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:

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 [17, 57], encyclopedic or commonsense knowledge [77, 84], etc.), and are actively used to enhance tasks such as entity recognition [72], query answering [88], or more in general Web search [14, 28, 29].

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 [36], enrich the results using embeddings of the graph [56], and then compute some centrality metrics for ranking the answers [41, 78]. 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 [18, 25, 26, 31, 32, 47, 66, 71], 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 [63]. 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 [63] supports graph analytics, RDF engines like Virtuoso [59] or RDFox [52] focus on SPARQL [36]). 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 light-weight 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 [55], VLog [81], SNAP [45]) 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 contribu-
tions 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 aggre-
gated to save further space. The adaptation of the physical
storage is, as far as we know, a unique feature which is par-
ticularly 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 main-
taining 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.
An extended version of this paper is available at [80].

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
the 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 $(ts, tr, td)$ where $ts, tr,$ and
$td$ are the IDs associated to $s, r,$ and $d$ respectively.

The numerical IDs allow us to sort the edges and by permuting
$ts, tr, td$ 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., $srdr$ specifies that the edges are ordered by source,
relation, and destination. We denote with $\mathcal{R} = \{srdr, sdr, \ldots\}$ 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 $\text{isprefix}$ to check
whether string $a$ is a prefix of $b$, i.e., $\text{isprefix}(a, b) = [\text{true}]$ and the operator $-$ to remove all characters of one string from
another one (e.g., if $a = \text{srdr}$ and $b = \text{sd}$, then $a – b = r$).

Let $\mathcal{V}$ be a set of variables. A simple graph pattern (or triple
pattern) is an instance of $L \cup \mathcal{V} \times L \cup \mathcal{V} \times L \cup \mathcal{V}$ and we denote
it as $(X, Y, Z)$ where $X, Y, Z \in L \cup \mathcal{V}$. A graph pattern is a finite
set of simple graph patterns. Let $\sigma : \mathcal{V} \rightarrow \mathcal{L}$ 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 $q$, the answers for $q$ on $G$ correspond to
the set $\text{ans}(G, q) = \{r(s, d) \in E \land qa = (s, r, d)\}$. Function
$\text{bound}(p)$ returns the positions of the labels in the simple graph
pattern $p$ left-to-right, i.e., if $p = (X, a, b)$ where $X \in \mathcal{V}$ and $a, b \in \mathcal{L}$,
then $\text{bound}(p) = rd$.

A Knowledge Graph (KG) is a directed labeled graph where nodes
are entities and edges establish semantic relations between them,
e.g., (Sadig, Khan, majorOf, London). Usually, KGs are published
on the Web using the RDF data model [40]. In this model, data is
represented as a set of triples of the form $(subject, predicate, object)$
drawn from $(I \cup B) \times I \times (I \cup L)$ where $I, B, L$ denote sets of
IRIs, blank nodes and literals respectively. Let $T = I \cup B \cup L$ 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 $L = T$.

3 GRAPH PRIMITIVES

We start our discussion with a description of the low-level prim-
itives that we wish to support. We distilled these primitives con-
sidering four types of workloads: SPARQL [36] 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 dis-
tribution, or within more complex pipelines; Statistical relational
models [56], which are effective techniques to make predictions
using the KG as prior evidence.

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 opera-
tions 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 answer-
ing, 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.

- $f_1 = f_2$. 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.

- $f_5 = f_6$. Function $\text{edg}_{\omega}(G, p)$ 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.

16. This group of primitives returns an aggregated version of the output of f5 – f10. For instance, grp(G,p) returns the list \((x_1, c_1), \ldots, (x_n, c_n)\) of all distinct sources in the edges ans(G, p) with the respective counts of the edges that share them. Let D be a set of edges, \(A(x, D) = \{r(s, d) \in D\}\) and \(B(D) = \{(s, r) | r(s, d) \in A(s, D) \land c = |A(s, D)|\}\). Then, grp(G, p) returns the list of all tuples in \(B(ans(G, p))\) sorted by the numerical ID of the first field. The other primitives are defined analogously.

17. This primitive returns the cardinality of the output of f5, …, f10. This computation is useful in a number of cases: For instance, it can be used to optimize the computation of SPARQL queries by rearranging the join ordering depending on the cardinalities of the triple patterns or to compute the degree of nodes in the graph.

18. These primitives return the \(i^{th}\) edge that would be returned by the corresponding primitives edg. In practice, this operation is needed in several graph analytics algorithms or for minibatching during the training of statistical relational models.

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 \(\text{DICT}_l\), for \(label \Rightarrow ID\). Using B+Trees here is usual, so we will not discuss it further. Note that Trident 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 [56].

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. More specifically, let us assume that we must store the graph \(G = (V, E, l, \phi_E, \phi_V)\). For each \(l \in L\), we consider three types of subsets: \(E_s(l) = \{r(l, d) \in E\}, E_r(l) = \{l(s, d) \in E\}, E_d(l) = \{r(s, l) \in E\}\), 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 \textit{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 \(F_s(l) = \{r, d) \in (l, d) \in E\}\) and

\[ f_{18} \text{pos}_s(d, p, i) \] \(=\) \(i^{th}\) edge returned by \(\text{edg}_s(d, p)\).

\[ f_{19} \text{pos}_s(d, p, i) \] \(=\) \(i^{th}\) edge returned by \(\text{edg}_s(d, p)\).

\[ f_{20} \text{pos}_s(d, p, i) \] \(=\) \(i^{th}\) edge returned by \(\text{edg}_s(d, p)\).

\[ f_{21} \text{pos}_s(d, p, i) \] \(=\) \(i^{th}\) edge returned by \(\text{edg}_s(d, p)\).

\[ f_{22} \text{pos}_s(d, p, i) \] \(=\) \(i^{th}\) edge returned by \(\text{edg}_s(d, p)\).

\[ f_{23} \text{pos}_s(d, p, i) \] \(=\) \(i^{th}\) edge returned by \(\text{edg}_s(d, p)\).

Table 1: Graph primitives

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f_1) (l_nod(e, G, r))</td>
<td>Label of node (n) that equals to (\phi_V(r(n))).</td>
</tr>
<tr>
<td>(f_2) (l_nod(e, G, r))</td>
<td>Label of edge (e) that equals to (\phi_E(e)).</td>
</tr>
<tr>
<td>(f_3) (n_{od}(G, l))</td>
<td>(l), i.e., the ID of node with label (l).</td>
</tr>
<tr>
<td>(f_4) (n_{edg}(G, l))</td>
<td>(l), i.e., the ID of edge label (l).</td>
</tr>
<tr>
<td>(f_5) (e_{od}(G, p))</td>
<td>(a(s, p)) sorted by (\text{trd}).</td>
</tr>
<tr>
<td>(f_6) (e_{od}(G, p))</td>
<td>(a(s, p)) sorted by (\text{trd}).</td>
</tr>
<tr>
<td>(f_7) (e_{od}(G, p))</td>
<td>(a(s, p)) sorted by (\text{dxi}).</td>
</tr>
<tr>
<td>(f_8) (e_{od}(G, p))</td>
<td>(a(s, p)) sorted by (\text{dxi}).</td>
</tr>
<tr>
<td>(f_9) (e_{od}(G, p))</td>
<td>(a(s, p)) sorted by (\text{rsd}).</td>
</tr>
<tr>
<td>(f_{10}) (e_{od}(G, p))</td>
<td>(a(s, p)) sorted by (\text{rsd}).</td>
</tr>
<tr>
<td>(f_{11}) (y_{od}(G, p))</td>
<td>All (s) of (a(s, p)).</td>
</tr>
<tr>
<td>(f_{12}) (y_{od}(G, p))</td>
<td>All (r) of (a(s, p)).</td>
</tr>
<tr>
<td>(f_{13}) (y_{od}(G, p))</td>
<td>All (d) of (a(s, p)).</td>
</tr>
<tr>
<td>(f_{14}) (y_{od}(G, p))</td>
<td>All (r) of (a(s, p)).</td>
</tr>
<tr>
<td>(f_{15}) (y_{od}(G, p))</td>
<td>All (d) of (a(s, p)).</td>
</tr>
<tr>
<td>(f_{16}) (y_{od}(G, p))</td>
<td>All (r) of (a(s, p)).</td>
</tr>
<tr>
<td>(f_{17}) (y_{od}(G, p))</td>
<td>All (d) of (a(s, p)).</td>
</tr>
<tr>
<td>(f_{18}) (y_{od}(G, p))</td>
<td>All (r) of (a(s, p)).</td>
</tr>
<tr>
<td>(f_{19}) (y_{od}(G, p))</td>
<td>All (d) of (a(s, p)).</td>
</tr>
<tr>
<td>(f_{20}) (y_{od}(G, p))</td>
<td>All (r) of (a(s, p)).</td>
</tr>
<tr>
<td>(f_{21}) (y_{od}(G, p))</td>
<td>All (d) of (a(s, p)).</td>
</tr>
<tr>
<td>(f_{22}) (y_{od}(G, p))</td>
<td>All (r) of (a(s, p)).</td>
</tr>
<tr>
<td>(f_{23}) (y_{od}(G, p))</td>
<td>All (d) of (a(s, p)).</td>
</tr>
</tbody>
</table>
We now discuss how we can implement the primitives in Table 1 with our architecture.

**Primitives $f_1, \ldots, f_4$ ($\text{lbl}_s$, $\text{nodid}$, $\text{edgid}$).** These primitives are executed consulting $\text{DICT}_1$ or $\text{DICT}_2$. Thus, the following holds.

**Proposition 1.** Let $G = (V, E, L, \phi_V, \phi_E)$. The time complexity of computing $f_1, \ldots, f_4$ is $O(\log(|L|))$.

**Primitives $f_5, \ldots, f_{10}$ ($\text{edgid}_s$).** Let $\text{edgid}_s(G, p)$ be a generic invocation of one of $f_5, \ldots, f_{10}$. First, we need to retrieve the numerical IDs associated to the labels in $p$ (if any). Then, we select an ordering that: 1) allows us to retrieve the answers for $p$ with a range scan; 2) sorts the answers in a way that complies with $\omega$. The orderings that satisfy 1) are

$$
\Omega_p = \{ \omega' \mid \omega' \in \mathcal{R} \land \text{isprefix}(\text{bound}(p), \omega') = \text{true} \} \tag{1}
$$

An ordering $\omega' \in \Omega_p$ which also satisfies 2) is one for which $\omega' - \text{bound}(p) = \omega - \text{bound}(p)$.

**Example 1.** Consider the execution of $\text{edgid}_s(G, p)$ where $p = (X, Y, a)$. In this case, $\text{bound}(p) = d$, $\Omega_p = \{ \text{drr}, \text{dsr} \}$ and $\omega' = \text{dss}$.

The selected $\omega'$ is associated to one byte stream. If $p$ contains one or more constants, then we can query NM to retrieve the appropriate binary table from that binary stream and (range-)scan it to retrieve the answers of $p$. In contrast, if $p$ only contains variables, the results can be obtained by scanning all tables in the byte stream. Note that the cost of retrieving the IDs for the labels in $p$ is $O(\log(|L|))$ since we use B+Trees for the dictionary. This is an operation that is applied any time the input contains a graph pattern. If we ignore this cost and look at the remaining computation, then we can make the following observation.

**Proposition 2.** Let $G = (V, E, L, \phi_V, \phi_E)$. The time complexity of $\text{edgid}_s(G, p)$, where $p$ contains variables, is $O(\log(|L|) + |E|)$ otherwise.

**Primitives $f_{11}, \ldots, f_{16}$ ($\text{grp}_s$).** Let $\text{grp}_s(G, p)$ be a general call to one of these primitives. Note that in this case $\omega \in \mathcal{R}'$, i.e., is a partial ordering. These functions can be implemented by invoking $f_5, \ldots, f_{10}$ and then returning an aggregated version. Thus, they have the same cost as the previous ones.

However, there are special cases where the computation is quicker, as shown in the next example.

**Example 2.** Consider a call to $\text{grp}_s(G, p)$ where $p = (a, X, Y)$. 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 3.** Let $G = (V, E, L, \phi_V, \phi_E)$. The time complexity of $\text{grp}_s(G, p)$ ranges between $O(\log(|L|))$ and $O(\log(|L|) + |E|)$ depending on $p$ and $\omega$.

**Primitive $f_{17}$ (count).** This primitive returns the cardinality of the output of $f_5, \ldots, f_{16}$. Therefore, it can be simply implemented by iterating over the results returned by these functions. However, there are cases when we cannot avoid this iteration. Some of such cases are the ones described below:
• C1 The input is $\text{edg}_{op}(G,p)$ and $p$ contains no constant nor repeated variables. In this case the output is $|E|$.
• C2 The input is $\text{edg}_{op}(G,p)$ and $p$ contains only one constant $c$ and no repeated variables. Then, the cardinality is stored in $M_c$.
• C3 The input is $\text{grp}_{op}(G,p)$, there is an ordering $\alpha' \in \Omega_p$ such that $\text{isprefix}(\alpha, \alpha') = \text{true}$, and $p$ contains at most one constant and no repeated variables. Then, the output can be computed either by consulting NM or the metadata of one byte stream.

Otherwise, we also need to access one binary table to compute the results, which, in the worst case, takes $O(|E|)$.

**Proposition 4.** Let $G = (V, E, L, \phi_V, \phi_E)$. The time complexity of executing $\text{count}()$ ranges between $O(\log(|L|))$ and $O(|L| + |E|)$.

**Primitives** $f_{18}, \ldots, f_{23}$ ($\text{pos}_p$). In order to efficiently support these primitives, we need to provide a fast random access to the edges. Given a generic $\text{pos}_p(G,p,i)$, we distinguish four cases:

• C4 If $p$ contains repeated variables, then we iterate over the results of $\text{edg}_{op}(G,p)$ and return the $i^{th}$ edge;
• C5 If $p$ contains only one constant, then the search space is restricted to a single binary table. In this case, the computation depends on how the content of the table is serialized on the byte stream. If it allows random access to the rows, then the cost reduces to $O(\log(|L|))$, i.e., query NM. Otherwise we also need to iterate through the table and count until the $i^{th}$ row;
• C6 If $p$ contains more than one constant, then we need to perform either a linear or binary search through the table to identify the portion of the table with the answers and retrieve the $i^{th}$ row;
• C7 Finally, if $p$ does not contain any constants or repeated variables, then we must consider all edges stored in one byte stream. In this case, we first search for the binary table that contains the $i^{th}$ edge. This operation requires a scan of the metadata associated to the byte stream, which can take up to $O(|L|)$. Then, the complexity depends on whether the physical storage of the table allows a random access, as in C5 and C6. If it does not, then the worst-case in terms of complexity is $O(|L| + |E|)$. Note that in this case, simply going through all edges would be faster ($O(|E|)$). However, in practice tables have more than one row so we can advance more quickly despite the higher worst-case complexity.

**Proposition 5.** Let $G = (V, E, L, \phi_V, \phi_E)$. The time complexity of executing $\text{pos}_p(G,p,i)$ 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 [44] (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 [82] to work in a multi-core environment. This technique first deconstructs the triples, then assigns unique IDs to all the terms, and finally reconstruct 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 presorted 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 [27] that discourages in-place updates, and it is inspired by the idea of differential indexing [55], 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 an extra dictionary 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 $T = \langle (t'_1, t''_1), \ldots, (t'_m, t''_m) \rangle$ 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.

### 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 $T$ 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 $\phi$ 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

```
Algorithm 1: selectlayout(T)
1 U := \{u \mid (u, v) \in T\}
2 if |T| \leq \tau and |U| \leq \phi then
3 m₁ := 0, m₂ := 0, m₃ := 0
4 foreach u ∈ U do
5 Z := \{v \mid (u, v) \in T\}
6 if u > m₁ then m₁ := u
7 if |Z| > m₂ then m₂ := |Z|
8 foreach z ∈ Z do
9 if z > m₃ then m₃ := z
10 t₁ := |U| * (sizeof(m₁) + sizeof(m₂)) + |T| * sizeof(m₃)
11 tₐ := |T| * (sizeof(m₁) + sizeof(m₂))
12 if t₁ < tₐ then
13 return (RO, sizeof(m₁), sizeof(m₂), 0)
14 else return (CLUSTER, sizeof(m₁), sizeof(m₂), sizeof(m₃))
15 else return (COLUMN, 5, 5, 0)
```

**Column layout.** With this layout, the elements in $T$ are serialized as $(t'_1, \ldots, t''_1), (t'_2, \ldots, t''_2)$. 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 $(t'_1, \ldots, t''_1)$. In fact, if $t'_1 = t'_2 = \ldots = t'_m$, then we can simply write $t'_1 \times n$. Also, this layout allows binary search and a 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 return the entire pair. On the other hand, this layout is more suitable than the row layout for aggregate reads (required, for instance, for executing gpr primitives) because in this case we only need to read the content of one column which is stored at contiguous locations.

**Cluster layout.** Let $g_1 = (t'_1, t''_1), \ldots, (t'_m, t''_m)$ be the longest sub-sequence of pairs in $T$ which share the first term $t$. With this layout, all groups are first ordered in the sequence $(g_1, \ldots, g_t, g_{t+1}, \ldots, g_m)$ such that $t_1 \leq t_2 \leq \ldots \leq t_m$ for all $1 \leq i < m$. Then, they are serialized one-by-one. Each group $g_i$ is serialized by first writing $t$, then $|g_i|$, and finally the list $t'_1 \times 1, \ldots, t'_m \times 1$. 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 $\lceil \log 2 \times n \rceil$ bits. Another disadvantage is that with this layout binary search is only possible within one group.
the cluster layout should not be used due to the lack of support of 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 $\phi$ 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 sizeof($\cdot$) 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 [55]. Note that more complex compression schemes could also be used (e.g., VByte [86]) 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 $m_n$ 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 $T_k$ maps to another table in $T'_k$ 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.

<table>
<thead>
<tr>
<th>Type</th>
<th>#Edges</th>
<th>#Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBSP</td>
<td>KG</td>
<td>Var</td>
</tr>
<tr>
<td>DBpedia</td>
<td>KG</td>
<td>1B</td>
</tr>
<tr>
<td>YAGO2S</td>
<td>KG</td>
<td>1B</td>
</tr>
<tr>
<td>Google</td>
<td>KG</td>
<td>16AM</td>
</tr>
<tr>
<td>Twitter</td>
<td>KG</td>
<td>1B</td>
</tr>
<tr>
<td>BTC2012</td>
<td>KG</td>
<td>1B</td>
</tr>
</tbody>
</table>

**Table 2: Details about the used datasets**

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 $T'_k$ which has less than $\eta$ rows, $\eta$ 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 [85]. Here, we limit their usage to the tables in $T'_k$ if they lead to a storage space reduction.

To illustrate the main idea, consider a generic table $t$ that contains the set of tuples $T'_k(isA)$. 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 $T'_k$ 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 [55].** 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 [45].** 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 [81].** 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 (BGP's) that applies greedy query optimization based on cardinalities, and uses either merge or index loop joins.

**Testbed.** We used a Linux machine (kernel 3.10, GCC 6.4, page size 4k) with dual Intel E5-2630v3 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 [89], a in-memory state-of-the-art RDF database (in contrast to RDF3X which uses
These differences show to what extent Trident adapted its physical storage to the structure of the KG. 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 with the row layout. We considered five types of row 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 [68]).

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 TR’ 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 two 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 type 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 one constant (e.g., return all objects of the predicate type). These two patterns are answered with gcc.

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 the 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 [68]).

6.2 SPARQL

Figure 4a shows the average of five warm runtimes with our system and with other state-of-the-art engines. For LUBM8k, DBpedia, Uniprot, and BTC2012, we considered some of the queries used to evaluate TripleBit [89]. For Wikidata, we designed example queries of various complexity looking at published examples. All queries are available at [80]. Unfortunately, we could not load Wikidata and BTC2012 with SYSTEM_A due to errors during the loading phase.

We can make a few observations from the obtained results. First, a direct comparison against TripleBit is problematic because...
(b) Runtime of graph analytics algorithms (ms)

![Figure 4: Performance of our storage engine on several tasks, with larger KGs and updates](image)

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 node-centric 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 MaxSCC compute the largest weak and strong connected components respectively; Diameter computes the diameter of the graph while ClustCoeff computes the clustering coefficient.

We executed these algorithms using the original SNAP library in combination with our engine. Note that the implementation of the algorithms is the same; only the storage changes. Figure 4b 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) [52, 81]. Then, we repeated the process with the native storage of VLog. The runtime, reported in Figure 4c, 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. We implemented TransE [15], 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 [33], a state-of-the-art library. Figure 4c reports the runtime to train a model using as input a subset of YAGO which was used in other works [61]. The results indicate competitive performance also in this case.

6.4 Scalability, updates, and bulk loading

We executed the five LUBM queries using our native SPARQL procedure on KGs of different sizes (between 1B-100B triples). We used another machine with 256GB of RAM for these experiments (which also costs $<5$K due to lack of disk space. The warm runtimes are shown in Figure 4e. The runtime of the first two queries remains constant. This was expected since their selectivity does not decrease as the size of the KG increases. In contrast, the runtime of the other queries increases as the KG becomes larger.

Figure 4d shows the runtime of four SPARQL queries after we added five sets of about 1M new triples to the KG, merged them
into single updates, removed five other sets of 1M triples, and then merged again. We selected the queries so that the content of the updates is considered. We observe that the runtime increases (because more deltas are considered) and that it drops after they are merged in a single update. Figure 5a reports the runtime to process five additions of ca. 1M novel triples, one merge, five removals of ca. 1M existing triples, and another merge. As we can see, with both datasets the runtime is much smaller than re-creating the database from scratch (>1h). The runtime with LUBM8k is faster than with Wikidata because the updates with the latter KG contained 4X more new entities.

In Figure 5b, we show the trace of the resource consumption during the loading of LUBM80k (10B triples). We plot the CPU (100% means all physical cores are used) and RAM usage. From it, we see that most of the runtime is taken to dictionary encoding, sorting the edges, and to create the binary tables.

In general, Trident has competitive loading times. Figure 5c shows the loading time of ours and other systems on LUBM1k. With larger KGs, RDF3X becomes significantly slower than ours (e.g., it takes ca. 7 hours to load LUBM8k on our smaller machine while Trident needs 1 hour and 18 minutes) due to a lack of parallelism. TripleBit is an in-memory database and thus it cannot scale to some very large graphs.

Another main novelty is that the storage effectively adapts to the graph topology. Our empirical evaluation, performed using a prototype called Trident, shows that our system is adaptable to different layout settings to exploit the multi-table nature of relational engines. Existing partitioning criteria group the triples either by predicates [2, 35, 46, 48, 54, 64, 65], clusters of predicates [19], or by using other entity-based splitting criteria [16]. The various partitioning schemes are designed to create few tables to meet the constraints of relational engines [73]. 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 [59], BlazeGraph [76], Titan [20], Neo4j [53] Sparksee [74], and InfiniteGraph [58]. We compared Trident against such a leading commercial system and observed that ours has very competitive performance; other comparisons are presented in [5, 73]. 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, 31, 34, 35, 43, 62, 70, 90]. 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 a distributed processing of very large graphs.

8 CONCLUSION

In this paper, 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, performed using a prototype called Trident, shows that our approach is competitive in multiple scenarios 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 [83], an emerging paradigm to query RDF datasets, and GraphQL [39], 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 [79] or HDT [23], can further improve the runtime and/or reduce the storage space.

Acknowledgments. We would like to thank (in alphabetical order) Peter Bonec, 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 (VWDdata) and NWO VENI project 639.021.335. Some experiments presented in this paper were performed on the clusters DA55 and SCILENS funded by NWO grants.

REFERENCES


