECS graph. As shown in Algorithm 3, we iterate over the edges of

$ecsLinks$ (line 2), an adjacency list with the ECS graph, and for each ECS as a starting point, we search for matching ECSs. The DFS traversal is performed by the recursive method $matchDataPatterns$ (lines 2-4) as detailed in Algorithm 4. The $matchDataPatterns$ method takes as input a query ECS, an indexed ECS (the starting/current node for the dfs traversal) and the two ECS graphs (i.e., the query and the indexed ECS graph) and returns a linked list of ECSs in the data that match the query ECSs. It first evaluates whether the input query ECS and the ECS in the data satisfy the conditions (5)-(7) (line 1-4), otherwise it returns an empty list. Subset checking is performed with bitwise operations on the property bitmaps. Specifically, a bitmap b_1 is a subset of a bitmap b_2 when it holds that $b_1 AND b_2 = b_1$. If all conditions are satisfied, it checks whether the examined ECS has already been visited or the query chain is empty (line 5) otherwise it marks the matching ECS as visited and adds it to the matching list of the input query ECS (Line 7-8). It then proceeds with the dfs traversal on the ECS graph (Line 9), and evaluates the matching ECSs for the rest of the chain starting from q_1, i.e., consecutive node in the query ECS chain. By performing depth-first traversal on the ECS graph, 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 output of this process is a set of ECS chains in the ECS graph that match the query's ECS chains. If the property of the ECS is unbound, then we match it to all properties found in the region of the PSO table that matches the rest of the ECS restrictions.

Algorithm 3 *matchQueryChainToECSIndex*

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

1:   $ecsMatches \leftarrow newMap()$
2:   **for each** $e \in ecsLinks.keySet()$ **do**
3:      matchDataPatterns($e, ecsLinks, c(q_0 \ldots q_{n-1})$,
4:      $ecsMatches$)
    return $ecsMatches$

In our running example of Fig. 1, the query of Fig. 5 defines three query ECSs, namely $Q_{x,y}$, $Q_{y,z}$ and $Q_{y,w}$, as can be seen in Fig. 5. The algorithm will match E_1, E_2 to $Q_{x,y}$ because the bitmap of S_x is a subset of both S_1 and S_2 that constitute the subject CSs of E_1 and E_2 respectively, and the bitmap of S_y is a subset of S_3 which is the common object CS for both E_1 and E_2. In a similar manner, E_4 will be matched to to $Q_{y,z}$ and E_3 to $Q_{y,w}$.

C. 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 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 ordering helps reduce intermediate results in object-subject joins between ECSs, that do not contribute to the final result.

To get the outer order of chains, each chain's cost is computed. The general rule is to order chains based on ascending

Algorithm 4 *matchDataPatterns*

Input: $ecsLinks$: The ECS adjacency list
Input: e: The ECS of the current iteration
Input: $c(q_0 \ldots q_{n-1})$: A chain of query ECSs
Input: $ecsMatches$: A linked list of ECS sets that match the ECSs in c
Output: $ecsMatches$: A linked list of ECS sets that match the ECSs in c

```
 1: if q₀.subjectCS.bitmap ⊈ e.subjectCS.bitmap
 2:     OR q₀.objectCS.bitmap ⊈ e.objectCS.bitmap
 3:     OR q₀.property ∉ e.properties  then
 4:         return null
 5: if visited(e) OR c.size == 1 then
 6:     return ecsMatches
 7: visited.add(e)
 8: ecsMatches.get(q₀).add(e)
 9: for each e_child ∈ ecsLinks.get(e) do
10:     matchDataPatterns(e_child, ecsLinks,
11:         c_{q₁...q_{n-1}}, ecsMatches)
```

---

cost, i.e., $cost(c_i) \leq cost(c_{i+1})$. The cost of evaluating a query ECS Q_i with unbound nodes (e.g. $?x$, $?y$ for $Q_{x,y}$ in Fig. 5) is the cost of reading all triples of $eval(Q_i)$, or its cardinality, that is, $cost(Q_i) = \sum_{n=1}^{k} |T(E_n)|$, with $E_n \in matches(Q_i)$. If either or both of Q_i are bound, we estimate the cost of its evaluation as a constant 1. For consecutive ECSs in a chain $c_k = Q_1 \bowtie Q_2 \bowtie \ldots \bowtie Q_k$, we estimate the cost of the resulting series of joins with the following recursive formula:

$$cost(c_{Q_1 \ldots Q_k}) = cost(c_{Q_1 \ldots Q_{k-1}}) \times m_{f,os}(Q_k) \quad (9)$$

where $m_{f,os}(Q_k)$ is the multiplication factor of Q_k for an object-subject join. The cost of a chain consisting of one ECS is given as the cardinality of the ECS, which is the base case of the recursion. The multiplication factor m_f of an ECS $E_i = \{S_{1,i}, S_{2,i}\}$, where $S_{1,i}$ and $S_{2,i}$ are the subject CS and object CS of E_i respectively, is an estimation of how many rows will be generated by performing an object-subject join with E_i at the right side. We define it as the ratio of (distinct) objects per subject in E_i, i.e., $m_{f,os}(E_i) = |o_{E_i}| / |s_{E_i}|$, where $|o_{E_i}|$ and $|s_{E_i}|$ are the distinct subject nodes of $S_{1,i}$ and $S_{2,i}$ respectively. We can use m_f instead of assuming independence between consecutive ECSs, because it is guaranteed from Algorithm 3 that the consecutive ECSs will be joined on the same sets of CSs, and not on the whole body of triples. We can afford adopting this type of estimation, because the cardinalities of the ECSs are generally bound to values much lower than the total size of the dataset.

For queries with bound nodes in the CSs of Q_i, we retrieve the triples of $eval(Q_i)$ by first retrieving the respective CSs, and scanning the regions of the SPO table that refer to the matched CSs. This, however, may affect the cardinalities of the ECSs; thus, the cost model adjusts the counts of distinct object and subject nodes to the numbers derived by the retrieved triples from the SPO table.

To get the inner ordering, we take into account the fact that all ECSs in a chain are linked with an object-subject join. This allows us to expand an existing node or sub-chain either left or right, one ECS at a time. Based on this, we employ a simple heuristic that starts from the ECS with the lowest cardinality, and expands the chain selecting the ECS with the minimum

cardinality from the left or right.

While other approaches use Dynamic Programming algorithms for finding the optimal join order based on the employed statistics in order to reduce intermediate results, in axonDB a large amount of the filtering of triples is already performed at the pattern matching stage. Thus, the order does not heavily affect the performance of the query processor, an observation that is reflected in our experiments, where we tested the system with the planner both disabled and enabled, and we found that while there is indeed a speed-up factor of 2-3 when the planner is enabled, the improvement is less than an order of magnitude for all experiments.

D. 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. 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 1 and 5, the restriction for the chain nodes to emit the bound properties is already enforced by the ECS definition.

For a query ECS Q_j, assuming that E_i is the most generic ECS that is matched to Q_j, this entails that *all supersets* of E_i, i.e., $E_i \succ E_{i'} \succ \cdots \succ E_{i''}$ will also belong to $matches(Q_j)$, thus the evaluation $eval(Q_j)$ must be the union of the triples in the hierarchically related ECSs that match the pattern, i.e., $T(E_i) \cup T(E_{i'}) \cup \cdots \cup T(E_{i''})$. Therefore, it is often expected to read the evaluations of ancestors or children of an ECS in the same evaluation process. This is the main reasoning behind our approach to store the triples of hierarchically related ECSs in close locality (see Section III), and essentially extend the range scan of a match to all of its matching neighbours as well.

V. EVALUATION

A. Experimental Setup

We have conducted an extensive experimental evaluation on *axonDB* with both synthetic and real-world data, and a comparative study with three high-performance RDF engines, namely *RDF-3x*, *Virtuoso opensource 7.2* and *TripleBit*. We have selected three native, high-performance, and centralized competitors that use different indexing approaches, in order to perform a system-wide comparison. For axonDB, we experiment with all four available optimization alternatives, i.e., a base configuration with the ECS hierarchy and query planner off (denoted with *axonDB*), two alternatives with one of them on (*axonDB-h* and *axonDB-qp*, respectively), and an optimized configuration with both features on (denoted with *axonDB+*), and assess the effect of these components to the performance of the system. 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

TABLE III: Size on disk (GB) and loading times (minutes)

	# triples	input	axonDB		RDF-3x		TripleBit		Virtuoso	
			size	time	size	time	size	time	size	time
LUBM2000	370m	54.2	8.12	68	16.54	58	10.88	45	14.6	45
Reactome	16m	2.8	0.71	3	1.07	2	0.74	2	0.91	2
Geonames	172m	18.8	8.24	81	12.48	34	8.6	20	8.56	27

the recommended tuning parameters given by Openlink, for RDF-3x and TripleBit we used the default deployment, which is non-tunable.

The aim of the experiments is to assess the performance of axonDB in *data loading*, *query execution* and *scalability* with synthetic data of increasing sizes. For the query runtime experiments, we execute the queries 20 times and report the best time. Furthermore, the experiments have been performed with cold caches, each time dropping the cache with the use of the sync; echo 1 > /proc/sys/vm/drop_caches command in linux. Our metrics are: *query execution time*, *loading time* and *disk storage size*. For query execution time, we set an upper timeout limit at 30 minutes.

Implementation. We have implemented axonDB as an open-source project[6], 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. axonDB 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, as well as all auxiliary indexes. All data structures except for the SPO and PSO tables, are stored as serialized Java objects using mapdBD. For triple serialization and persistence, axonDB uses byte arrays and random access files and writes all data in a single binary file, similar to RDF-3x and Virtuoso. The triples in the SPO/PSO tables are serialized as contiguous arrays of integers, and can be retrieved using range scans defined by the ECS/CS indexes. This format carries the benefit of being easily partitioned, while reducing disk reads to the number of matched ECSs per query. The ID-to-String/URI dictionary, which holds values for the compressed node and predicate ids, as well as the literals, is stored in the form of a clustered B^+-tree, with keys being sorted in ascending order. For this release, axonDB only supports conjunctive SPARQL queries with equi-joins.

Datasets and Queries. We employ one synthetic and two real-world datasets, widely used in the literature [10], [19], [20]. For synthetic data, we have used the *Lehigh University Benchmark* (LUBM) data generator to create RDF 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 axonDB does not support inferencing, we extended the LUBM generator to add all superclasses of an instance's class, in order to generate the transitive closure of the subclass relationships, as well as the *memberOf* and *hasAlumnus* properties. For our real-world experiments, we have chosen 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.

Regarding the queries, we create two sets of queries for LUBM, one set for Reactome, and one set for Geonames. LUBM defines 14 queries; most of them are simple, pertaining to a range of 1 to 6 triple patterns. From these, we select 6 representative queries, namely 2, 4, 7, 8, 9, and 12, which are the most challenging and have the largest numbers of triple patterns, in order to assess the performance of the system on traditional settings. To assess the performance on more complex queries, we create a second set of queries, by modifying 7 of the original queries (2, 3, 4, 8, 10, 11 and 12), converting all bound nodes to variables, and extending their characteristic sets, and we also define 5 additional ones. The queries are ordered by complexity[10], and Q1-8 are highly selective, while Q9-12 are low in selectivity. For the Reactome and Geonames datasets, we construct 8 and 6 queries, respectively, with increasing selectivity and numbers of chain patterns, i.e., 1-3 chains and 3-6 query ECSs. These take advantage of the long paths in the two datasets, and have progressively larger result sizes. In what follows, we present the results.

B. Experimental Results

Loading. The size in GB and loading time in minutes for the two real datasets and LUBM2000 can be seen in Table III. Overall, axonDB exhibits the lowest space overhead for the input data, along with TripleBit which comes second. This is a derivative of the low degree of data replication imposed by ECS indexing, and the fact that it only uses two triples tables (SPO and PSO). However, axonDB suffers from longer loading times compared to all three competitors, 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.

Comparison of different optimizations. Table IV compares the four versions of axonDB (based on the employed optimizations). We experiment with all queries from the modified LUBM and the two real-world datasets and we report the GM of all queries as well as the performance for four representative queries that exhibit the highest complexity in each dataset. The numbers denote the ratio of runtime of each configuration w.r.t. the runtime of the base implementation, which is shown at the first line. Overall, the relative performance improvement with all optimizations on is most cases better than its respective counterparts. The effects of the planner are minimized when the queries have only one chain, as the outer ordering and thus the cost model are redundant in such cases (e.g., LUBM Q5, Q12, Geonames Q6). The hierarchy optimization affects most queries, as multiple related ECSs often differ by a small number of properties in the data. Next, we only consider the worst and optimal configurations to compare them with the competitors.

---

[6]All code and queries are available in http://github.com/mmeimaris/axonDB

[7]www.mapdb.org

[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)

TABLE IV: Comparison of different optimization settings for representative queries.

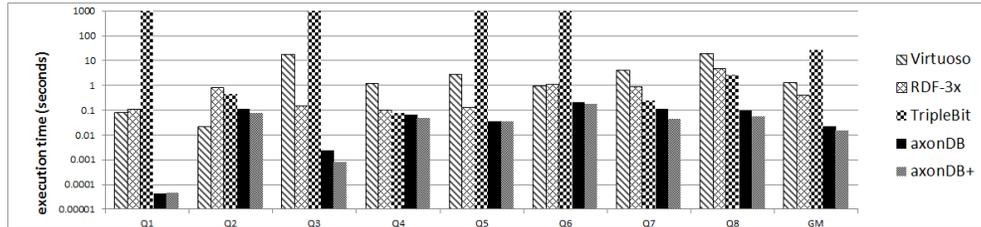|	LUBM					Reactome					Geonames				
	Q1	Q5	Q8	Q12	GM	Q2	Q3	Q7	Q8	GM	Q1	Q2	Q4	Q6	GM
axonDB	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
axonDB-h	0.97	**0.79**	0.83	0.82	0.79	0.76	0.56	0.73	0.99	0.82	0.73	0.81	0.75	**0.70**	0.74
axonDB-qp	0.77	1.01	0.86	0.98	0.83	0.85	0.57	0.53	**0.61**	0.73	0.71	0.56	0.65	1.05	0.72
axonDB+	**0.69**	0.87	**0.66**	**0.82**	**0.73**	**0.68**	**0.38**	**0.49**	**0.61**	**0.62**	**0.69**	**0.49**	**0.51**	**0.70**	**0.64**
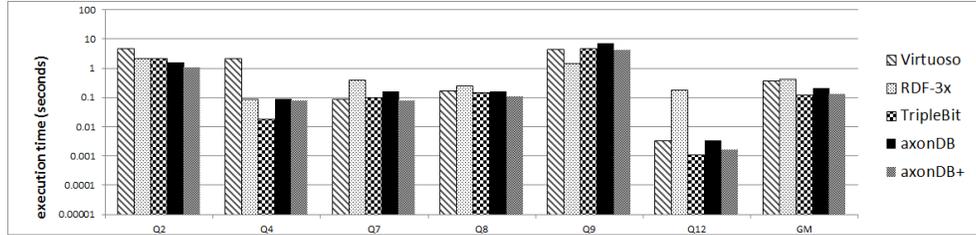
(a) Query runtimes for the LUBM dataset (original)
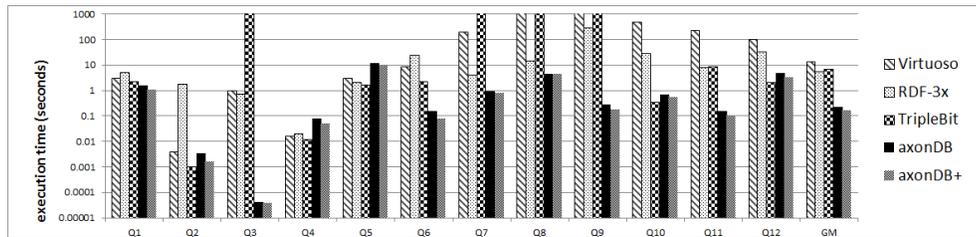
(b) Query runtimes for the LUBM dataset (modified)

(c) Query runtimes for the Reactome dataset
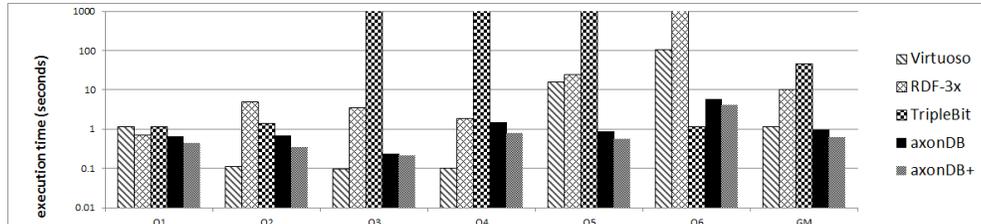
(d) Query runtimes for the Geonames dataset

Fig. 6: Query runtimes in seconds

Query performance - LUBM. All systems can address quite efficiently the original LUBM queries . This can be seen from the geometric means of the queries in Fig. 6(a), where we experiment with queries 2, 4, 7, 8, 9 and 12, which are the most challenging, and have more than one triple patterns. Overall, the performance of all systems lies in the same order of magnitude. Even though axonDB is designed to address more complex queries, this experiment shows that it can handle simple patterns efficiently as well.

The runtimes for *axonDB* and *axonDB+* against the three competitors for the modified LUBM queries can be seen in Fig. 6(b) along with their geometric mean (GM). The actual GM for Virtuoso and TripleBit 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, both axonDB configurations outperform the rest, with their geometric mean improving the competition by at least[11] 1 order of magnitude. Especially in the case of queries with complex patterns (Q7-12), axonDB is better by several orders of magnitude, while Virtuoso and TripleBit suffer several 30-minute timeouts. For Q3, which does not yield any results, the preprocessor cannot match the query graph to any ECS chains in the data, and thus

---

[11]In reality, it is more for the timeouts.

does not perform any joins, giving axonDB an advantage of up to 4 orders of magnitude compared to RDF-3x and Virtuoso (TripleBit timed out). In the more selective queries Q4 and Q5, axonDB is outmatched by the rest of the systems, because it does not have permuted indexes that quickly filter out triples that do not contribute to the solution, and has to suffer a full scan of the matched ECSs instead. However, in Q7, Q8 and Q9, both TripleBit and Virtuoso exhibited times over 30 minutes. These queries have long chains with up to 14 triple patterns with all nodes unbound except the predicates. Thus, the optimizers of these systems spend a lot of time dealing with large intermediate results created by the abundance of variables.

Reactome. The results against the three competitors are shown in Fig. 6(c). Again, both axonDB and axonDB+ outperform the rest for all 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), axonDB improves the competition by at least one order of magnitude, while TripleBit fails to answer four queries within 30 minutes. As in LUBM, these queries (Q1, Q3, Q5 and Q6) exhibit a large amount of unbound variables, in long chain patterns. 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 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. The results for Geonames are shown in Fig. 6(d). While axonDB and axonDB+ configurations outperform in all queries but Q4 and Q6, the improvements against RDF-3x and Virtuoso are not at the same scale with the previous datasets. Geonames 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, axonDB improves the competition by one order of magnitude overall, based on the observed GM. While TripleBit performs very fast on Q1, Q2 and Q6, it fails to answer three queries under the 30-minute timeframe, because its vertical partitioning storage scheme suffers from large intermediate joins in queries with long chains. This is an indication that such approaches that use inherent schema retrieval, are not suited for highly versatile RDF datasets.

Scalability. We have experimented with increasing input sizes of LUBM, starting from 15M triples, up to 370M triples. In Fig. 7, 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 axonDB+ 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, axonDB+ is outperformed by Virtuoso and RDF-3x as the input size increases. In any case, our experiments indicate that the methods presented herein are indeed scalable for larger input sizes.

VI. RELATED WORK

RDF data management systems follow three storage schemes, 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 [11] and Hexastore [17] 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 [6] uses a large 4-column table for quads, and a combination of full and partial indexes, while Jena TDB relies on three permutations. Other centralized RDF systems are built on top of relational backbones, such as Jena SDB, Virtuoso, and DB2RDF [4]. These methods have been established in centralized systems 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.

In distributed settings, a growing body of literature exists, with systems such as H2RDF+[12], S2RDF [14], SemStore [19], and TrinityRDF [21]. H2RDF+ employs all six permutations of the triples table, implemented over Hadoop and HBase. S2RDF uses a vertical partitioning schema named ExtVP, that takes into account the joins between vertical partitioning tables, while SemStore focuses on the partitioning aspects of data in different nodes, and uses TripleBit[20] in its reference implementation. TrinityRDF is designed to work in memory, and thus has no disk-based storage component. However, in this paper, our focus is on the limitations of the core aspects of centralized RDF systems to answer complex queries even in relatively small datasets, such as Reactome in our experiments. Thus, it is out of scope to perform a quantitative comparison with distributed RDF engines, and we leave it as future work to assess how ECS indexing can work on distributed settings.

Property Tables [18], [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 [7].

Vertical partitioning is a technique that partitions 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 [15]. TripleBit [20] 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. It does not consider the inherent schema of the triples in order to speed up the evaluation of complex query patterns, as is the case for axonDB.
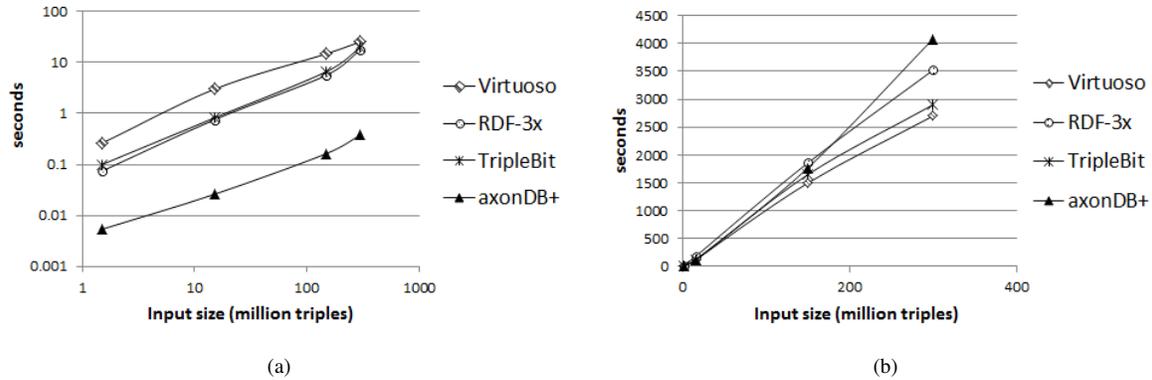
Fig. 7: Query execution (a) and dataset loading (b) for increasing sizes of LUBM

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 [10], and implemented in the RDF-3X high performance triple store. In this regard, Brodt et al [5] 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 axonDB with the use of the CS index, which is an SPO permutation partitioned among all CSs of a dataset. Our notion of the ECS is in fact inspired by the Characteristic Set, but focuses on triples, rather than nodes. In [8] we have presented a visionary layout of similar indexing, without providing an implementation or algorithmic contributions. To the best of our knowledge, this is the first work to use such a structure for RDF indexing and query processing.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented *axonDB*, a native RDF engine that employs the ECS indexing scheme, 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 with two optimizations, that take into account query planning and hierarchical relationships between ECSs. Finally, we performed an extensive experimental evaluation against three high-performance RDF storage and querying engines. The experimental evaluation has shown that axonDB outperforms the state of the art approaches, especially for answering complex query patterns with low selectivity. As future work, we will address data updates in existing ECS indexes, and study the application of the approach in a distributed setting.

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] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient rdf store over a relational database. In *SIGMOD*, pages 121–132. ACM, 2013.

[5] A. Brodt, O. Schiller, and B. Mitschang. Efficient resource attribute retrieval in rdf triple stores. In *CIKM*, pages 1445–1454. ACM, 2011.

[6] O. Erling and I. Mikhailov. *Virtuoso: RDF support in a native RDBMS*. Springer, 2010.

[7] 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.

[8] M. Meimaris. Evogen: a generator for synthetic versioned rdf. In T. Palpanas, E. Pitoura, W. Martens, S. Maabout, and K. Stefanidis, editors, *EDBT*, number 1558 in CEUR Workshop Proceedings, Aachen, 2016.

[9] 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.

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

[11] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19(1):91–113, 2010.

[12] N. Papailiou, D. Tsoumakos, I. Konstantinou, P. Karras, and N. Koziris. H 2 rdf+: an efficient data management system for big rdf graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 909–912. ACM, 2014.

[13] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In *ISWC*, volume 4273, pages 30–43. Springer, 2006.

[14] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen. S2rdf: Rdf querying with sparql on spark. *Proc. VLDB Endow.*, 9(10):804–815, June 2016.

[15] 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.

[16] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. Boncz. Heuristics-based query optimisation for sparql. In *EDBT*, pages 324–335. ACM, 2012.

[17] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *VLDB*, 1(1):1008–1019, 2008.

[18] K. Wilkinson and K. Wilkinson. Jena property table implementation, 2006.

[19] B. Wu, Y. Zhou, P. Yuan, H. Jin, and L. Liu. Semstore: A semantic-preserving distributed rdf triple store. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 509–518. ACM, 2014.

[20] 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.

[21] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *Proceedings of the VLDB Endowment*, volume 6, pages 265–276. VLDB Endowment, 2013.