Adaptive Low-level Storage of Very Large Knowledge Graphs

Jacopo Urbani  
jacopo@cs.vu.nl  
Vrije Universiteit Amsterdam  
Amsterdam, The Netherlands

Ceriel Jacobs  
ceriel@cs.vu.nl  
Vrije Universiteit Amsterdam  
Amsterdam, The Netherlands

ABSTRACT

The increasing availability and usage of Knowledge Graphs (KGs) on the Web calls for scalable and general-purpose solutions to store this type of data structures. We propose Trident, a novel storage architecture for very large KGs on centralized systems. Trident uses several interlinked data structures to provide fast access to nodes and edges, with the physical storage changing depending on the topology of the graph to reduce the memory footprint. In contrast to single architectures designed for single tasks, our approach offers an interface with few low-level and general-purpose primitives that can be used to implement tasks like SPARQL query answering, reasoning, or graph analytics. Our experiments show that Trident can handle graphs with $10^{11}$ edges using inexpensive hardware, delivering competitive performance on multiple workloads.

ACM Reference Format:

Jacopo Urbani and Ceriel Jacobs. 2020. Adaptive Low-level Storage of Very Large Knowledge Graphs. In Proceedings of The Web Conference 2020 (WWW '20), April 20–24, 2020, Taipei, Taiwan. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3366423.3380246

1 INTRODUCTION

Motivation. Currently, a large wealth of knowledge is published on the Web in the form of interlinked Knowledge Graphs (KGs). These KGs cover different fields (e.g., biomedicine [19, 65], encyclopedic or commonsense knowledge [86, 92], etc.), and are actively used to enhance tasks such as entity recognition [80], query answering [96], or more in general Web search [15, 30, 31].

As the size of KGs keeps growing and their usefulness expands to new scenarios, applications increasingly need to access large KGs for different purposes. For instance, a search engine might need to query a KG using SPARQL [39], enrich the results using embeddings of the graph [64], and then compute some centrality metrics for ranking the answers [46, 87]. In such cases, the storage engine must not only efficiently handle large KGs, but also allow the execution of multiple types of computation so that the same KG does not have to be loaded in multiple systems.

Problem. In this paper, we focus on providing an efficient, scalable, and general-purpose storage solution for large KGs on centralized architectures. A large amount of recent research has focused on distributed architectures [20, 27, 28, 33, 52, 74, 79], because they offer many cores and a large storage space. However, these benefits come at the price of higher communication cost and increased system complexity [71]. Moreover, sometimes distributed solutions cannot be used either due to financial or privacy-related constraints.

Centralized architectures, in contrast, do not have network costs, are commonly affordable, and provide enough resources to load all-but-the-largest graphs. Some centralized storage engines have demonstrated that they can handle large graphs, but they focus primarily on supporting one particular type of workload (e.g., Ringo [71] supports graph analytics, RDF engines like Virtuoso [67] or RDFox [60] focus on SPARQL [39]). To the best of our knowledge, we still lack a single storage solution that can handle very large KGs as well as support multiple workloads.

Our approach. In this paper, we fill this gap presenting Trident, a novel storage architecture that can store very large KGs on centralized architectures, support multiple workloads, such as SPARQL querying, reasoning, or graph analytics, and is resource-savvy. Therefore, it meets our goal of combining scalability and general-purpose computation.

We started the development of Trident by studying which are the most frequent access types performed during the execution of tasks like SPARQL answering, reasoning, etc. Some of these access types are node-centric (i.e., access subsets of the nodes), while others are edge-centric (i.e., access subsets of the edges). From this study, we distilled a small set of low-level primitives that can be used to implement more complex tasks. Then, the research focused on designing an architecture that supports the execution of these primitives as efficiently as possible, resulting in Trident.

At its core, Trident uses a dedicated data structure (a B+Tree or an in-memory array) to support fast access to the nodes, and a series of binary tables to store subsets of the edges. Since there can be many binary tables — possibly billions with the largest KGs — handling them with a relational DBMS can be problematic. To avoid this problem, we introduce a lightweight storage scheme where the tables are serialized on byte streams with only a little overhead per table. In this way, tables can be quickly loaded from the secondary storage without expensive pre-processing and offloaded in case the size of the database exceeds the amount of available RAM.

Another important benefit of our approach is that it allows us to exploit the topology of the graph to reduce its physical storage. To this end, we introduce a novel procedure that analyses each binary table and decides, at loading time, whether the table should be stored either in a row-by-row, column-by-column, or in a cluster-based fashion. In this way, the storage engines effectively adapt to the input. Finally, we introduce other dynamic procedures that decide, at loading time, whether some tables can be ignored due to their small sizes or whether the content of some tables can be aggregated to further reduce the space.

Since Trident offers low-level primitives, we built interfaces to several engines (RDF3X [63], VLog [89], SNAP [50]) to evaluate
the performance of SPARQL query answering, datalog reasoning and graph analytics on various types of graphs. Our comparison against the state-of-the-art shows that our approach can be highly competitive in multiple scenarios.

**Contribution.** We identified the followings as the main contributions of this paper.

- We propose a new architecture to store very large KGs on a centralized system. In contrast to other engines that store the KG in few data structures (e.g., relational tables), our architecture exhaustively decomposes the storage in many binary tables such that it supports both node- and edge-centric via a small number of primitives;
- Our storage solution adapts to the KG as it uses different layouts to store the binary tables depending on its topology. Moreover, some binary tables are either skipped or aggregated to save further space. The adaptation of the physical storage is, as far as we know, a unique feature which is particularly useful for highly heterogeneous graphs, such as the KGs on the Web;
- We present an evaluation with multiple workloads and the results indicate highly competitive performance while maintaining good scalability. In some of our largest experiments, Trident was able to load and process KGs with up to $10^{11}$ (100B) edges with hardware that costs less than $5K$.

The source code of Trident is freely available with an open source license at https://github.com/karmaresearch/trident, along with links to the datasets and instructions to replicate our experiments.

2 PRELIMINARIES

A graph $G = (V, E, L, \phi_V, \phi_E)$ is a tuple where $V, E, L$ represent the sets of nodes, edges and labels respectively, $\phi_V$ is a bijection that maps each node to a label in $L$, while $\phi_E$ is a function that maps each edge to a label in $L$. We assume that there is at most one edge with the same label between any pair of nodes. Throughout, we use the notation $r(s, d)$ to indicate the edge with label $r$ from node with label $s$ (source) to the node with label $d$ (destination).

We say that the graph is **undirected** if $r(s, d) \in E$ implies that also $r(d, s) \in E$. Otherwise, the graph is **directed**. A graph is **unlabeled** if all edges map to the same label. In this paper, we will mostly focus on labeled directed graphs since undirected or unlabeled graphs are special cases of labeled directed graphs.

In practice, it is inefficient to store the graph using the raw labels as identifiers. The most common strategy, which is the one we also follow, consists of assigning a numerical ID to each label in $L$, and stores each edge $r(s, d)$ with the tuple $(t_e, t_r, t_d)$ where $t_e$, $t_r$, and $t_d$ are the IDs associated to $s$, $r$, and $d$ respectively.

The numerical IDs allow us to sort the edges and by permuting $t_e, t_r, t_d$ we can define six possible ordering criteria. We use strings of three characters over the alphabet {$s, r, d$} to identify these orderings, e.g., srd specifies that the edges are ordered by source, relation, and destination. We denote with $\mathcal{R} = \{srd, sdr, drs, ...\}$ the collection of six orderings while $\mathcal{R}^+ = \{s, r, d, sr, rs, sd, ds, dr, rd\}$ specifies all partial orderings. We use the function isprefix to check whether string $a$ is a prefix of $b$, i.e., isprefix($a, b$) = $[true/false]$ and the operator $\sim$ to remove all characters of one string from another one (e.g., if $a = srd$ and $b = sd$, then $a \sim b = r$).

Let $V$ be a set of variables. A **simple graph pattern** (or **triple pattern**) is an instance of $L \cup V \times L \cup V \times L \cup V$ and we denote it as $(X, Y, Z)$ where $X, Y, Z \in L \cup V$. A **graph pattern** is a finite set of simple graph patterns. Let $\sigma : V \rightarrow L$ be a partial function from variables to labels. With a slight abuse of notation, we also use $\sigma$ as a postfix operator that replaces each occurrence of the variables in $\sigma$ with the corresponding node. Given the graph $G$ and a simple graph pattern $\psi$, the answers for $\psi$ on $G$ correspond to the set $\text{ans}(G, \psi) = \{r(s, d) \mid r(s, d) \in E \land q(a, b, c, d, \psi)\}$. Function $\text{bound}(\psi)$ returns the positions of the labels in the simple graph pattern $\psi$ left-to-right, i.e., if $\psi = (X, a, b)$ where $X \in V$ and $a, b \in L$, then $\text{bound}(\psi) = rd$.

A **Knowledge Graph** (KG) is a directed labeled graph where nodes are entities and edges establish semantic relations between them, e.g., (Sadiq Khan, majorOf, London). Usually, KGs are published on the Web using the RDF data model [45]. In this model, data is represented as a set of RDF triples of the form $(subject, predicate, object)$ drawn from $(I \cup \mathcal{B}) \times (I \cup \mathcal{L})$ where $I, \mathcal{B}, \mathcal{L}$ denote sets of IRIs, blank nodes and literals respectively. Let $T = I \cup \mathcal{B} \cup \mathcal{L}$ be the set of all RDF terms. RDF triples can be trivially seen as a graph where the subjects and objects are the nodes, triples map to edges labeled with their predicate name, and $L = T$.

SPARQL [39] is a language for querying knowledge graphs which has been standardized by W3C. It offers many SQL-like operators like UNION, FILTER, DISTINCT to specify complex queries and to further process the answers. Every query contains at its core a graph pattern, which is called Basic Graph Pattern (BGP) in the SPARQL terminology. SPARQL graph patterns are defined over $T \cup V$ and their answers are mappings $\sigma$ from $V$ to $T$. Therefore, answering a SPARQL graph pattern $P$ over a KG $G$ corresponds to computing $\text{ans}(G, p_1) \cap \ldots \cap \text{ans}(G, p_{|P|})$ and retrieving the corresponding labels.

**Example 1.** An example of a SPARQL query is:

```
SELECT ?s ?o { ?s isa ?o . ?s livesIn Rome . }
```

If the KG contains the RDF triples (Eli, isa, Professor) and (Eli, livesIn, Rome) then one answer to the query is $\{(?s \rightarrow \text{Eli}, ?o \rightarrow \text{Professor})\}$.

3 GRAPH PRIMITIVES

We start our discussion with a description of the low-level primitives that we wish to support. We distilled these primitives considering four types of workloads: SPARQL [39] query answering, which is the most popular language for querying KGs; Rule-based reasoning [7], which is an important task in the Semantic Web to infer new knowledge from KGs; Algorithms for graph analytics, or network analysis, since these are widely applied on KGs either to study characteristics like the graph’s topology or degree distribution, or within more complex pipelines; Statistical relational models [64], which are effective techniques to make predictions using the KG as prior evidence.

If we take a closer look at the computation performed in these tasks, we can make a first broad distinction between edge-centric and node-centric operations. The first ones can be defined as operations that retrieve subsets of edges that satisfy some constraints. In contrast, operations of the second type retrieve various data about
the nodes, like their degree. Some tasks, like SPARQL query answering, depend more heavily on edge-centric operations while others depend more on node-centric operations (e.g., random walks).

**Graph Primitives.** Following a RISC-like approach, we identified a small number of low-level primitives that can act as basic building blocks for implementing both node- and edge-centric operations. These primitives are reported in Table 1 and are described below.

$f_1$ to $f_4$. These primitives retrieve the numerical IDs associated with labels and vice-versa. The primitives $f_1$ and $f_2$ retrieve the labels associated with nodes and edges respectively. The primitives $f_3$ and $f_4$ retrieve the labels associated with numerical IDs.

$f_5$ to $f_{10}$. Function $\text{edg}_{\text{grp}}(G, p)$ retrieves the subset of the edges in $G$ that matches the simple graph pattern $p$ and returns it sorted according to $\omega$. Primitives in this group are particularly important for the execution of SPARQL queries since they encode the core operation of retrieving the answers of a SPARQL triple pattern.

$f_{11}$ to $f_{16}$. This group of primitives returns an aggregated version of the output of $f_5$ to $f_{10}$. For instance, $\text{grp}_p(G, p)$ returns the list $\langle(x_1, c_1), \ldots, (x_n, c_n)\rangle$ of all distinct sources in the edges $\text{ans}(G, p)$ with the respective counts of the edges that share them. Let $D$ be a set of edges, $\text{A}(x, D) = \{r(x, d) \in D\}$ and $\text{B}(D) = \{\{s, e\} | r(s, d) \in \text{A}(s, D) \land c = |\text{A}(s, D)|\}$. Then, $\text{grp}_p(G, p)$ returns the list of all tuples in $\text{B}(\text{ans}(G, p))$ sorted by the numerical ID of the first field. The other primitives are defined analogously.

$f_{17}$. This primitive returns the cardinality of the output of $f_5$ to $f_{16}$. For instance, $\text{grp}_p(G, p)$ returns the list $\langle(x_1, c_1), \ldots, (x_n, c_n)\rangle$ of all distinct sources in the edges $\text{ans}(G, p)$ with the respective counts of the edges that share them. Let $D$ be a set of edges, $\text{A}(x, D) = \{r(x, d) \in D\}$ and $\text{B}(D) = \{\{s, e\} | r(s, d) \in \text{A}(s, D) \land c = |\text{A}(s, D)|\}$. Then, $\text{grp}_p(G, p)$ returns the list of all tuples in $\text{B}(\text{ans}(G, p))$ sorted by the numerical ID of the first field. The other primitives are defined analogously.

$f_{18}$ to $f_{33}$. These primitives return the $i^{th}$ edge that would be returned by the corresponding primitives $\text{edg}_{\text{grp}}$. In practice, this operation is needed in several graph analytics algorithms or for mini-batching during the training of statistical relational models.

**Table 1: Graph primitives**

| Name | Output |
|------|--------|
| $f_1$ | $\text{lbl}_{\text{grp}}(G, e)$ |
| $f_2$ | $\text{lbl}_{\text{grp}}(G, e)$ |
| $f_3$ | $\text{node}_{\text{grp}}(G, l)$ |
| $f_4$ | $\text{edge}_{\text{grp}}(G, l)$ |
| $f_5$ | $\text{edg}_{\text{grp}}(G, p)$ |
| $f_6$ | $\text{edg}_{\text{grp}}(G, p)$ |
| $f_7$ | $\text{edg}_{\text{grp}}(G, p)$ |
| $f_8$ | $\text{edg}_{\text{grp}}(G, p)$ |
| $f_9$ | $\text{edg}_{\text{grp}}(G, p)$ |
| $f_{10}$ | $\text{edg}_{\text{grp}}(G, p)$ |
| $f_{11}$ | $\{s \in \text{ans}(s, p) \land \forall r \in \text{r}_{\text{ans}}(s, p) \land \forall d \in \text{d}_{\text{ans}}(s, p)\}$ |
| $f_{12}$ | $\{s \in \text{ans}(s, p) \land \forall r \in \text{r}_{\text{ans}}(s, p) \land \forall d \in \text{d}_{\text{ans}}(p, s)\}$ |
| $f_{13}$ | $\{s \in \text{ans}(s, p) \land \forall r \in \text{r}_{\text{ans}}(s, p) \land \forall d \in \text{d}_{\text{ans}}(p, s)\}$ |
| $f_{14}$ | $\{s \in \text{ans}(s, p) \land \forall r \in \text{r}_{\text{ans}}(s, p) \land \forall d \in \text{d}_{\text{ans}}(p, s)\}$ |
| $f_{15}$ | $\{s \in \text{ans}(s, p) \land \forall r \in \text{r}_{\text{ans}}(s, p) \land \forall d \in \text{d}_{\text{ans}}(p, s)\}$ |
| $f_{16}$ | $\{s \in \text{ans}(s, p) \land \forall r \in \text{r}_{\text{ans}}(s, p) \land \forall d \in \text{d}_{\text{ans}}(p, s)\}$ |
| $f_{17}$ | $\text{count}(f_1, \ldots, f_{10})$ |

**Figure 1: Architectural overview of Trident**

**Example 2.** We show how we can use the primitives in Table 1 to answer the SPARQL query of Example 1, assuming that the KG is called $I$.

- First, we retrieve the IDs of the labels isA, livesIn, and Rome. To this end, we can use the primitives $f_3$ and $f_4$.
- Then, we create two single graph patterns $p_1$ and $p_2$ which map to the first and second triple patterns respectively. Then, we execute $\text{edg}_{\text{grp}}(I, p_1)$ and $\text{edg}_{\text{grp}}(I, p_2)$ so that the edges are returned in an order suitable for a merge join.
- We invoke the primitive $f_1$ to retrieve all the labels of the nodes which are returned by the join algorithm. These labels are then used to construct the answers of the query.

4 ARCHITECTURE

One straightforward way to implement the primitives in Table 1 is to store the KG in many independent data structures that provide optimal access for each function. However, such solution will require a large amount of space and updates will be slow. It is challenging to design a storage engine that uses fewer data structures without excessively compromising the performance.

Moreover, KGs are highly heterogeneous objects where some subgraphs have a completely different topology than others. The storage engine should take advantage of this diversity and potentially store different parts of the KGs in different ways, effectively adapting to its structure. This adaptation lacks in current engines, which treat the KG as a single object to store.

Our architecture addresses these two problems with a compact storage layer that supports the execution of primitives $f_1, \ldots, f_{23}$ with a minimal compromise in terms of performance, and in such a way that the engine can adapt to the KG in input selecting the best strategy to store its parts.

Figure 1 gives a graphical view of our approach. It uses a series of interlinked data structures that can be grouped in three components. The first one contains data structures for the mappings $ID \Rightarrow label$. The second component is called edge-centric storage and contains data structures for providing fast access to the edges. The third one is called node-centric storage and offers fast access to the nodes. Section 4.1 describes these components in more detail. Section 4.2 discusses how they allow an efficient execution of the primitives, while Section 4.3 focuses on loading and updating the database.
4.1 Architectural Components

**Dictionary.** We store the labels on a block-based byte stream on disk. We use one B+Tree called $\text{DICT}$, to index the mappings $ID \Rightarrow label$ and another one called $\text{DICT}_1$ for $label \Rightarrow ID$. Using B+Trees here is usual, so we will not discuss it further. It is important to note that assigning a certain ID to a term rather than another one might have a significant impact on the performance. For instance, Urbani et al. [88] have shown that a careful choice of the IDs can introduce speedups due to the improved data locality. Typically, current graph engines assign unique IDs to all labels, irrespectively whether a label is used as an entity or as a relation. This is desirable for SPARQL query answering because all data joins can operate on the IDs directly. There are cases, however, where unique ID assignments are not optimal. For instance, most implementations of techniques for creating KG embeddings (e.g., TranSE [16]) store the embeddings for the entities and the ones for relations in two contiguous vectors, and use offsets in the vectors as IDs. If the labels for the relations share the IDs with the entities, then the two vectors must have the same number of elements. This is highly inefficient because KGs have many fewer relations than entities which means that much space in the second vector will be unused. To avoid this problem, we can assign IDs to entities and relationships in an independent manner. In this way, no space is wasted in storing the embeddings. Note that Tridart supports both global ID assignments and independent entity/relationship assignments with an additional index specifically for the relation labels. The first type of assignment is needed for tasks like SPARQL query answering while the second is useful for operations like learning graph embeddings [64].

**Edge-centric storage.** In order to adapt to the complex and non-uniform topology of current KGs, we do not store all edges in a single data structure, but store subsets of the edges independently. These subsets correspond the edges which share a specific entity/relationship set. More specifically, let us assume that we must store the graph $G = (V, E, L, \phi_r, \phi_e)$. For each $l \in L$, we consider three types of subsets: $E_s(l) = \{r(l, d) \in E\}$, $E_r(l) = \{l(s, d) \in E\}$, $E_d(l) = \{l(s, l) \in E\}$, i.e., the subsets of edges that have $l$ as source, edge, or destination respectively.

The choice of separating the storage of various subsets allows us to choose the best data structure for a specific subset, but it hinders the execution of *inter-table scans*, i.e., scans where the content of multiple tables must be taken into account. To alleviate this problem, we organize the physical storage in such a way that all edges can still be retrieved by scanning a contiguous memory location.

We proceed as follows: First, we compute $E_s(l)$, $E_r(l)$, $E_d(l)$ for every $l \in L$. Let $\Omega$ be the collection of all these sets. For each $E_s(l) \in \Omega$, we construct two sets of tuples, $F_s(l)$ and $G_s(l)$, by extracting the free fields left-to-right and right-to-left respectively. For instance, the set $E_s(l)$ results into the sets $F_s(l) = \{(r, d) | r(l, d) \in E\}$ and $G_s(l) = \{(d, r) | r(l, d) \in E\}$. Since these sets contain pairs of elements, we view them as binary tables. These are grouped into the following six sets:

- $T_s = \{F_s(l) | l \in L\}$ and $T'_s = \{G_s(l) | l \in L\}$
- $T_r = \{E_r(l) | l \in L\}$ and $T'_r = \{G_r(l) | l \in L\}$
- $T_d = \{E_d(l) | l \in L\}$ and $T'_d = \{G_d(l) | l \in L\}$

The content of these six sets is serialized on disk in corresponding byte streams called $T_s$, $T'_s$, $T_r$, $T'_r$, $T_d$, and $T'_d$ respectively (see middle section of Figure 1). The serialization is done by first sorting the binary tables by their defining label IDs, and then serializing each table one-by-one. For instance, if $F_s(l_1), F_s(l_2) \in T_s$, then $F_s(l_1)$ is serialized before $F_s(l_2)$ if $l_1 < l_2$. At the beginning of the byte stream, we store the list of all IDs associated to the tables, pointers to the tables’ physical location and instructions to parse them.

Since the binary tables and tuples are serialized on the byte stream with a specific order, we can retrieve all edges sorted with any ordering in $R$ with a single scan of the corresponding byte stream, using the content stored at the beginning of the stream to decode the binary tables in it. For instance, we can scan $T_s$ to retrieve all edges sorted according to $sr$. The IDs stored at the beginning of the stream specify the sources of the edges ($s$) while the content of the tables specify the remaining relations and destinations ($r$ and $d$).

**Node-centric storage.** In order to provide fast access to the nodes, we map each ID $l$ (i.e., the ID assigned to label $l$) to a tuple $M_l$ that contains 15 fields:

- the cardinalities $|E_s(l)|$, $|E_r(l)|$, and $|E_d(l)|$;
- six pointers $p_1, \ldots, p_6$ to the physical storage of $F_s(l)$, $G_s(l)$, $F_r(l)$, $G_r(l)$, $F_d(l)$, and $G_d(l)$;
- six bytes $m_1, \ldots, m_6$ that contain instructions to read the data structures pointed by $p_1, \ldots, p_6$. These instructions are necessary because the tables are stored in different ways (see Section 5).

We index all $M_l$ tuples by the numerical IDs using one global data structure called NM (Node Manager), shown on the left side of Figure 1. This data structure is implemented either with an on-disk B+Tree or with a in-memory sorted vector (the choice is done at loading time). The B+Tree is preferable if the engine is used for edge-based computation because the B+Tree does not need to load all nodes in main memory and the nodes are accessed infrequently anyway. In contrast, the sorted vector provides much faster access ($O(1)$ vs. $O(\log|L|)$) but it requires that the entire vector is stored in main memory. Thus, it is suitable only if the application accesses the nodes very frequently and there are enough hardware resources.

Note that the contents of the binary tables are stored both in NM and in the meta-data in front of the byte stream. This means that the table can be accessed either by accessing NM, or by scanning the beginning of the byte stream. In our implementation, we consult NM when we need to answer graphs patterns with at least one constant element (e.g., for answering the query in Example 1). In contrast, the meta-content at the beginning of the stream is used when must perform a full scan.

The way we store the binary tables in six byte streams resembles six-permutation indexing schemes such as proposed in engines like RDF3X [63] or Hexastore [93]. There are, however, two important differences: First, in our approach the edges are stored in multiple independent binary tables rather than a single series of ternary tuples (as, for instance, in RDF3X [63]). This division is important because it allows us to choose different serialization strategies for subgraphs or to avoid storing some tables (Section 5.3). The second difference is that in our case most access patterns go through a single B+Tree instead of six different data structures. This allows us to
We now discuss how we can implement the primitives in Table 1 with our architecture.

**4.2 Primitive Execution**

We discuss the execution of the primitives from Table 1 with our architecture. Consider a call to $\text{grp}_{\omega}(G, p)$ where $p = \langle a, X, Y \rangle$. In this case, we can query NM with a and return at most one tuple with the cardinality stored in $M_a$, which has a cost of $O(\log(|L|))$.

If $\omega$ has length two or $p$ contains a repeated variable, then we also need to access one or more binary tables, similarly as before.

**Proposition 5.** Let $G = (V, E, L, \phi_V, \phi_E)$. The time complexity of executing $\text{pos}_{\omega}(G, p, i)$ ranges between $O(\log(|L|))$ and $O(|L| + |E|)$.

**4.3 Bulk Loading and Updates**

**Bulk Loading.** Loading a large KG can be a lengthy process, especially if the resources are constrained. In Trident, we developed a loading routine which exploits the multi-core architecture and maximizes the (limited) I/O bandwidth.
The main operations are shown in Figure 2. Our implementation can receive the input KG in multiple formats. Currently, we considered the N-Triples format (popular in the Semantic Web) and the SNAP format [49] (used for generic graphs). The first operation is encoding the graph, i.e., assigning unique IDs to the entities and relation labels. For this task, we adapted the MapReduce technique presented at [90] to work in a multi-core environment. This technique first deconstructs the triples, then assigns unique IDs to all the terms, and finally reconstructs the triples. If the graph is already encoded, then our procedure skips the encoding and proceeds to the second operation of the loading, the creation of the database.

The creation of the binary tables requires that the triples are pre-sorted according to a given ordering. We use a disk-based parallel merge sort algorithm for this purpose. The tables are serialized one-by-one selecting the most efficient layout for each of them. After all the tables are created, the loading procedure will create the NM and the B+Trees with the dictionaries. The encoding and sorting procedures are parallelized using threads, which might need to communicate with the secondary storage. Modern architectures can have >64 cores, but such a number of threads can easily saturate the disk bandwidth and cause serious slowdowns. To avoid this problem, we have two types of threads: Processing threads, which perform computation like sorting, and I/O threads, which only read and write from disk. In this way, we can control the maximum number of concurrent accesses to the disks.

**Updates.** To avoid a complete re-loading of the entire KG after each change, our implementation supports incremental updates. Our procedure is built following the well-known advice by Jim Gray [29] that discourages in-place updates, and it is inspired by the idea of differential indexing [63], which proposes to create additional indices and perform a lazy merging with the main database when the number of indices becomes too high.

Our procedure first encodes the update, which can be either an addition or removal, and then stores it in a smaller “delta” database with its own NM and byte streams. Multiple updates will be stored in multiple databases, which are timestamped to remember the order of updates. Also, updates create a new update database if they introduce new terms. Whenever the primitives are executed, the content of the updates is combined with the main KG so that the execution returns an updated view of the graph.

In contrast to differential indexing, our merging does not copy the updates in the main database, but only groups them in two updates, one for the additions and one for the removals. This is to avoid the process of rebuilding binary tables with possibly different layouts. If the size of the merged updates becomes too large, then we proceed with a full reload of the entire database.

### 5 ADAPTIVE STORAGE LAYOUT

The binary tables can be serialized in different ways. For instance, we can store them row-by-row or column-by-column. Using a single serialization strategy for the entire KG is inefficient because the tables can be very different from each other, so one strategy may be efficient with one table but inefficient with another. Our approach addresses this inefficiency by choosing the best serialization strategy for each table depending on its size and content.

For example, consider two tables $T_1$ and $T_2$. Table $T_1$ contains all the edges with label “isA”, while $T_2$ contains all the edges with label “isbnValue”. These two tables are not only different in terms of sizes, but also in the number of duplicated values. In fact, the second column of $T_1$ is likely to contain many more duplicate values than the second column of $T_2$ because there are (typically) many more instances than classes while “isbnValue” is a functional property, which means that every entity in the first column is associated with a unique ISBN code. In this case, it makes sense to serialize $T_1$ in a column-by-column fashion so that we can apply run-length-encoding (RLE) [1], a well-known compression scheme of repeated values, to save space when storing the second column. This type of compression would be ineffective with $T_2$ since there each value appears only once. Therefore, $T_2$ can be stored row-by-row.

In our approach, we consider three different serialization strategies, which we call serialization layouts (or simply layouts) and employ an ad-hoc procedure to select, for each binary table, the best layout among these three.

#### 5.1 Serialization Layouts

We refer to the three layouts that we consider as row, column, and cluster layouts respectively. The first layout stores the content row-by-row, the second column-by-column, while the third uses an intermediate representation.

**Row layout.** Let $T = \langle t_1, t_2, \ldots, t_n \rangle$ be a binary table that contains $n$ sorted pairs of elements. With this layout, the pairs are stored one after the other. In terms of space consumption, this layout is optimal if the two columns do not contain any duplicated value. Moreover, if each row takes a fixed number of bytes, then it is possible to perform binary search or perform a random access...
to a subset of rows. The disadvantage is that with this layout all values are explicitly written on the stream while the other layouts allow us to compress duplicate values.

**Column layout.** With this layout, the elements in $T$ are serialized as $(t'_{1}, \ldots, t'_{n})$. The space consumption required by this layout is equal to the previous one but with the difference that here we can use RLE to reduce the space of $(t'_{1}, \ldots, t'_{n})$. In fact, if $t'_{i} = t'_{i} = \ldots = t'_{m}$, then we can simply write $t'_{i} \times m$. Also this layout allows binary search and random access to the table. However, it is slightly less efficient than the row layout for full scans because here one row is not stored at contiguous locations, and the system needs to "jump" between columns in order to retrieve the entire pair. On the other hand, this layout is more suitable than the row layout for aggregate reads (required, for instance, for executing grp primitives) because in this case we only need to read the content of one column which is stored at contiguous locations.

**Cluster layout.** Let $gt = \langle (t_{1}, t'_{1}), \ldots, (t_{n}, t'_{n}) \rangle$ be the longest subsequence of pairs in $T$ which share the first term $t$. With this layout, all groups are first ordered in the sequence $(g_{1}, \ldots, g_{t_{1}})$ such that $g_{i} < t_{i+1}$ for all $1 \leq i < m$. Then, they are serialized one-by-one. Each group $g_{i}$ is serialized by first writing $t_{i}$, then $g_{i}$, and finally the list $t'_{1}, \ldots, t'_{n}$. This layout needs less space than the row layout if the groups contain multiple elements. Otherwise, it uses more space because it also stores the size of the groups, and this takes an extra $\lceil \log n \rceil$ bits. Another disadvantage is that with this layout binary search is only possible within one group.

### 5.2 Dynamic Layout Selection

The procedure for selecting the best layout for each table is reported in Algorithm 1. Its goal is to select the layout which leads to the best compression without excessively compromising the performance. In our implementation, Algorithm 1 is applied by default, but the user can disable it and use one layout for all tables.

The procedure receives as input a binary table $T$ with $n$ rows and returns a tuple that specifies the layout that should be chosen. It proceeds as follows. First, it makes a distinction between tables that have less than $\tau$ rows (default value of $\tau$ is 1M) and contain less than $v$ unique elements in the first column from tables that do not (line 2). We make this distinction because 1) if the number of rows is too high then searching for the most optimal layout becomes expensive and 2) if the number of unique pairs is too high, then the cluster layout should not be used due to the lack of support for binary search. With small tables, this is not a problem because it is well known that in these cases linear search is faster than binary search due to a better usage of the CPU’s cache memory. The value for $v$ is automatically determined with a small routine that performs some micro-benchmarks to identify the threshold after which binary search becomes faster. In our experiments, this value ranged between 16 and 64 elements.

If the table satisfies the condition of line 2, then the algorithm selects either the ROW or the CLUSTER layout. The COLUMN layout is not considered because its main benefit against the other two is a better compression (e.g., RLE) but this is anyway limited if the table is small. The procedure scans the table and keeps track of the largest numbers and groups used in the table ($m_{1}, m_{2}, m_{3}$). Then, the function invokes the subroutine sizeoff() to retrieve the number of bytes needed to store these numbers. It uses this information to compute the total number of bytes that would be needed to store the table with the ROW and CLUSTER layout respectively (variables $t_{r}$ and $t_{c}$). Then, it selects the layout that leads to maximum compression.

If the condition in line 2 fails, then either the ROW or the COLUMN layout can be selected. An exact computation would be too expensive given the size of the table. Therefore, we always choose COLUMN since the other one cannot be compressed with RLE.

Next to choosing the best layout, Algorithm 1 also returns the maximum number of bytes needed to store the values in the two fields in the table ($m_{1}$ and $m_{2}$) and (optionally) also for storing the cluster size ($m_{3}$, this last value is only needed for CLUSTER). The reason for doing so is that it would be wasteful to use four- or eight-byte integers to store small IDs. In the worst case, we assume that all IDs in both fields can be stored with five bytes, which means it can store up to $2^{40} - 1$ terms. We decided to use byte-wise compression rather than bit-wise compression because the latter does not appear to be worthwhile [63]. Note that more complex compression schemes could also be used (e.g., VByte [94]) but this should be seen as future work.

The tuple returned by selectlayout contains the information necessary to properly read the content of the table from the byte stream. The first field is the chosen layout while the other fields are the number of bytes that should be used to store the entries of the table. We store this tuple both in NM (in one of the $m_{1}$ fields) and at the beginning of the byte stream.

### 5.3 Table Pruning

With Algorithm 1, the system adapts to the KG while storing a single table. We discuss two other forms of compression that consider multiple tables and decide whether some tables should be skipped or stored in aggregated form.

**On-the-fly reconstruction (OFR).** Every table in one stream $T_{x}$ maps to another table in $T'_{x}$ where the first column is swapped with the second column. If the tables are sufficiently small, one of them can be re-constructed on-the-fly from the other whenever needed. While this operation introduces some computational overhead, the saving in terms of space may justify it. Furthermore, the overhead
can be limited to the first access by serializing the table on disk after the first re-construction.

We refer to this strategy as on-the-fly reconstruction (OFR). If the user selects it at loading time, the system will not store any binary table in $T'_r$ which has less than $n$ rows, $n$ being a value passed by the user (default value is 20, determined after microbenchmarking).

**Aggregate Indexing.** Finally, we can construct aggregate indices to further reduce the storage space. The usage of aggregate indices is not novel for KG storage [93]. Here, we limit their usage to the tables in $T'_r$ if they lead to a storage space reduction.

To illustrate the main idea, consider a generic table $t$ that contains the set of tuples $F_t$($isa$). This table stores all the (object, subject) pairs of the triples with the predicate $isa$. Since there are typically many more instances than classes, the first column of $t$ (the classes) will contain many duplicate values. If we range-partition $t$ with the first field, then we can identify a copy of the values in the second field of $t$ in the partitions of tables in $T'_d$ where the first term is $isa$. With this technique, we avoid storing the same sequence of values twice but instead store a pointer to the partition in the other table.

### 6 EVALUATION

Trident is developed in C++, is freely available, and works under Windows, Linux, MacOS. Trident is also released in the form of a Docker image. The user can interact via command line, web interface, or HTTP requests according to the SPARQL standard.

**Integration with other systems.** Since our system offers low-level primitives, we integrated it with the following other engines with simple wrappers to evaluate our engine in multiple scenarios:

- **RDF3X** [63]. RDF3X is one of the fastest and most well-known SPARQL engines. We replaced its storage layer with ours so that we can reuse its SPARQL operators and query optimizations.
- **SNAP** [50]. Stanford Network Analysis Platform (SNAP) is a high-performance open-source library to execute over 100 different graph algorithms. As with RDF3X, we removed the SNAP storage layer and added an interface to our own engine.
- **VLog** [89]. VLog is one of most scalable datalog reasoners. We implemented an interface allowing VLog to reason using our system as underlying database.

We also implemented a native procedure to answer basic graph patterns (BGPs) that applies greedy query optimization based on cardinalities, and uses either merge joins or index loop joins if the first cannot be used.

**Testbed.** We used a Linux machine (kernel 3.10, GCC 6.4, page size 4k) with dual Intel ES-2630v3 eight-core CPUs of 2.4 GHz, 64 GB of memory and two 4TB SATA hard disks in RAID-0 mode. The commercial value is well below $5k. We compared against RDF3X and SNAP with their native storages, TripleBit [97], a in-memory state-of-the-art RDF database (in contrast to RDF3X which uses disks), and SYSTEM_A, a widely used commercial SPARQL engine¹. As inputs, we considered a selection of real-world and artificial KGs, and other non-KG graphs from SNAP [49] (see Table 2 for statistics).

- **KGs.** LUBM [32], a well-known artificial benchmark that creates KGs of arbitrary sizes. The KG is in the domain of universities and each university contributes ca. 100k new triples. Henceforth, we write LUBMx to indicate a KG with $x$ universities, e.g. LUBM10 contains 1M triples; DBpedia [14], YAGO2S [84] and Wikidata [92], three widely used KGs with encyclopedic knowledge; Uniprot [75], a KG that contains biomedical knowledge, and BTC2012 [41], a collection of crawled interlinked KGs.

- **Other graphs.** We considered the graphs: Google1, a Web graph from Google, Twitter, which contains a social circle, and Astro, a collaboration network in Physics.

Trident was configured to use the B+Tree for NM and table pruning was disabled, unless otherwise stated.

#### 6.1 Lookups

During the loading procedure, Trident applies Algorithm 1 to determine the best layout for each table. Figure 3a shows the number of tables of each type for the KGs. The vast majority of tables is stored either with the ROW or CLUSTER layout. Only a few tables are stored with the COLUMN layout. These are mostly the ones in the TR and TR² byte streams. It is interesting to note that the number of tables varies differently among different KGs. For instance, the number of row tables is twice the number of cluster tables with LUBM. In contrast, with Wikidata there are more cluster tables than row ones. These differences show to what extent Trident adapted its physical storage to the structure of the KG.

One key operation of our system is to retrieve answers of simple triple patterns. First, we generated all possible triple patterns that return non-empty answers from YAGO2S. We considered five types of patterns. Patterns of type 0 are full scans; of type 2 contain one constants and two variables (e.g., $X$, $type$, $Y$), while of type 4 contain two constants and one variable (e.g., $X$, $type$, person). These patterns are answered with edg., Patterns of types 1 request an aggregated version of a full scan (e.g., retrieve all subjects) while patterns of type 3 request an aggregation where the pattern contains

1¹We hide the real name as it is a commercial product, as usual in database research.
We can construct a similar query if we consider the variables in the which would return non-empty answers.

Table 3 reports such details also for queries of type 3 and 4. By
with the count of triples that share each subject. With this input,

Since these two techniques trade speed for space. Their benefit is
OFR the table, we see that if the two pruning strategies are enabled,

Adaptive Low-level Storage of Very Large Knowledge Graphs
WWW ’20, April 20–24, 2020, Taipei, Taiwan

one constant (e.g., return all objects of the predicate type). These

The number, types of patterns, and average number of answers per type is reported in Table 3. The first column reports the type of pattern and the orderings we can apply when we retrieve it. The second column reports an example pattern of this type. The third column contains the number of different queries that we can construct of this type. The fourth column reports the average number of answers that we get if we execute queries of this type.

For example, the first row describes the pattern of type 0, which is a full scan. For this type of pattern, we can retrieve the answers with all the orderings in \( R \). There is only one possible query of this type (column 3) and if we execute it then we obtain about 76M answers (column 4). Patterns of type 1 correspond to full aggregated scans. An example pattern of this type is shown in the second row. If this query is executed, the system will return the list of all subjects with the count of triples that share each subject. With this input, this query will return about 8M results (i.e., the number of subjects). We can construct a similar query if we consider the variables in the second or third position. Details for these two cases are reported in the third and fourth rows.

Patterns of type 2 have one constant and two variables. Like before the constant can appear in three positions. Note that in this case we can construct many more queries by using different constants. For instance, we can construct 8.6M queries if the constant appears as subject, and 99 if it appears as predicate. Similarly, Table 3 reports such details also for queries of type 3 and 4. By testing our system on all these types of patterns, we are effectively evaluating the performance over all possible queries of these types which would return non-empty answers.

We used the primitives to retrieve the answers for these patterns with various configurations of our system, and compared against RDF3X, which was the system with fastest runtimes. The median warm runtimes of all executions are reported in Figure 3b.

The row “Default” reports the results with the adaptive storage selected by Algorithm 1 but without table pruning. The rows “With OFR” and “With AGGR” use Algorithm 1 and the two techniques for table pruning discussed in Section 5.3 respectively. The rows “Only ROW (COLUMN)” use only the ROW and COLUMN layouts (the CLUSTER is not competitive alone due to the lack of binary search). From the table, we see that if the two pruning strategies are enabled, then the runtimes increase, especially with OFR. This was expected since these two techniques trade speed for space. Their benefit is
that they reduce the size of the database, as shown in Figure 3c. In particular, OFR is very effective, and they can reduce the size by 35%. Therefore, they should only be used if space is critical. The ROW layout returns competitive performance if used alone but then the database size is about 9% larger due to the suboptimal compression. Figure 3c also reports the size of the databases with the other systems as reference. Note that the reported numbers for Trident do not include the size of the dictionary (764MB). This size should be added to the reported numbers for a fair comparison with the other systems’ databases.

A comparison against RDF3X shows that the latter is faster with full scans (patterns of type 0) because our approach has to visit more tables stored with different configurations. However, our approach has comparable performance with the second pattern type and performs significantly better when the scan is limited to a single table, with, in the best case, improvements of more than two orders of magnitude (pattern 3). Note that in contexts like SPARQL query answering, patterns that contain at least one constant are much more frequent than full scans (e.g., see Table 2 of [76]).

6.2 SPARQL

Table 4 reports the average of five cold and warm runtimes with our system and with other state-of-the-art engines. For LUBM, DBPedia, Uniprot, and BTC2012, we considered queries used to evaluate previous systems [97]. For Wikidata, we designed five example queries of various complexity looking at published examples. The queries are reported in Appendix A. Unfortunately, we could not load Wikidata and BTC2012 with SYSTEM_A due to raised exceptions during the loading phase.

We can make a few observations from the obtained results. First, a direct comparison against TripleBit is problematic because sometimes TripleBit crashed or returned wrong results (checked after manual inspection). Looking at the other systems, we observe that our approach returned the best cold runtimes for 20 out of 25 queries, counting in both the executions with our native SPARQL engine and the integration with RDF3X. If we compare the warm runtimes, our system is faster 20 out of 25 times. Furthermore, we observe that Trident/N is faster than Trident/R mostly with selective queries that require only a few joins. Otherwise the second is faster. The reason is that RDF3X uses a sophisticated query optimizer that builds multiple plans in a bottom-up fashion. This procedure is costly if applied to simple queries, but it pays off for more complex ones because it can detect a better execution plan.
| Q. | N. Query Answers | Cold runtime (ms.) | Warm runtime (ms.) |
|----|-----------------|-------------------|-------------------|
|    |                 | TN | TR | EN | Expiry | SYSTEL_A | TN | TR | EN | Expiry | SYSTEL_A |
| 1  | 10              | 136,165 | 50,839 | 186,127 | 61.78 | 956.34 | 0.107 | 0.164 | 1.331 | 11.76 | 0.39 |
| 2  | 10              | 307,625 | 3,112 | 174,060 | 0.0 | 10,620 | 0.181 | 1.067 | 3.528 | N.A. | 0.33 |
| 3  | 0               | 807,870 | 6,468 | 712,365 | 0.31 | 24,538 | 0.806 | 194.1 | 2,387 | 791,738 | 0.04 | 9,365 |
| 4  | 2,528           | 643,072 | 24,848 | 738,473 | 20,308 | 49,816 | 611,083 | 19,435 | 795,541 | 20,298 | 9,244 |
| 5  | 353,939         | 9,801 | 25,694 | 105,366 | 0.0 | 171,859 | 4,794 | 14,852 | 22,448 | 3,308 | 100,007 |

Table 4: Average runtimes of SPARQL queries. Column “TN” reports the runtime of our approach with the native SPARQL implementation while “TR” is the runtime with the RDF3X SPARQL engine. LUM8k is a generated database with about 1B RDF triples. Red background means that TripleBit returned wrong answers (’x’ means it crashed); “N.A” means that the experiment was not possible due to failure at loading time.

| Task | ASTRO | GOOGLE | TWITTER |
|------|-------|--------|---------|
| Hits | 372 | 195 | 288 |
| BFS  | 81993 | 62241 | 1604097 | 1709823 | 215704 | 243740 |
| Triangles | 69 | 31 | 1533 | 526 | 607 | 105 |
| Random Walks | 25 | 34 | 30 | 41 | 26 | 32 |
| MaxWCC | 22 | 15 | 594 | 351 | 132 | 65 |
| MaxSSC | 47 | 29 | 1177 | 712 | 228 | 148 |
| Diameter | 1176 | 5233 | 168211 | 243669 | 56312 | 42581 |
| PageRank | 515 | 319 | 14616 | 7771 | 4482 | 1738 |
| ClustCoef | 375 | 417 | 8519 | 7114 | 5886 | 4178 |
| Ours | 6.6s | OpenKE [35] | 18.72s |

Table 5: Runtime of various graph analytics algorithms

6.3 Graph Analytics, Reasoning, Learning

Graph analytics. Algorithms for graph analytics are used for path analysis (e.g., find the shortest paths), community analysis (e.g., triangle counting), or to compute centrality metrics (e.g., PageRank). They use frequently the primitives pos and count to traverse the graph or to obtain the nodes’ degree. For these experiments, we used the sorted list as NODEMGR since these algorithms are node-centric.

We selected ten well-known algorithms: HITS and PageRank compute centrality metrics; Breadth First Search (BFS) performs a search; MOD computes the modularity of the network, which is used for community detection; Triangle Counting counts all triangles; Random Walks extracts random paths; MaxWCC and MaxSSC compute the largest weak and strong connected components respectively; Diameter computes the diameter of the graph while ClustCoef computes the clustering coefficient.

We executed these algorithms using the original SNAP library and in combination with our engine. Note that the implementation of the algorithms is the same; only the storage changes. Table 5 reports the runtimes. From it, we see that our engine is faster in most cases. It is only with random walks that our approach is slower. From these results, we conclude that also with this type of computation our approach leads to competitive runtimes.

Reasoning and Learning. We also tested the performance of our system for rule-based reasoning. In this task, rules are used to materialize all possible derivations from the KG. First, we computed reasoning considering Trident and VLog, using LUBM and two popular rulesets (LUBM-L and LUBM-LE) [60, 89]. Then, we repeated the process with the native storage of VLog. The runtime, reported in Table 6, shows that our engine leads to an improvement of the performance (48% faster in the best case).

Finally, we considered statistical relational learning as another class of problems that could benefit from our engine. These techniques associate each entity and relation label in the KG to a numerical vector (called embedding) and then learn optimal values for the embeddings so that truth values of some unseen triples can be computed via algebraic operations on the vectors.

We implemented TransE [16], one of the most popular techniques of this kind, on top of Trident and compared the runtime of training vs. the one produced by OpenKE [35], a state-of-the-art library. Table 6 reports the runtime to train a model using as input a subset of LUBM-LE.

Table 6: Runtime of reasoning and learning
7 RELATED WORK

In this section, we describe the most relevant works to our problem. For a broader introduction to graph and RDF processing, we redirect to existing surveys [3, 24, 57, 59, 68, 77, 95]. Current approaches can be classified either as native (i.e., designed for this task) or non-native (adapt pre-existing technology). Native engines have better performance [17], but less functionalities [17, 23]. Our approach belongs to the first category.

Research on native systems has focused on advanced indexing structures. The most popular approach is to extensively materialize a dedicated index for each permutation. This was initially proposed by YARS [43], and further explored in RDF3X [10, 12, 13, 26, 63]. Also Hexastore [93] proposes a six-way permutation-based indexing, but implemented it using hierarchical in-memory Java hash maps. Instead, we use on-disk data structures and therefore can scale to larger inputs. Recently, other types of indices, based on 2D or 3D bit matrices [8, 97], hash-maps [60], or data structures scale to larger inputs. Recently, other types of indices, based on 2D or 3D bit matrices [8, 97], hash-maps [60], or data structures used for graph matching approaches [47, 99] have been proposed. If compared with these works, our approach uses a novel layout of data structures and uses multiple layouts to store the subgraphs.

Non-native approaches offload the indexing to external engines (mostly DBMS). Here, the challenge is to find efficient partitioning/replication criteria to exploit the multi-table nature of relational engines. Existing partitioning criteria group the triples either by existing surveys [3, 24, 57, 59, 68, 77, 95]. Current approaches can be classified either as native (i.e., designed for this task) or non-native (adapt pre-existing technology). Native engines have better performance [17], but less functionalities [17, 23]. Our approach belongs to the first category.

Research on native systems has focused on advanced indexing structures. The most popular approach is to extensively materialize a dedicated index for each permutation. This was initially proposed by YARS [43], and further explored in RDF3X [10, 12, 13, 26, 63]. Also Hexastore [93] proposes a six-way permutation-based indexing, but implemented it using hierarchical in-memory Java hash maps. Instead, we use on-disk data structures and therefore can scale to larger inputs. Recently, other types of indices, based on 2D or 3D bit matrices [8, 97], hash-maps [60], or data structures used for graph matching approaches [47, 99] have been proposed. If compared with these works, our approach uses a novel layout of data structures and uses multiple layouts to store the subgraphs.

Non-native approaches offload the indexing to external engines (mostly DBMS). Here, the challenge is to find efficient partitioning/replication criteria to exploit the multi-table nature of relational engines. Existing partitioning criteria group the triples either by predicates [2, 38, 51, 56, 62, 72, 73], clusters of predicates [21], or by using other entity-based splitting criteria [17]. The various partitioning schemes are designed to create few tables to meet the constraints of relational engines [81]. Our approach differs because
we group the edges at a much higher granularity generating a number of binary tables that is too large for such engines.

Some popular commercial systems for graph processing are Virtuoso [67], BlazeGraph [85], Titan [22], Neo4J [61] Sparksee [83], and InfiniteGraph [66]. We compared Trident against such a leading commercial system and observed that ours has very competitive performance; other comparisons are presented in [5, 81]. In general, a direct comparison is challenging because these systems provide end-to-end solutions tailored for specific tasks while we offer general-purpose low-level APIs.

Finally, many works have focused on distributed graph processing [4, 6, 9, 33, 36, 38, 48, 70, 78, 98]. We do not view these approaches as competitors since they operate on different hardware architectures. Instead, we view ours as a potential complement that can be employed by them to speed up distributed processing.

In our approach, we use numerical IDs to store the terms. This form of compression has been the subject of some studies. First, some systems use the Hash-code of the strings as IDs [37, 38, 40]. Most systems, however, use counters to assign new IDs [18, 42, 43, 53, 63]. It has been shown in [88] that assigning some IDs rather than others can improve the query answering due to data locality. It is straightforward to include these procedures in our system. Finally, some approaches focused on compressing RDF collections [54] and on the management of the strings [11, 55, 82]. We adopted a conventional approach to store such strings. Replacing our dictionary with these proposals is an interesting direction for future work.

8 CONCLUSION
We proposed a novel centralized architecture for the low-level storage of very large KGs which provides both node- and edge-centric access to the KG. One of the main novelties of our approach is that it exhaustively decomposes the storage of the KGs in many binary tables, serializing them in multiple byte streams to facilitate inter-table scanning, akin to permutation-based approaches. Another main novelty is that the storage effectively adapts to the KG by choosing a different layout for each table depending on the graph topology. Our empirical evaluation in multiple scenarios shows that our approach offers competitive performance and that it can load very large graphs without expensive hardware.

Future work is necessary to apply or adapt our architecture for additional scenarios. In particular, we believe that our system can be used to support Triple Pattern Fragments [91], an emerging paradigm to query RDF datasets, and GraphQL [44], a more complex graph query language. Finally, it is also interesting to study whether the integration of additional compression techniques, like locality-based dictionary encoding [88] or HDT [25], can further improve the runtime and/or reduce the storage space.

Acknowledgments. We would like to thank (in alphabetical order) Peter Boncz, Martin Kersten, Stefan Manegold, and Gerhard Weikum for discussing and providing comments to improve this work. This project was partly funded by the NWO research programme 400.17.605 (VWData) and NWO VENI project 639.021.335.

A SPARQL QUERIES

A.1 LUBM

PREFIX rdf: <http://www.w3.org/1999/02/}
SELECT ?a ?b ?ab WHERE {
  ?p u:organism <http://purl.uniprot.org/taxonomy/9606>.
  ?p r:type u:Protein.
  SELECT ?p ?a WHERE {
    Q4:
    ?a r:type u:Transmembrane_Annotation .
  }  
  ?p u:annotation ?a.
  SELECT ?p ?a WHERE {
    Q3:
    ?a u:annotation ?vo.
    SELECT ?a ?vo WHERE {
      Q2:
      ?a rs:seeAlso <http://purl.uniprot.org/prints/PR00992>.
      ?a rs:seeAlso <http://purl.uniprot.org/pfam/PF00842>.
      ?a rs:seeAlso <http://purl.uniprot.org/eggnog/COG0787>.
      ?a u:encodedBy ?vo.
      SELECT ?a ?vo WHERE {
        Q1:
        PREFIX u: <http://purl.uniprot.org/core/>.
        PREFIX r: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
        A.4 Uniprot
        ?b ?bn ?n.
        ?a rdf:type dbpedia:Protein.
        SELECT ?a ?y WHERE {
          Q5:
          ?c2 pos:long ?long.
        }
        ?a dbpedia:placeOfBirth ?c.
        ?b dbpedia:placeOfBirth ?c.
        ?b rdf:type <http://dbpedia.org/ontology/Person>.
        ?a rdf:type <http://dbpedia.org/ontology/Person>.
        ?a dbpedia:spouse ?b.
        SELECT ?a ?b ?lat ?long WHERE {
          Q4:
          ?b geo:inCountry <http://www.geonames.org/countries/US>.
          ?a owl:sameAs ?b.
          ?b r:type u:Protein.
          SELECT ?b r:type u:Protein.
          SELECT ?b rdf:type <http://dbpedia.org/class/yago/Politician110451263>.
         }
        ?a dbpedia:years 7?.
        ?a <http://xmlns.com/foaf/0.1/name> ?n.
        ?b ?bn 7n.
        ?b rdf:type <http://dbpedia.org/ontology/OfficeHolder>.
      }  
    }
  }
}

A.4 Uniprot

PREFIX r: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rs: <http://www.w3.org/2000/01/rdf-schema#>  
PREFIX u: <http://purl.uniprot.org/core/>.
PREFIX r: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

A.5 Wikidata

PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdtp: <http://www.wikidata.org/prop/direct/>

REFERENCES
[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-Oriented Database Systems. In Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (Chicago, IL, USA) (SIGMOD ‘06). Association for Computing Machinery, New York, NY, USA, 671â€“682. https://doi.org/10.1145/1142473.1142548
[2] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Holenbach. 2009. SW-Store: a vertically partitioned DBMS for Semantic Web data management. The VLDB Journal 18, 2 (2009), 385–406.
[3] Ibrahim Abdelaziz, R. Harbi, Zuhair Khayyat, and Panos Kalnis. 2017. A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. Proceedings of the VLDB Endowment 10, 13 (Sept. 2017), 2049â€“2066. https://doi.org/10.14778/3151106.3151109
[4] I. Abdelaziz, R. Harbi, S. Sahighi, and P. Kalnis. 2017. Combining vertex-centric graph processing with SPARQL for large-scale RDF data analytics. IEEE Transactions on Parallel and Distributed Systems 28, 12 (Dec. 2017), 3374–3388.
[5] Gunes Alpc, M. Tamer Ozsu, Khuzaaima Daudjee, and Olaf Hartig. 2015. Executing queries over schemable RDF databases. In 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13–17, 2015. IEEE Computer Society, Seoul, South Korea, 807–818. https://doi.org/10.1109/ICDE.2015.7113335
[6] Bernd Aumann, Olivier Cuisl, and Hubert Naacke. 2018. Distributed SPARQL Query Processing: A Case Study with Apache Spark. John Wiley & Sons, Ltd, Chapter 2, 21–55. https://doi.org/10.1002/9781119152827.ch2
[7] Gregoris Antoniou, Sotiris Batsakis, Raghaba Mathurap, Jeff Z. Pan, Guolin Qi, Ilia Tachmazidis, Jacopo Urbani, and Zhangquan Zhou. 2018. A survey of large-scale reasoning on the Web of data. The Knowledge Engineering Review 33 (2018), 1–43.
[8] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. 2010. Matrix “B” Loaded: A Scalable Lightweight Join Query Processor for RDF Data. In Proceedings of the 19th International Conference on World Wide Web (Raleigh, North Carolina, USA) (WWW ’10). Association for Computing Machinery, New York, NY, USA, 949–959. https://doi.org/10.1145/1772690.1772769
[9] A. Azzam, S. Kirrane, and A. Polleres. 2018. Towards Making Distributed RDF Processing FLINKer. In 2018 4th International Conference on Big Data Innovations and Applications (Innovate-Data). IEEE Computer Society, Los Alamitos, CA, USA, 9–16. https://doi.org/10.1109/Innovate-Data.2018.00009
[10] Liu Baolin and Hu Bo. 2007. HPID: A High Performance RDF Database. In IFIP International Conference on Network and Parallel Computing (Lecture Notes in Computer Science), Kequi Li, Chris Jesshope, Hai Jin, and Jean-Luc Gaudiot (Eds.). Springer Berlin Heidelberg, Dalian, China, 364–374.
[11] Hamid R. Bazoobandi, Steven de Rooij, Jacopo Urbani, Annette ten Teije, Frank van Harmelen, and Henk Bal. 2015. A Compact In-memory Dictionary for RDF. In The Semantic Web. Latest Advances and New Domains. Springer-Verlag New York, Inc., Portorož, Slovenia, 205–220.
[12] David Beckett. 2001. The Design and Implementation of the Redland RDF Application Framework. In Proceedings of the 10th International Conference on World Wide Web (Hong Kong) (WWW’2001). Association for Computing Machinery, New York, NY, USA, 449â€“456. https://doi.org/10.1145/371920.372099
[13] Barry Bishop, Atanas Kiriakov, Damyan Ognyanov, Ivan Peikov, Zdravko Tashiev, and Ruslan Velkov. 2011. OWLIM: A family of scalable semantic repositories. Semantic Web 2, 1 (2011), 33–42.
[14] Christian Bizer, Jens Lehmann, Georgi Kobilarov, S. Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. 2009. DBpedia-A crystallization and Execution in Column-Oriented Database Systems. In Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (Chicago, IL, USA) (SIGMOD ‘06). Association for Computing Machinery, New York, NY, USA, 671â€“682. https://doi.org/10.1145/1142473.1142548
[15] Roi Blanco, Berkant Barla Cambazoglu, Peter Miks, and Nicolas Torzez. 2013. Entity Recommendations in Web Search. In The Semantic Web – ISWC 2013. Springer Berlin Heidelberg, Berlin, Heidelberg, 33–48.
[16] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Olivier Yakhnenko. 2013. Translating Embeddings for Modeling Multi-relational Entity Recommendations in Web Search. In The Semantic Web – ISWC 2013. Springer Berlin Heidelberg, Berlin, Heidelberg, 33–48.
[17] Barry Bishop, Atanas Kiriakov, Damyan Ognyanov, Ivan Peikov, Zdravko Tashiev, and Ruslan Velkov. 2011. OWLIM: A family of scalable semantic repositories. Semantic Web 2, 1 (2011), 33–42.
[18] Christian Bizer, Jens Lehmann, Georgi Kobilarov, S. Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. 2009. DBpedia-A crystallization and Execution in Column-Oriented Database Systems. In Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (Chicago, IL, USA) (SIGMOD ‘06). Association for Computing Machinery, New York, NY, USA, 671â€“682. https://doi.org/10.1145/1142473.1142548
[19] Roi Blanco, Berkant Barla Cambazoglu, Peter Miks, and Nicolas Torzez. 2013. Entity Recommendations in Web Search. In The Semantic Web – ISWC 2013. Springer Berlin Heidelberg, Berlin, Heidelberg, 33–48.
[20] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Ok-sana Yakhnenko. 2013. Translating Embeddings for Modeling Multi-relational
Adaptive Low-level Storage of Very Large Knowledge Graphs
WWW '20, April 20–24, 2020, Taipei, Taiwan

Systems. In Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27–31, 2014, Quebec City, Quebec, Canada. AAAI Press, Quebec, Canada, 129–137.

61. Neo4j, Inc. 2019. Neo4j Graph Platform. https://neo4j.com/ 2019.

62. Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In 27th International Conference on Data Engineering (ICDE). IEEE, Hannover, Germany, 984–994.

63. Thomas Neumann and Gerhard Weikum. 2018. The RDF-XML engine for scalable management of RDF data. The VLDB Journal 19, 1 (2010), 91–113.

64. Maximilian Nickel, Kevin Murphy, Volker Tresp, and Eyevyn Grahalkov. 2015. A review of relational learning for knowledge graphs. Proc. IEEE 103, 1 (2015), 11–33.

65. Natayla F. Noy, Nigam H. Shah, Patricia L. Whetzel, Benjamin Dai, Michael Dorf, Nicholas Griffith, Clement Jonquet, Daniel L. Rubin, Margaret-Anne Storey, and George Chute. 2009. BioPortal: ontologies and integrated data resources at the click of a mouse. Nucleic acids research 37, suppl. 2 (2009), W170–W173.

66. Objectivity Inc. 2019. InfiniteGraph. https://www.objectivity.com/products/infinitegraph/ 2019.

67. OpenLink Software. 2019. Virtuoso RDF Engine. https://virtuoso.openlinksw.com/ 2019.

68. M. Tamer Özsu. 2016. A survey of RDF data management systems.

69. OpenLink Software. 2019. Virtuoso RDF Engine. https://virtuoso.openlinksw.com/ 2019.

70. M. Tamer Ozsu. 2016. A survey of RDF data management systems. Frontiers of Computer Science 10, 3 (2016), 418–432.

71. Soumrajit Pal and Jacopo Urbani. 2017. Enhancing Knowledge Graph Completion by Embedding Correlations. In Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017. ACM, New York, NY, USA, 2247–2250.

72. Ming-Duc Pham and Peter Boncz. 2016. Exploiting Emergent Schemas to Make RDF Systems More Efficient. In The 15th International Semantic Web Conference âŠ€ SWC’2016 (Lecture Notes in Computer Science). Springer International Publishing, Kobe, Japan, 463–479.

73. Ming-Duc Pham, Linnea Passing, Orri Erling, and Peter Boncz. 2015. Deriving an Emergent Schema from RDF Data. In Proceedings of the 24th International Conference on World Wide Web (WWW ’15). International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 864–874.

74. Lu Qin, Jeffrey Xu Yu, Lijun Chang, Hong Cheng, Chengqi Zhang, and Xuemin Lin. 2014. Scalable Big Graph Processing in MapReduce. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (Snowbird, Utah, USA) (SIGMOD ’14). Association for Computing Machinery, New York, NY, USA, 827–838. https://doi.org/10.1145/2588555.2595661

75. Nicole Redaschi and UniProt Consortium. 2009. UniProt in RDF: Tackling Data Integration and Distributed Annotation with the Semantic Web. Nature Proceedings (2009).

76. Laurens Rietveld and Rinke Hoekstra. 2014. YASGUI: Feeling the Pulse of Linked Data. In Knowledge Engineering and Knowledge Management - 19th International Conference, EKAW 2014, Linköping, Sweden, November 24-28, 2014. Proceedings (Lecture Notes in Computer Science). Springer International Publishing, Linköping, Sweden, 441–452.

77. Sherif Sarker and Ghanzi Al-Naymat. 2010. Relational Processing of RDF Queries: A Survey. SIGMOD Record 38, 4 (2010), 23â–²38. https://doi.org/10.1145/1815948.1815953

78. Alexander Schatzle, Martin Pritzmayr-Zablocki, Simon Skilevic, and Georg Lausen. 2016. S2RDF: RDF Querying with SPARQL on Spark. Proceedings of the VLDB Endowment 9, 10 (2016), 804–815.

79. Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A Distributed Graph Engine on a Memory Cloud. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (New York, New York, USA) (SIGMOD ’13). Association for Computing Machinery, New York, NY, USA, 505–516. https://doi.org/10.1145/2463676.2467799

80. W. Shen, J. Wang, and J. Han. 2015. Entity linking with a knowledge base: issues, techniques, and solutions. IEEE Transactions on Knowledge and Data Engineering 27, 2 (2015), 443–460.

81. Letiferis Sidirooglou, Romulo Goncalves, Martin Kersten, Niels Ne, and Stefan Manegeold. 2008. Column-Store Support for RDF Data Management: not all swans are white. Proceedings of the VLDB Endowment 1, 2 (2008), 1553–1563.

82. Deepak Singh, Dibawal Upadhyay, and Medha Attre. 2018. Efficient RDF Dictionaries with B+ Trees. In Proceedings of the ACM India Joint International Conference on Data Science and Management of Data (CoDs-COMAD ’18). ACM, New York, NY, USA, 128–136.

83. Sparsity Technologies. 2019. Sparksee. http://sparsity-technologies.com/ 2019.

84. Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2008. YAGO: A Large Ontology from Wikipedia and WordNet. Web Semantics: Science, Services and Agents on the World Wide Web 6, 3 (2008), 203–217.

85. Systap. 2019. BlazeGraph. https://blazegraph.com/ 2019.

86. Niket Tandon, Gerhard de Melo, Fabian M. Suchanek, and Gerhard Weikum. 2014. WebCloud: Harvesting and Organizing Commonsense Knowledge from the Web. In Seventh ACM International Conference on Web Search and Data Mining, WSDM 2014. ACM, New York, NY, USA, 523–532.

87. Alberto Tonon, Michele Catasta, Roman Pagonalov, Gianluca Demartini, Karl Aberer, and Philippe Cudre-Mauroux. 2016. Contextualized ranking of entity types based on knowledge graphs. Web Semantics: Science, Services and Agents on the World Wide Web 37 (2016), 170–183.

88. Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2014. KGNCN: Efficient Encoding of Large Knowledge Graphs. In Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016. New York, NY, USA, 9-15 July 2016. AAAI Press, New York, NY, USA, 3896–3902.

89. Jacopo Urbani, Criel Jacobs, and Markus Krötsch. 2016. Column-Oriented Data-parallel Materialization for Large Knowledge Graphs. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. AAAI Press, Phoenix, Arizona, USA, 258–264.

90. Jacopo Urbani, Jason Maassen, Niels Drost, Frank Steinstra, and Henri Bal. 2013. Scalable RDF data compression with MapReduce. Concurrency and Computation: Practice and Experience 25, 1 (2013), 23–39.

91. Ruben Verborgh, Miriel Vander Sande, Odal Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haersendornck, and Pieter Colpaert. 2016. Triple Pattern Fragments: a low-cost knowledge graph interface for the Web. Semantics: Science, Services and Agents on the World Wide Web 37 (2016), 184–206.

92. Denny Vrandečić and Markus Krötsch. 2014. Wikidata: a free collaborative knowledge base. Commun. ACM 57, 10 (2014), 78–85.

93. Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: sextuple indexing for semantic web data management. Proceedings of the VLDB Endowment 1, 1 (2008), 1008–1019.

94. Hugh E. Williams and Justin Zobel. 1999. Compressing Integers for Fast File Access. Comput. J. 42, 3 (1999), 193–201. https://doi.org/10.1093/comjnl/42.3.193

95. Marcin Wiyol, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. 2018. RDF Data Storage and Query Processing Schemes: A Survey. ACM Computing Surveys (CSUR) 51, 4 (2018), 84:1–84:36.

96. Mohamed Yahya, Denilson Barbosa, Klaus Berberich, Quyue Wang, and Gerhard Weikum. 2016. Relationship Queries on Extended Knowledge Graphs. In Proceedings of the Ninth ACM International Conference on Web Search and Data Mining (WSDM ’16). ACM, New York, NY, USA, 605–614.

97. Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenyin Zhang, and Ling Liu. 2013. TripleFest: a fast and compact system for large scale RDF data. Proceedings of the VLDB Endowment 6, 7 (2013), 517–528.

98. Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A distributed graph engine for web scale RDF data. Proceedings of the VLDB Endowment 6, 4 (2013), 265–276.

99. Lei Zou, M. Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhi Huang, and Dongyan Zhao. 2014. gStore: a graph-based SPARQL query engine. The VLDB Journal 23, 4 (2014), 565–590.