Revisiting RDF storage layouts for efficient query answering

M. Buron[1,2], F. Goasdoué[3], I. Manolescu[1,2], T. Merabti[1,2], and M.-L. Mugnier[4]

[1] Inria `firstname.lastname@inria.fr`
[2] Institut Polytechnique de Paris, France
[3] Univ Rennes, CNRS, IRISA, France `fg@irisa.fr`
[4] Univ Montpellier, LIRMM, Inria, France `mugnier@lirmm.fr`

Abstract. The performance of query answering in an RDF database strongly depends on the data *layout*, that is, the way data is split in persistent data structures. We consider answering Basic Graph Pattern Queries (BGPQs), and in particular those with variables (also) in class and property positions, in the presence of RDFS ontologies, both through data saturation and query reformulation. We show that such demanding queries often lead to inefficient query answering on two popular storage layouts, so-called T and CP. We present *novel query answering algorithms on the TCP layout*, which combines T and CP. In exchange to occupying more storage space, e.g. on an inexpensive disk, TCP avoids the bad or even catastrophic performance that T and/or CP sometimes exhibit. We also introduce *summary-based pruning*, a novel technique based on existing RDF quotient summaries, which improves query answering performance on the T, CP and the more robust TCP layouts.

1 Introduction

We consider the problem of efficiently querying an **RDF database**, which stores RDF graphs persistently (e.g., on a disk) and allows queries and updates on the graphs, possibly concurrently by several users. We are interested in **answering** queries on a graph, taking into account an **RDF Schema** (RDFS, in short) ontology and associated **RDFS entailment**. We consider **general SPARQL conjunctive queries** (a.k.a. basic graph pattern queries, or BGPQs), which *allow variables in any subject, property, or object position* of query triples. For instance, in the query $q(x, u) \leftarrow (x, \text{:name}, \text{:Alice}), (x, y, z), (z, \text{rdf:type}, u)$, where x, y, z, u are variables and the other terms are IRIs, the property y in the second triple is a variable, just like u which is the type of z. Such queries allow to fully take advantage of the freedom RDF provides: one does not need to know the relation between x and z, nor the exact type of z, to query the graph.

Answering a BGPQ in an RDF database requires *translating* it into a description of work that the execution engine must perform; without loss of generality, we call this work description a *query plan*, as is common in the database literature. Specifically, we distinguish a *logical plan* specifying the operations to use to answer the query, from a *physical (executable) plan*, derived from the logical one with the help of statistics and cost parameters characterizing the data (size, value frequencies etc.) and the execution environment (hardware etc.)

Both plans start by accessing some data from the store and continue with various other processing steps (e.g., filtering, combining multiple inputs etc.). The set of

persistent data structures that hold the data of an RDF graph in the database are called *storage layout*. When a set of frequent BGPQs are known in advance, they can be used to design a workload-aware layout, which optimizes data access for these queries, e.g., [7,15,20]. Lacking a known query set, a workload-unaware layout is used, with the two most popular being: **T (triple)**, e.g., [8,25,19], which stores all triples together as a single (s, p, o) (subject, predicate, object) collection; and **CP (class and property)** [4], which separates the data for each property and class, i.e., as (i) a collection of (s, o) pairs for every property p, and (ii) a collection of all the resources s that have a given type c in the graph. Indexes can be added to both the T and CP layouts.

In this work, we focus on **translating BGPQs**, including general ones, for **query answering under RDFS** ontologies and entailment, into **logical plans**, on **workload-unaware layouts**. We target logical plans for generality, since physical plans strongly depend on the RDF database implementation, the presence of indexes etc.; these decisions are best left to the optimization and execution layer, and we do not study them here. However, as we will show, *the choice of the logical plan can massively impact the space of alternatives (physical plans)* considered for the query, and thus the query answering performance. In particular, we translate our plans in SQL (if the RDF database has a relational back-end) or SPARQL (if a native RDF engine evaluates them), which enables to retain all benefits of system-specific optimization. We study translation for the **two classical ways of taking the ontology into account for query answering**: by materializing entailed triples in the RDF graph (*graph saturation*) or by compiling relevant parts of the ontology in the query, which yields a union of BGPQs (*query reformulation*). Our contributions are the following:

1. We introduce the *novel workload-unaware TCP layout*, which combines the data structures of both T and CP, and an associated algebraic translation of BGPQs into logical plans over TCP.
2. We introduce *summary-based pruning*, an optimization technique of independent interest, that reduces query answering costs on the T, CP and TCP layouts, both when using graph saturation and query reformulation.
3. We *experimentally validate the performance benefits* of the TCP layout and translation, and of summary-based pruning, on a relational and a native RDF database. Our experiments are detailed online[5] with the code and data necessary in order to reproduce them.

Below, after the preliminaries, Section 3 recalls algebraic query translations for the T and CP layouts. We explain why naïve algebraic translation on CP leads to *poor performance*, not only for *general* BGPQs (as noted since [21]), but also for *reformulated* ones (whether general or not). This is due to *interleaved joins and unions*, which limit the optimization opportunities in the RDF database. At the same time, the T layout entails repeated self-joins of the whole triple set, degrading performance on large graphs and complex queries (this motivated introducing the CP layout [4]). Section 4 presents our technical contributions and Section 5 our experiments. We then discuss related works and conclude.

[5] `https://gitlab.inria.fr/mburon/graph-layout-experiments`

RDFS constraint	Schema triple notation	**RDFS constraint**	Schema triple notation
Subclass	$(\mathtt{s}, \prec_{sc}, \mathtt{o})$	Subproperty	$(\mathtt{s}, \prec_{sp}, \mathtt{o})$
Domain typing	$(\mathtt{s}, \hookleftarrow_{d}, \mathtt{o})$	Range typing	$(\mathtt{s}, \hookrightarrow_{r}, \mathtt{o})$

RDF assertion	Data triple notation
Class assertion	$(\mathtt{s}, \tau, \mathtt{o})$
Property assertion	$(\mathtt{s}, \mathtt{p}, \mathtt{o})$ with $\mathtt{p} \notin \{\tau, \prec_{sc}, \prec_{sp}, \hookleftarrow_{d}, \hookrightarrow_{r}\}$

Table 1. RDF statements.

$G_{\text{ex}} =$
(:OpenArt, $\prec_{sc}$, :Article), (:GOpenArt, $\prec_{sc}$, :OpenArt), (:Prof, $\prec_{sc}$, :Person),
(:teaches, $\hookleftarrow_{d}$, :Prof), (:author, $\hookrightarrow_{r}$, :Person), (:firstAuth, $\prec_{sp}$, :author),

(:art1, :title, *"RDF storage"*), (:Alice, :name, *"Alice"*)(:art1, :firstAuth, :Alice),
(:Alice, :teaches, :algo101), (:art1, :author, :Bob), (:Bob, :name, *"Bob"*),
(:art1, rdf:type, :GOpenArt)

Fig. 1. Sample RDF graph G_{ex} (schema triples on top and data triples below).

Name [2]	Entailment rule	**Name**	Entailment rule
`rdfs5`	$(\mathtt{p}_1, \prec_{sp}, \mathtt{p}_2), (\mathtt{p}_2, \prec_{sp}, \mathtt{p}_3) \rightarrow (\mathtt{p}_1, \prec_{sp}, \mathtt{p}_3)$	`ext4`	$(\mathtt{p}, \prec_{sp}, \mathtt{p}_1), (\mathtt{p}_1, \hookrightarrow_{r}, \mathtt{o}) \rightarrow (\mathtt{p}, \hookrightarrow_{r}, \mathtt{o})$
`rdfs11`	$(\mathtt{s}, \prec_{sc}, \mathtt{o}), (\mathtt{o}, \prec_{sc}, \mathtt{o}_1) \rightarrow (\mathtt{s}, \prec_{sc}, \mathtt{o}_1)$	`rdfs2`	$(\mathtt{p}, \hookleftarrow_{d}, \mathtt{o}), (\mathtt{s}_1, \mathtt{p}, \mathtt{o}_1) \rightarrow (\mathtt{s}_1, \tau, \mathtt{o})$
`ext1`	$(\mathtt{p}, \hookleftarrow_{d}, \mathtt{o}), (\mathtt{o}, \prec_{sc}, \mathtt{o}_1) \rightarrow (\mathtt{p}, \hookleftarrow_{d}, \mathtt{o}_1)$	`rdfs3`	$(\mathtt{p}, \hookrightarrow_{r}, \mathtt{o}), (\mathtt{s}_1, \mathtt{p}, \mathtt{o}_1) \rightarrow (\mathtt{o}_1, \tau, \mathtt{o})$
`ext2`	$(\mathtt{p}, \hookrightarrow_{r}, \mathtt{o}), (\mathtt{o}, \prec_{sc}, \mathtt{o}_1) \rightarrow (\mathtt{p}, \hookrightarrow_{r}, \mathtt{o}_1)$	`rdfs7`	$(\mathtt{p}_1, \prec_{sp}, \mathtt{p}_2), (\mathtt{s}, \mathtt{p}_1, \mathtt{o}) \rightarrow (\mathtt{s}, \mathtt{p}_2, \mathtt{o})$
`ext3`	$(\mathtt{p}, \prec_{sp}, \mathtt{p}_1), (\mathtt{p}_1, \hookleftarrow_{d}, \mathtt{o}) \rightarrow (\mathtt{p}, \hookleftarrow_{d}, \mathtt{o})$	`rdfs9`	$(\mathtt{s}, \prec_{sc}, \mathtt{o}), (\mathtt{s}_1, \tau, \mathtt{s}) \rightarrow (\mathtt{s}_1, \tau, \mathtt{o})$

Table 2. Sample set $\mathcal{R}$ of RDFS entailment rules.

2 Preliminaries

We rely on three pairwise disjoint sets of values: the sets $\mathscr{I}$ of IRIs (resource identifiers), $\mathscr{L}$ of literals (constants) and $\mathscr{B}$ of blank nodes modeling unknown IRIs or literals. A triple $(\mathtt{s}, \mathtt{p}, \mathtt{o}) \in (\mathscr{I} \cup \mathscr{B}) \times \mathscr{I} \times (\mathscr{L} \cup \mathscr{I} \cup \mathscr{B})$ states that its *subject* $\mathtt{s}$ has the *property* $\mathtt{p}$ with the *object* value $\mathtt{o}$ [1]. We denote by $\text{Val}(G)$ the set of all values (IRIs, blank nodes and literals) occurring in an RDF graph G. In triples, we use $_{:}b$ (possibly with indices) to denote blank nodes, and strings between quotes to denote literals.
We distinguish **schema** triples from **data** ones. The former state RDF Schema (RDFS) constraints on classes and properties: **subclass** (specialization relation between classes), **subproperty** (specialization relation between properties), **domain** (typing of the first attribute of a property), and **range** (typing of the second attribute of a property). An *RDFS ontology* (or ontology, in short) is a set of schema triples. The *ontology of an RDF graph G* is the set of schema triples of G. The other triples, i.e., data triples, describe data by typing resources with classes or stating how resources are related through properties. Table 1 introduces our short notations for schema and data triples.

Example 1 (Running example). Figure 1 shows a sample graph G_{ex}, describing articles and their authors; some articles are Open Access (:OpenArt), a subclass of which are Green Open Access ones (:GOpenArt).

An *entailment rule* (or simply *rule*) r has the form $\text{body}(r) \rightarrow \text{head}(r)$, where $\text{body}(r)$ and $\text{head}(r)$ are RDF graphs, respectively called *body* and *head* of the

rule r. In this work, we consider the set of **RDFS entailment rules** $\mathcal{R}$ shown in Table 2, which allow reasoning with an RDFS ontology; in the table, all values except RDF reserved IRIs are blank nodes. These rules either combine schema triples to entail schema triples (`rdfs5`, `rdfs11`, `ext1` to `ext4`), or combine schema triples together with data triples to entail data triples (`rdfs2`, `rdfs3`, `rdfs7` and `rdfs9`). The **saturation** of a graph G with the rule set $\mathcal{R}$, denoted $G^{\mathcal{R}}$, allows materializing its semantics, by iteratively augmenting it with the triples it entails using $\mathcal{R}$, until reaching a fixpoint; this process is finite [2].

Example 2. The saturation of G_{ex} with $\mathcal{R}$ (Table 2) is: $(G_{\text{ex}})^{\mathcal{R}} = G_{\text{ex}} \cup \{(\text{:GOpenArt}, \prec_{sc}, \text{:Article}), (\text{:teaches}, \hookleftarrow_d, \text{:Person}), (\text{:firstAuth}, \hookrightarrow_r, \text{:Person}), (\text{:Alice}, \tau, \text{:Prof}), (\text{:Bob}, \tau, \text{:Person}), (\text{:art1}, \text{:author}, \text{:Alice}), (\text{:art1}, \tau, \text{:OpenArt}), (\text{:Alice}, \tau, \text{:Person}), (\text{:art1}, \tau, \text{:Article})\}$

Given a set of variables $\mathscr{V}$ disjoint from $\mathscr{I} \cup \mathscr{B} \cup \mathscr{L}$, a **basic graph pattern** (BGP in short) is a set of *triple patterns* (or triples in short, when non-ambiguous) belonging to $(\mathscr{I} \cup \mathscr{B} \cup \mathscr{V}) \times (\mathscr{I} \cup \mathscr{V}) \times (\mathscr{I} \cup \mathscr{B} \cup \mathscr{L} \cup \mathscr{V})$. The set of variables (resp. values: IRIs, blank nodes, literals and variables) occurring in a BGP P is denoted by Var(P) (resp. Val(P)). Note that **a variable may occur in any position** of a triple pattern. In particular, we say that a variable x occurs in *property position* for a triple of the form $(-, x, -)$ and in *class position* for a triple of the form $(-, \tau, x)$.

Definition 1 (BGPQ). *A* BGPQ q *is of the form* $q(\bar{x}) \leftarrow P$*, where* P *is a BGP (also denoted* body(q)*) and* $\bar{x} \subseteq \text{Var(P)}$ *are the* answer variables *of* q*.*

For simplicity, below, we will assume that BGPQs have no blank nodes, as it is well-known that these can be replaced by non-answer variables [3]. We also consider a slight generalization of BGPQs, namely **partially instantiated** BGPQs: such queries are obtained from BGPQs by substituting *some* variables with values from $\mathscr{I} \cup \mathscr{L} \cup \mathscr{B}$. For simplicity again, we will not distinguish between a standard and a partially instantiated BGPQ. The semantics of a BGPQ on a graph is defined through homomorphisms from the query body to the graph:

Definition 2 (BGP-to-RDF homomorphism). *A* homomorphism *from a BGP* P *to an RDF graph* G *is a function* φ *from* Val(P) *to* Val(G) *such that for any triple* $(s, p, o) \in P$*, the triple* $(\varphi(s), \varphi(p), \varphi(o))$ *is in* G*, with* φ *the identity on IRIs and literals.*

We distinguish **query evaluation**, whose result is just based on the explicit graph triples, from **query answering** that also accounts for its implicit triples.

Definition 3 (Evaluation and answering). *Let* q *be a (partially instantiated) BGPQ. The* answer set *to* q *on a graph* G *w.r.t. rule set* $\mathcal{R}$ *is:* $q(G, \mathcal{R}) = \{\langle \varphi(\bar{x}) \rangle \mid \varphi$ *homomorphism from body(q) to* $G^{\mathcal{R}}\}$*. If* $\bar{x} = \emptyset$*, i.e.,* q *is a* Boolean *query,* q *is true iff* $q(G, \mathcal{R}) = \{\langle\rangle\}$*. The* evaluation *of* q *on* G*, denoted* $q(G, \emptyset)$*, or* $q(G)$ *in short, is obtained through homomorphisms from body(q) to* G *only.*

These notions and notations naturally extend to *unions* of (partially instantiated) BGPQs, or UBGPQs in short.

Example 3 (Example query). Consider the BGPQ asking *who is writing which*

kind of articles: $q(x, y) \leftarrow (z, \text{:author}, x), (z, \tau, y), (y, \prec_{sc}, \text{:Article})$. Its evaluation on G_{ex} is empty. However, the answer set of q on G_{ex} w.r.t. $\mathcal{R}$ is $q(G_{\text{ex}}, \mathcal{R}) = \{\langle\text{:Alice}, \text{:GOpenArt}\rangle, \langle\text{:Alice}, \text{:OpenArt}\rangle, \langle\text{:Bob}, \text{:GOpenArt}\rangle, \langle\text{:Bob}, \text{:OpenArt}\rangle\}$.

Many RDF data management systems use **saturation-based** query answering, i.e., query evaluation on the previously saturated graph; clearly, from the above definition, $q(G, \mathcal{R}) = q(G^{\mathcal{R}})$ holds. An alternative technique is **reformulation-based** query answering, e.g., [6,18,9], which injects the ontological knowledge into a reformulated query, whose simple evaluation on G yields the complete answer set of the original query. More precisely, given a BGPQ q asked on a graph G, the reformulation of q w.r.t. to $\mathcal{R}$ and the ontology of G, denoted $\mathcal{Q}_{\mathcal{R}}$, is such that $q(G, \mathcal{R}) = \mathcal{Q}_{\mathcal{R}}(G)$. A property of the technique proposed in [9], on which we rely in this paper, is that the reformulated query does not contain schema triples anymore; intuitively, such triples are evaluated on the ontology of the queried graph during the query reformulation. Further, from now on, we assume that (U)BGPQs, in particular those produced through reformulation, are *minimal*: non-minimality incurs redundancy of triples in BGPQs, or of BGPQs within UBGPQs. Well-known query minimization techniques exist for this purpose.

Example 4. The reformulation of the example query q (Example 3) is:

$$\mathcal{Q}_{\mathcal{R}} = \begin{array}{l} q(x, \text{:OpenArt}) \leftarrow (z, \text{:author}, x), (z, \tau, \text{:OpenArt}) \\ \cup\, q(x, \text{:OpenArt}) \leftarrow (z, \text{:firstAuth}, x), (z, \tau, \text{:OpenArt}) \\ \cup\, q(x, \text{:OpenArt}) \leftarrow (z, \text{:author}, x), (z, \tau, \text{:GOpenArt}) \\ \cup\, q(x, \text{:OpenArt}) \leftarrow (z, \text{:firstAuth}, x), (z, \tau, \text{:GOpenArt}) \\ \cup\, q(x, \text{:GOpenArt}) \leftarrow (z, \text{:author}, x), (z, \tau, \text{:GOpenArt}) \\ \cup\, q(x, \text{:GOpenArt}) \leftarrow (z, \text{:firstAuth}, x), (z, \tau, \text{:GOpenArt}) \end{array}$$

It can be checked that $\mathcal{Q}_{\mathcal{R}}(G_{\text{ex}}) = q(G_{\text{ex}}, \mathcal{R})$.

3 BGPQ answering through translation on T and CP

We now detail how (saturation- or reformulation-based) query answering can be performed on the T and the CP storage layouts (Sections 3.1 and 3.2).

3.1 BGPQ answering on the T layout

Let $t(S, P, O)$ be the table storing the triples of a graph G for the T layout.

Saturation-based query answering. The saturation $G^{\mathcal{R}}$ of G is stored in the table t. The algebraic translation of a BGPQ $q(\bar{x}) \leftarrow t_1, \ldots, t_n$ on the T layout is:

$$T(q) = \pi_q(\bowtie_{jcond} (\alpha_T(t_1), \ldots, \alpha_T(t_n)))$$

where α_T, the *query triple translation for the T layout*, translates the i-th query triple $t_i(s_i, p_i, o_i)$ into an algebraic expression of the form $\sigma_{scond}(t)$, where t is the triple table, and *scond* is a (possibly empty) set of selections over the attributes of t; specifically, if s_i (respectively, p_i, o_i) is an IRI or a literal, *scond* contains a predicate of the form $S = s_i$ (and similarly for p_i, o_i); *jcond* is a conjunction of join predicates containing, for every variable appearing in several positions (in one or several triples) in q, an equality between the respective attributes in the $\alpha_T(t_i)$ triple translations; finally π_q is a projection on the attributes from the $\alpha(t_i)$'s corresponding to the answer variables of q, or the values to which such variables are bound in case of a partially instantiated query.

Example 5. The example query translates on the T layout as:

$$\pi_{t_1.O,t_2.O}(\bowtie_{t_1.S=t_2.S,t_2.O=t_3.S}(\sigma_{P=\text{:author}}(t),\sigma_{P=\tau}(t),\sigma_{P=\prec_{sc}\wedge O=\text{:Article}}(t)))$$

In the above, the *selection* $\sigma_{P=\text{:author}}(t)$ restricts the triples from the t table to those having the attribute P equal to :author. Similarly, $\sigma_{P=\tau}(t)$ corresponds to the selection $P = \tau$. On the atom t_3, α_T applies a double selection $\sigma_{P=\prec_{sc}\wedge O=\text{:Article}}(t)$, since t_3 has only one variable in position S. Further, $\bowtie_{jcond}=\bowtie_{t_1.S=t_2.S,t_2.O=t_3.S}$ *joins* the three previous selections, where $t_1.S = t_2.S$ and $t_2.O = t_3.S$ respectively reflect the co-occurrences of variables z and y. The final *projection* $\pi_{t_1.O,t_2.O}$ returns the pairs of values obtained for (x, y).

Reformulation-based query answering. Here, the graph G is stored in t (but not its saturation), and every incoming BGPQ q is reformulated into a (partially instantiated) UBGPQ $\mathcal{Q}_{\mathcal{R}} = \bigcup_{i=1}^{n} q_i$, whose algebraic translation on the T layout is the union of the algebraic translations of its (partially instantiated) BGPQs:

$$T(\mathcal{Q}_{\mathcal{R}}) = \bigcup_{i=1}^{n} T(q_i)$$

Example 6. Consider again the example query q and its reformulation $\mathcal{Q}_{\mathcal{R}}$ shown in Example 4. The algebraic translation $T(\mathcal{Q}_{\mathcal{R}})$ is:

$$\begin{aligned}
&\pi_{t_1.O,\text{:OpenArt}}(\bowtie_{t_1.S=t_2.S}(\sigma_{P=\text{:author}}(t),\sigma_{P=\tau\wedge O=\text{:OpenArt}}(t))\\
&\cup\,\pi_{t_1.O,\text{:OpenArt}}(\bowtie_{t_1.S=t_2.S}(\sigma_{P=\text{:firstAuth}}(t),\sigma_{P=\tau\wedge O=\text{:OpenArt}}(t))\\
&\cup\,\pi_{t_1.O,\text{:OpenArt}}(\bowtie_{t_1.S=t_2.S}(\sigma_{P=\text{:author}}(t),\sigma_{P=\tau\wedge O=\text{:GOpenArt}}(t))\\
&\cup\,\pi_{t_1.O,\text{:OpenArt}}(\bowtie_{t_1.S=t_2.S}(\sigma_{P=\text{:firstAuth}}(t),\sigma_{P=\tau\wedge O=\text{:GOpenArt}}(t))\\
&\cup\,\pi_{t_1.O,\text{:GOpenArt}}(\bowtie_{t_1.S=t_2.S}(\sigma_{P=\text{:author}}(t)),\sigma_{P=\tau\wedge O=\text{:GOpenArt}}(t))\\
&\cup\,\pi_{t_1.O,\text{:GOpenArt}}(\bowtie_{t_1.S=t_2.S}(\sigma_{P=\text{:firstAuth}}(t),\sigma_{P=\tau\wedge O=\text{:GOpenArt}}(t))
\end{aligned}$$

3.2 BGPQ answering on the CP layout

With the CP layout, an RDF graph is stored as a set of tables corresponding to classes and properties: for each class c, there is a table $t_c(S)$ that stores all subjects s of triples (s, τ, c), and for each data or schema property $p \neq \tau$, there is a table $t_p(S, O)$ that stores all subject-object pairs (s, o) for triples (s, p, o). We call any such t_c a *class table*, and t_p a *property table*. The CP layout speeds up data access for queries which *specify the class in every triple whose property is* τ and *specify the property in every triple*. However, as noted in [21], it may render the evaluation of general queries, with variables in class or property position, inefficient, as the triples they refer to may be in *any* t_c or t_p tables, respectively.

Saturation-based query answering. Assume that $G^{\mathcal{R}}$ is stored using the CP layout. To obtain the answers to a BGPQ $q(\bar{x}) \leftarrow t_1, \ldots, t_n$, a first simple *naïve translation*, which can be traced back to [4,21], is:

$$CP(q) = \pi_q(\bowtie_{jcond}(\alpha_{CP}(t_1), \cdots, \alpha_{CP}(t_n)))$$

where π_q and $jcond$ are defined as for the T layout, and α_{CP}, the *query triple translation for the CP layout*, is:

$$\alpha_{CP}(t) = \begin{cases}
\pi_{S,\tau,c}(\sigma_{scond}(t_c)) \text{ if } t = (s, \tau, c) \text{ with } c \notin \mathscr{V} & (1)\\
\pi_{S,p,O}(\sigma_{scond}(t_p)) \text{ if } t = (s, p, o) \text{ with } p \notin \mathscr{V} \cup \{\tau\} & (2)\\
\bigcup_{c\in\mathcal{C}} \alpha_{CP}((s, \tau, c)) \text{ if } t = (s, \tau, x) \text{ with } x \in \mathscr{V} & (3)\\
\alpha_{CP}((s, \tau, o)) \cup \bigcup_{p\in\mathcal{P}} \alpha_{CP}((s, p, o)) \text{ if } t = (s, x, o) \text{ with } x \in \mathscr{V} & (4)
\end{cases}$$

where $\mathcal{C}$ and $\mathcal{P}$ are, respectively, the set of classes and of properties other than τ in the queried graph, and *scond* is a (possibly empty) *conjunction* of selections, just as we defined it for α_T, but over the class and property tables instead of the triple table t.

Example 7. The naive translation of the example query q on the CP layout is:
$\pi_{t_1.O,t_2.O}(\bowtie_{t_1.S=t_2.S,t_2.O=t_3.S} (\pi_{S,:\text{author},O}(t_{:\text{author}}),$
$\pi_{s,\tau,:\text{GOpenArt}}(t_{:\text{GOpenArt}}) \cup \pi_{s,\tau,:\text{OpenArt}}(t_{:\text{OpenArt}}) \cup \pi_{s,\tau,:\text{Article}}(t_{:\text{Article}})$
$\cup\, \pi_{s,\tau,:\text{Prof}}(t_{:\text{Prof}}) \cup \pi_{s,\tau,:\text{Person}}(t_{:\text{Person}}) \cup\ \pi_{S,\prec_{sc},O}(\sigma_{O=:\text{Article}}(t_{\prec_{sc}})))$

Note that in cases (3) and (4) above, α_{CP} introduces *unions under joins*, as illustrated by the previous example. This leads to **suboptimal evaluation performance** in many data management engines, which *may optimize and execute efficiently a join over several data collections, but do not attempt to reorder (commute) joins with unions*. For instance, the query $(x, :\text{a}, :\text{a}_1)$, (x, y, z), (z, τ, u), $(z, :\text{b}, :\text{b}_1)$ translates into a plan that joins (among others) the union of all the tables (for (x, y, z)) with the union of all class tables (for (z, τ, u)). Most systems execute this “literally”, i.e., they build and materialize these two very large unions, which is very costly, before joining them with the first and last triple[6]. To avoid such unions under joins, we rely on the notion of instantiation, which has been used in various query answering techniques e.g., [15,18]:
Query instantiation. The instantiation of a BGPQ q consists in instantiating the variables in q that must be bound to classes and properties of the queried graph, in all possible ways, which yields a (partially instantiated) UBGPQ. Given a BGPQ q and a graph G, we denote by $q^{p,G}$ (resp. $q^{c,G}$) its *property instantiation* (resp. *class instantiation*), which is the UBGPQ obtained by instantiating all its variables in property position (resp. in class position), by all combinations of properties (resp. classes) of G.

Example 8. The class instantiation $q^{c,G_{ex}}$ of the example query q, where the only instantiated variable is y, is:

$q(x, :\text{GOpenArt}) \leftarrow (z, :\text{author}, x), (z, \tau, :\text{GOpenArt}), (:\text{GOpenArt}, \prec_{sc}, :\text{Article})$
$\cup\, q(x, :\text{OpenArt}) \leftarrow (z, :\text{author}, x), (z, \tau, :\text{OpenArt}), (:\text{OpenArt}, \prec_{sc}, :\text{Article})$
$\cup\, q(x, :\text{Article}) \leftarrow (z, :\text{author}, x), (z, \tau, :\text{Article}), (:\text{Article}, \prec_{sc}, :\text{Article})$
$\cup\, q(x, :\text{Prof}) \leftarrow (z, :\text{author}, x), (z, \tau, :\text{Prof}), (:\text{Prof}, \prec_{sc}, :\text{Article})$
$\cup\, q(x, :\text{Person}) \leftarrow (z, :\text{author}, x), (z, \tau, :\text{Person}), (:\text{Person}, \prec_{sc}, :\text{Article})$

Class and property instantiations extend from BGPQs to UBGPQs in the natural way. Given a UBGPQ of the form $Q = q^1 \cup q^2 \cdots \cup q^n$, we set:

$$Q^{p,G} = q_1^{p,G} \cup q_2^{p,G} \cdots \cup q_n^{p,G} \text{ and } Q^{c,G} = q_1^{c,G} \cup q_2^{c,G} \cdots \cup q_n^{c,G}$$

Then, the *instantiation* of Q w.r.t. a graph G is the following:

$$Q^G = (Q^{c,G})^{p,G} \cup (Q^{p,G})^{c,G}$$

[6]We checked this on systems that disclose their query execution strategy; experiments with others who do not, confirm the same hypothesis (see Section 5).

We need both terms of the above union, *exactly* in the case when some variable of Q appears both in a property and in a class position. These cases are rare and easy to detect, thus in practice we only use one of the unions as soon as we detect both are not needed. Crucially, (U)BGPQ instantiation is *correct* for saturation- and reformulation-based query answering. Intuitively, this is because instantiation enumerates all possible combinations of classes and properties that query reformulation or evaluation may find in G.

We can now define the *instantiation-based query translation.* A BGPQ q is first instantiated into $q^G = \bigcup_{i=1}^n q^i$, then translated on the CP layout as:

$$CP(q^G) = \bigcup_{i=1}^{n} CP(q^i)$$

Importantly, because q^G does not contain any variable in class or property position, every naïve translation $CP(q^i)$ within $CP(q^G)$ avoids both (3) or (4) in the α_{CP} triple transformation function. Hence, the translation avoids the introduction of unions under joins, with their potential bad impact on performance.

Example 9. Consider the query q and its instantiation $q^{G_{ex}} = q^{c,G_{ex}}$ in Example 8. The instantiation-based translation of q corresponds to the naïve translation of $q^{G_{ex}}$:

$$\bigcup_{\mathtt{u}\in\{\text{:GOpenArt,:OpenArt,:Article,:Prof,:Person}\}} \pi_{t_1.O,\mathtt{u}}(\bowtie_{t_1.S=t_2.S,t_2.O=t_3.S} (\pi_{S,\text{:author},O}(t_{\text{:author}}), \pi_{S,\tau,\mathtt{u}}(t_{\mathtt{u}}), \pi_{S,\prec_{sc},O}(\sigma_{S=\mathtt{u},O=\text{:Article}}(t_{\prec_{sc}}))$$

Reformulation-based query answering. The graph G is again stored using the CP layout (but not saturated). In this case, answering a BGPQ q starts by computing its reformulation $\mathcal{Q}_\mathcal{R}$ w.r.t. the ontology of G. Then, we obtain the answers $q(G,\mathcal{R})$ either through $CP(\mathcal{Q}_\mathcal{R})$, the naïve translation of $\mathcal{Q}_\mathcal{R}$, or through $CP(\mathcal{Q}_\mathcal{R}^G)$, the instantiation-based translation of $\mathcal{Q}_\mathcal{R}$, i.e., the naïve translation of its instantiation $\mathcal{Q}_\mathcal{R}^G$; as in the previous section, the algebraic translation of a UBGPQ is defined as the union of the algebraic translations of its BGPQs. Instantiating $\mathcal{Q}_\mathcal{R}$ generally increases its size, but, by removing variables in class and property positions, it avoids the unions under joins introduced in cases (3) and (4) by the α_{CP} triple translation function.

Example 10. Consider the query q and its reformulation $\mathcal{Q}_\mathcal{R}$ from Example 4. Here, since no variable of $\mathcal{Q}_\mathcal{R}$ occurs in class or property position, $CP(\mathcal{Q}_\mathcal{R})$ and $CP(\mathcal{Q}_\mathcal{R}^G)$ lead to the same algebraic expression:

$$\begin{aligned}
&\pi_{t_1.O,\text{:OpenArt}}(\bowtie_{t_1.S=t_2.S} (\pi_{S,\text{:author},O}(t_{\text{:author}}), \pi_{S,\tau,\text{:OpenArt}}(t_{\text{:OpenArt}}))\\
&\cup \pi_{t_1.O,\text{:OpenArt}}(\bowtie_{t_1.S=t_2.S} (\pi_{S,\text{:firstAuth},O}(t_{\text{:firstAuth}}), \pi_{S,\tau,\text{:OpenArt}}(t_{\text{:OpenArt}}))\\
&\cup \pi_{t_1.O,\text{:OpenArt}}(\bowtie_{t_1.S=t_2.S} (\pi_{S,\text{:author},O}(t_{\text{:author}}), \pi_{S,\tau,\text{:GOpenArt}}(t_{\text{:GOpenArt}}))\\
&\cup \pi_{t_1.O,\text{:OpenArt}}(\bowtie_{t_1.S=t_2.S} (\pi_{S,\text{:firstAuth},O}(t_{\text{:firstAuth}}), \pi_{S,\tau,\text{:GOpenArt}}(t_{\text{:GOpenArt}}))\\
&\cup \pi_{t_1.O,\text{:GOpenArt}}(\bowtie_{t_1.S=t_2.S} (\pi_{S,\text{:author},O}(t_{\text{:author}}), \pi_{S,\tau,\text{:GOpenArt}}(t_{\text{:GOpenArt}}))\\
&\cup \pi_{t_1.O,\text{:GOpenArt}}(\bowtie_{t_1.S=t_2.S} (\pi_{S,\text{:firstAuth},O}(t_{\text{:firstAuth}}), \pi_{S,\tau,\text{:GOpenArt}}(t_{\text{:GOpenArt}}))
\end{aligned}$$

4 Taming general BGP answering performance

Below, we present our technical contributions: the TCP layout and its algebraic translation (Section 4.1), and summary-based pruning (Section 4.2).

4.1 BGPQ answering based on the TCP layout

The TCP layout combines T and CP with the aim of getting the best of both, while avoiding the performance problems they respectively entail (Sections 3.1 and 3.2). Here, an RDF graph is stored *both* in the triple table t of the T layout *and* in the t_c class and t_p property tables of the CP layout. The rationale behind this is that CP is efficient to access triples when *the data structures holding the triples we need to access are immediately clear from the query, and small*; this is the case with query triples of the form (s, τ, c) or (s, p, o) for a known class c or property p. However, with query triples of the form (s, τ, x) and (s, x, o), the CP translation introduces unions, typically executed before joins, degrading performance. Interestingly, *direct access to a potentially large share of a graph's triples is exactly what the T layout supports well* - thus our idea to combine them. As we show in the next section, this allows to significantly improve performance, at expense of some extra storage space, typically inexpensive since it is on disk.

Saturation-based query answering. Let us assume that the saturation $G^{\mathcal{R}}$ of a graph G is stored in the TCP layout. The answers to a BGPQ $q \leftarrow t_1, \ldots, t_n$ are obtained through its algebraic transformation for the TCP layout:

$$TCP(q) = \pi_q(\bowtie_{jcond} (\alpha_{TCP}(t_1), \cdots, \alpha_{TCP}(t_n)))$$

where π_q and $jcond$ are defined as for the T and CP layouts, and α_{TCP}, the *query triple translation for the TCP layout*, is:

$$\alpha_{TCP}(t) = \begin{cases} \alpha_{CP}(t) \text{ if } t = (s, \tau, c) \text{ or } t = (s, p, o) \text{ with } c \notin \mathscr{V}, p \notin \mathscr{V} \cup \{\tau\} \\ \alpha_T(t) \text{ otherwise, i.e., if } t = (s, \tau, x) \text{ or } t = (s, x, o) \text{ with } x \in \mathscr{V} \end{cases}$$

Importantly, α_{TCP} translates the triples that penalize the CP layout into t atoms, and never into a union: hence, α_{TCP} avoids the cases (3) and (4) of α_{CP}.

Example 11. The translation of the example query for the TCP layout combines the T layout for the second triple and the CP layout for the others:

$\pi_{t_1.O,t_2.O}(\bowtie_{t_1.S=t_2.S,t_2.O=t_3.S} (\pi_{S,:\text{author},O}(t_{:\text{author}}), \sigma_{P=\tau}(t), \pi_{S,\prec_{sc},O}(\sigma_{O=:\text{Article}}(t_{\prec_{sc}}))))$

Reformulation-based query answering. Again, the answers to a query q are computed by evaluating the algebraic translation of its reformulation $\mathcal{Q}_{\mathcal{R}} = \bigcup_{i=1}^{n} q^i$, but now for the TCP layout:

$$TCP(\mathcal{Q}_{\mathcal{R}}) = \bigcup_{i=1}^{n} TCP(q^i)$$

Example 12. Consider the query $q(x, y, z) \rightarrow (x, \tau, z), (x, :\text{firstAuth}, y)$ asking for all resources with their type and first author. Its reformulation w.r.t. G_{ex}'s ontology is shown below (the last four union terms are omitted for space reasons):

$$\mathcal{Q}_{\mathcal{R}} = \begin{array}{l} q(x, y, z) \leftarrow (x, \tau, z), (x, :\text{firstAuth}, y) \\ \cup\, q(x, y, :\text{Article}) \leftarrow (x, \tau, :\text{OpenArt}), (x, :\text{firstAuth}, y) \\ \cup\, q(x, y, :\text{Article}) \leftarrow (x, \tau, :\text{GOpenArt}), (x, :\text{firstAuth}, y) \\ \cup\, q(x, y, :\text{OpenArt}) \leftarrow (x, \tau, :\text{GOpenArt}), (x, :\text{firstAuth}, y) \\ \cup\, q(x, y, :\text{Person}) \leftarrow (x, \tau, :\text{Prof}), (x, :\text{firstAuth}, y) \ldots \end{array}$$

Its algebraic translation on the TCP layout (similarly abridged) is:

$\pi_{t_1.S,t_2.O,t_1.O}(\bowtie_{t_1.S=t_2.S} (\sigma_{P=\tau}(t), \pi_{S,:\text{firstAuth},O}(t_{:\text{firstAuth}})))$
$\cup\, \pi_{t_1.S,t_2.O,:\text{Article}}(\bowtie_{t_1.S=t_2.S} (\pi_{S,\tau,:\text{OpenArt}}(t_{:\text{OpenArt}}), \pi_{S,:\text{firstAuth},O}(t_{:\text{firstAuth}})))$
$\cup\, \pi_{t_1.S,t_2.O,:\text{Article}}(\bowtie_{t_1.S=t_2.S} (\pi_{S,\tau,:\text{GOpenArt}}(t_{:\text{GOpenArt}}), \pi_{S,:\text{firstAuth},O}(t_{:\text{firstAuth}})))$
$\cup\, \pi_{t_1.S,t_2.O,:\text{OpenArt}}(\bowtie_{t_1.S=t_2.S} (\pi_{S,\tau,:\text{GOpenArt}}(t_{:\text{GOpenArt}}), \pi_{S,:\text{firstAuth},O}(t_{:\text{firstAuth}})))$
$\cup\, \pi_{t_1.S,t_2.O,:\text{Person}}(\bowtie_{t_1.S=t_2.S} (\pi_{S,\tau,:\text{Prof}}(t_{:\text{Prof}}), \pi_{S,:\text{firstAuth},O}(t_{:\text{firstAuth}}))) \ldots$

Above, the first union term refers to the triple table t, while the others do not.

4.2 Summary-based query pruning

We now introduce an optimization technique, which can be applied on *any* storage layout to reduce (U)BGPQ answering time. It allows detecting some BGPQs with an empty answer set on a graph, without evaluating them, by using a (typically much smaller) structural summary of this graph.
Given an RDF graph G and an *equivalence relation* $\equiv$ among the nodes in G, i.e., the subjects and objects of triples, an *RDF quotient summary* [12] is an RDF graph $G_{/\equiv}$ built as follows. A node is created in $G_{/\equiv}$ for each equivalence class among G's nodes; further, for any triple $(n_1, p, n_2) \in G$, the triple (m_1, p, m_2) appears in $G_{/\equiv}$, where m_1 and m_2 represent the equivalence class of n_1 and n_2 respectively. If there are large equivalence classes in G, summarization is a form of compression. Several types of RDF quotient summaries have been proposed [12]; in our experiments, we used the RDFQuotient summary construction tool [14], due to its online availability and low summary construction cost (linear in the number of triples of G). An RDFQuotient summary represents each class and property node *by itself*, and consider they are not equivalent to any other G node; thus, G and any quotient summary $G_{/\equiv}$ have the *same* schema triples.
Crucially, it holds that if $q(G_{/\equiv}) = \emptyset$ then $q(G) = \emptyset$, for q a *structural* (U)BGPQ, i.e., in which the subjects and objects of query triples are either class and property IRIs, or variables. Intuitively, this result holds because structural queries only allow selecting subject, property and object values that are preserved through summarization (class and property IRIs). Note however that the opposite does not hold in general, i.e., $q(G_{/\equiv})$ may have results while $q(G)$ does not. We exploit this result by defining the *structural version* of a BGPQ q, denoted q^{str}, which is obtained by replacing in q the literals and the IRIs that are not class or property IRIs, by fresh variables. For example, the structural version of the query $q(x) \leftarrow (x, \tau, :\text{OpenArt}), (x, :\text{firstAuth}, :\text{Alice})$ is: $q^{\text{str}}(x) \leftarrow (x, \tau, :\text{OpenArt}), (x, :\text{firstAuth}, y)$, with :Alice replaced by y. Hence, when a summary $G_{/\equiv}$ is available, we can use it to *prune* a UBGPQ $\mathcal{Q} = \bigcup_i q_i$ by removing from the union all the q_i terms for which $q_i^{\text{str}}(G_{/\equiv}) = \emptyset$. As explained above, this may fail to prune some q_i with no results on G, but it preserves query results:

$$\mathcal{Q}(G) = \mathcal{Q}^{\text{pruned}}(G)$$

where $\mathcal{Q}^{\text{pruned}}$ is the result of pruning $\mathcal{Q}$. As our experiments will show, this generally leads to a significant reduction of query answering time.

5 Experimental evaluation

We now describe experiments comparing the query answering methods presented in the previous sections, on the T, CP and TCP layouts.

5.1 Experimental settings

We implemented the T, CP and TCP layouts in **OntoSQL** (`https://ontosql.inria.fr`), a Java platform providing efficient RDF storage and saturation- and reformulation-based query answering on top of an RDBMS [9,10,16] (Postgres 9.6 in these experiments). OntoSQL encodes IRIs and literals as integers, and a dictionary table allows going from one to the other; each table (t, t_p or t_c) is indexed on all the subsets of its attributes. To use OntoSQL, we *express our algebraic translations in SQL*. We checked that in Postgres query plans, the relative positions of unions and joins in the query plans chosen by the RDBMS are those from our translations; [10,11] showed that this holds for two other major RDBMSs. However, *the RDBMS takes all optimization decisions*, based on its cost model and statistics. To put this into perspective also with respect to native RDF engines, we ran the same experiments also on **Virtuoso** Open Source Edition 7.2, to whom we provided *SPARQL queries*, which correspond exactly to our algebraic translation on the T layout. Virtuoso also controls its optimization decisions, and has full control over its store.

For summary-based pruning, we used the **RDFQuotient** (`https://rdfquotient.inria.fr`) tool to build the "typed strong" summary [14] of a graph G; this summary is denoted $G_{/\mathtt{TS}}$. The summary groups typed nodes according to their types, and untyped nodes by exploiting the similarity of their incoming/outgoing properties (see [14] for details). In general, any quotient summary could be used; a large (more detailed) summary makes pruning more accurate, but also slower since it needs to query the summary.

Hardware We used a server with 2,7 GHz Intel Core i7 processors and 160 GB of RAM, running CentOs Linux 7.5.

Graph	$\|G\|$	$\|G_{/\mathtt{TS}}\|$	$\|G^{\mathcal{R}}\|$	$\|(G^{\mathcal{R}})_{/\mathtt{TS}}\|$	$\|G\|_{\mathrm{T}}$	$\|G\|_{\mathrm{CP}}$	$\|G\|_{\mathrm{TCP}}$	$\|G^{\mathcal{R}}\|_{\mathrm{T}}$	$\|G^{\mathcal{R}}\|_{\mathrm{CP}}$	$\|G^{\mathcal{R}}\|_{\mathrm{TCP}}$
LUBM	100M	340	131M	439	28.95	11.95	37.89	32.52	13.52	39.31
DBLP	88M	290	147M	708	26.07	16.35	35.89	38.39	22.91	52.72

Table 3. Graph and summary sizes (number of triples), OntoSQL database sizes (in GB), including all indexes, for the T, CP and TCP layouts.

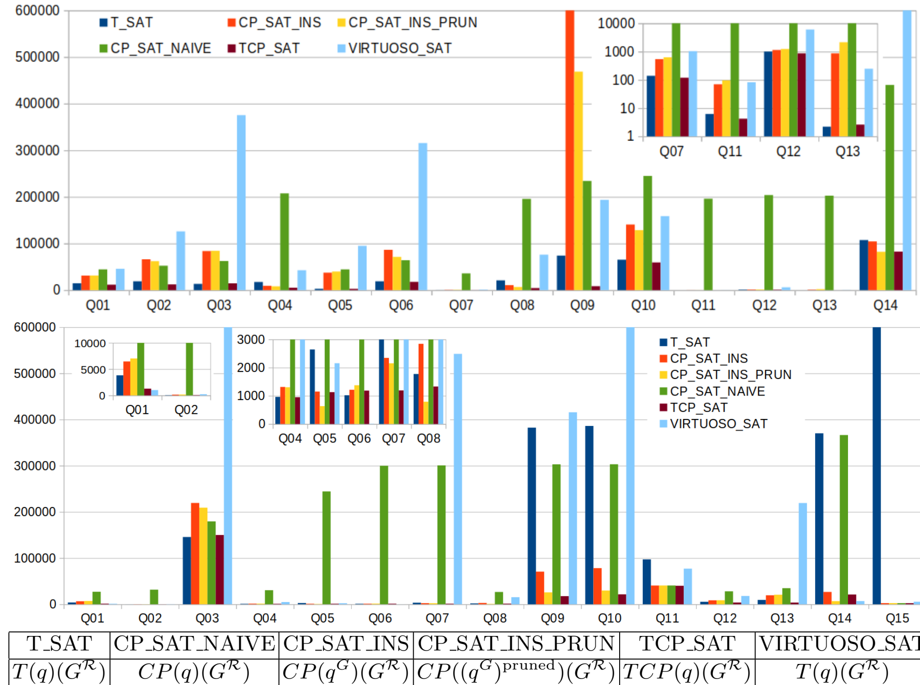
RDF graphs We used two benchmark graphs: a **LUBM** [17] graph of 100M triples, as well as a graph of **DBLP** bibliographic data endowed with an ontology of 14 classes and 44 schema triples. Table 3 shows, for these graphs and their saturation, the graph and summary sizes, and the sizes of the OntoSQL databases storing them in the T, CP and TCP layout. As expected, TCP takes most space, approx. 90% of the sum of the T and CP database sizes. However, this is stable storage (e.g., disk) space; the selective data access enabled by the class and property tables, and by indexes, as well as cost-based optimization, ensure that the data loaded in memory to process a query is much smaller.

Queries We used two **diverse** sets of queries, having from 1 to 11 triples (4 on average) on LUBM, and from 2 to 9 triples (5.9 on average) on DBLP. Each query has 1 or 2 triple(s) of the form (s, τ, x) and/or (s, x, o), except Q11 and Q15 on DBLP which do not contain any. Table 4 shows their number of answers, and the number of BGPQs in: their instantiation (q^G), reformulation ($\mathcal{Q}_{\mathcal{R}}$), and instantiation of their reformulation ($\mathcal{Q}_{\mathcal{R}}^G$), before and after summary-based

Query	Q01	Q02	Q03	Q04	Q05	Q06	Q07	Q08	Q09	Q10	Q11	Q12	Q13	Q14
$\|q(G,\mathcal{R})\|$	2.78M	0.59M	2.15M	1.72M	0.47M	2.77M	25K	3696	2003	17.35M	187	518K	857	9.85M
$\|q^G\|$	45	45	45	82	2025	45	45	82	3690	82	82	37	1369	82
$\|(q^G)^{\text{pruned}}\|$	45	45	45	67	1806	44	44	3	88	68	59	23	43	51
$\|\mathcal{Q}_\mathcal{R}\|$	318	106	146	68	216	80	318	9	36	88	9	27	39	1152
$\|(\mathcal{Q}_\mathcal{R})^{\text{pruned}}\|$	120	56	59	48	108	34	173	7	2	78	7	21	21	889
$\|\mathcal{Q}_\mathcal{R}^G\|$	447	149	226	68	216	121	447	9	36	169	9	63	1797	1188
$\|(\mathcal{Q}_\mathcal{R}^G)^{\text{pruned}}\|$	206	99	135	48	108	73	299	7	2	144	7	42	50	892

Query	Q01	Q02	Q03	Q04	Q05	Q06	Q07	Q08	Q09	Q10	Q11	Q12	Q13	Q14	Q15
$\|q(G,\mathcal{R})\|$	72K	24K	96.3M	361K	4K	10	138	42K	1.52M	3.09M	957K	414K	409K	16.3K	460K
$\|q^G\|$	18	18	18	18	50	900	900	18	900	900	1	18	324	50	1
$\|(q^G)^{\text{pruned}}\|$	18	18	18	18	3	136	136	17	85	85	1	18	324	3	1
$\|\mathcal{Q}_\mathcal{R}\|$	117	297	265	117	873	4257	3163	36	1500	3652	243	39	381	129	243
$\|(\mathcal{Q}_\mathcal{R})^{\text{pruned}}\|$	56	205	151	56	616	3089	2166	32	1184	2620	36	24	174	114	3
$\|\mathcal{Q}_\mathcal{R}^G\|$	252	440	408	252	1529	5927	3345	72	1	324	3	84	2088	270	243
$\|(\mathcal{Q}_\mathcal{R}^G)^{\text{pruned}}\|$	161	331	277	161	1224	3275	2189	37	1	64	3	69	1497	117	3

Table 4. Statistics of our queries on LUBM (top) and DBLP (bottom); M stands for millions and K for thousands.



T_SAT	CP_SAT_NAIVE	CP_SAT_INS	CP_SAT_INS_PRUN	TCP_SAT	VIRTUOSO_SAT
$T(q)(G^\mathcal{R})$	$CP(q)(G^\mathcal{R})$	$CP(q^G)(G^\mathcal{R})$	$CP((q^G)^{\text{pruned}})(G^\mathcal{R})$	$TCP(q)(G^\mathcal{R})$	$T(q)(G^\mathcal{R})$

Fig. 2. Query answering times (milliseconds) on LUBM (top) and DBLP (bottom), through saturation.

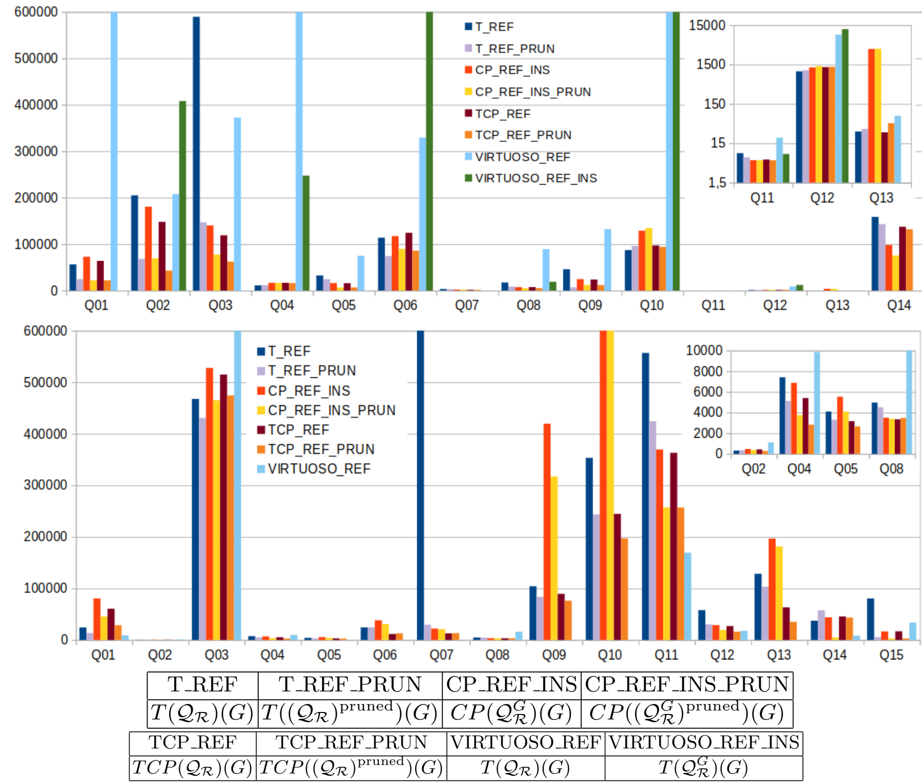
pruning. The impact of pruning ranges from none (in particular for q^G, on LUBM Q01 to Q03 and on 8 DBLP queries) to very significant (97.8% of the q^G BGPs are pruned on LUBM Q09). Details on our experiments, and code which can be used to reproduce them, are online[5].

5.2 Experiment results: query answering times

For each query, we report the average of the last five ("hot") runs out of six.

Through saturation Figure 2 shows the query answering times through saturation, for LUBM (top) and DBLP (bottom), with a timeout of 10 minutes;

in all our graphs, executions that reached the timeout have been interrupted. Below the graphs, we show the label used in the plot for each query answering strategy, e.g., T_SAT stands for $T(q)(G^{\mathcal{R}})$. For readability, some very fast queries are repeated in a "zoom" plot (the LUBM one has a logarithmic y axis).
On **LUBM** (top), we notice some very high running times for VIRTUOSO_SAT, e.g., on Q03, Q06, and a time-out on Q14. Among the SQL-based strategies, the naïve translation on CP (green bars) is slowest in 10 out of 14 queries, with large performance penalties for Q04, Q07, Q08, Q11-13. Instantiation (CP_SAT_INS, red bars) is generally faster than naïve CP. It strongly speeds up Q04, Q07, Q08, Q10-Q14; it brings a modest improvement to Q01 and Q05, but also a modest overhead for Q02, Q03, Q06. However, on the complex Q09, which has the largest q^G size, namely 3690, instantiating each of these more than doubles the answering time w.r.t. naïve CP translation (and ran until the timeout); pruning (yellow bars) brings it back below the timeout. T_SAT is generally faster than all executions on the CP layout, because all queries contain triples of the form (s, τ, x) and/or (s, x, o), which, as explained in Section 3.2, challenge CP execution. *TCP_SAT avoids the (sometimes drastic) performance problems of all CP variants, and is the fastest (by up to several orders of magnitude) on all queries but Q14*, where the CP variant with pruning is a bit faster. Virtuoso is also always slower than TCP (by up to 95×, almost two orders of magnitude).
On **DBLP** (bottom), poor performance is exhibited by Virtuoso (Q03, Q07, Q09, Q10), and on even more queries by the naïve CP strategy (green bars, Q05-Q07, Q09-Q10, Q14). T_SAT performs very badly on Q09, Q10, Q14 and Q15. These are rather large (6 to 9-triples) queries; an analysis of their plans shows significant errors in the RDBMS' estimation of join cardinality. As is well-known, join cardinality estimation errors multiply along subsequent joins; when all joins carry over a single, very large table, the negative impact of such cumulated errors can be quite strong. Historically, this observation actually motivated the introduction of the CP layout, on which join estimation errors still multiply, but usually much smaller tables are involved. Indeed, as expected, for the queries Q11 and Q15, exactly those where no triple has a variable in class and property position, naïve CP largely outperforms T_SAT (by very far for Q15). Again, we observe the robust behavior of TCP_SAT. We conclude that through saturation, *T and CP execution each underperform on some queries, but TCP avoids all these pitfalls and is consistently very efficient.*
Through reformulation Figure 3 shows reformulation-based query answering times (note the logarithmic y axis in the zoom), again with the correspondences between the bar labels and the strategy names previously used in the paper.
On **LUBM**, among the evaluation strategies *without pruning*, TCP_REF is generally the fastest (or very close to it), with the exception of Q14, where CP_REF_INS is 1.4× faster. This query with the most results (9.85M) and a large reformulation (Table 4) has two atoms of the form $(x, p, z), (y, p, z)$. On CP, this leads to a large number of self-joins of the form $t_p \bowtie_o t_p$, executed very fast because loading t_p in memory once ensures the join runs completely in-memory. While the rather unusual Q14 shows a case when CP may still out-



T_REF	T_REF_PRUN	CP_REF_INS	CP_REF_INS_PRUN
$T(\mathcal{Q}_{\mathcal{R}})(G)$	$T((\mathcal{Q}_{\mathcal{R}})^{\text{pruned}})(G)$	$CP(\mathcal{Q}^{G}_{\mathcal{R}})(G)$	$CP((\mathcal{Q}^{G}_{\mathcal{R}})^{\text{pruned}})(G)$

TCP_REF	TCP_REF_PRUN	VIRTUOSO_REF	VIRTUOSO_REF_INS
$TCP(\mathcal{Q}_{\mathcal{R}})(G)$	$TCP((\mathcal{Q}_{\mathcal{R}})^{\text{pruned}})(G)$	$T(\mathcal{Q}_{\mathcal{R}})(G)$	$T(\mathcal{Q}^{G}_{\mathcal{R}})(G)$

Fig. 3. LUBM (top) and DBLP (bottom) query answering times (ms) through reformulation.

perform TCP, the difference is not dramatic. *On the three layouts, pruning generally helps*: it saves, e.g., more than half of the CP answering time for Q01, Q02. In the zoomed view (shortest-running queries), pruning brings an overhead (takes more time that the query evaluation time it saves) of a few milliseconds. Among the strategies *with* pruning, TCP_REF_PRUN is the fastest, except for Q14 discussed above. As Virtuoso did not support reasoning with our rule set $\mathcal{R}$ (details online[5]), we gave it reformulations expressed in SPARQL; for Q07 and Q14, they failed to run, with the error *"union nesting is too deep"*. The impact of instantiation for Virtuoso is unclear; it helped for Q04, Q08 but not for Q02, Q06 etc. All missing Virtuoso bars in Figure 3 are *execution failures*, mostly due to large unions.

On **DBLP**, VIRTUOSO_REF failed for Q06, Q07, Q09, Q10, Q13; VIRTUOSO_REF_INS was consistently worse, and we omitted it from the plot. The rest of the analysis is similar to the one above, except that T_REF performs very badly in a few cases (Q07, Q11). Overall, in Figure 3, *TCP query answering with pruning is the fastest, or very close to it, on all queries*, while all other strategies's weaknesses are exposed by one or more queries.

5.3 Experiment conclusion

We studied the performance of query answering in RDF databases through saturation and reformulation, on challenging queries that remain poorly supported:

those with variables in class or property position. We have exhibited queries that lead to *poor to catastrophic performance* of query answering *on the T layout (mainly due to many self-joins on a large table)* and/or *on the CP layout (mainly due to large unions, brought by variables in class and property positions, and/or by reformulation)*. Query answering on the TCP layout is extremely *robust*; it avoids all these pitfalls by taking the best of both T and CP, at the expense of more storage space. As disks are getting ever cheaper[7], TCP appears to be a robust, practical layout, compatible with well-established large-scale RDF storage and query engines. For the challenging queries we study, summary-based pruning helps improve performance, in particular for the TCP layout.

6 Related work and conclusion

Our work studies the impact of RDF graph storage on query answering in the presence of RDFS ontologies, both through graph saturation (SatQA) and query reformulation (RefQA). Prior works such as [4,7,19,20,21] only considered SatQA. While [4] advocated the CP layout, [21] nuanced the analysis: in a row store, they show that proper indexing (such as we used here) can significantly improve performance using T, while many distinct properties may hurt CP performance. It is not fully clear if [4,21] used the naïve or instantiation-based CP translation; as our experiments show, TCP outperforms both, in particular with summary pruning. The optimized T layout of [24], indexed on all (s, p, o) permutations, has been used for RefQA in [15,16,10]; in our experiments, TCP avoids all its bad-performance scenarios. Storage was optimized based on a known workload by creating materialized views in [15]. Query reformulation has also been used with the CP layout in [9,11]. Both [10] for T and [11] for CP explored how to express a reformulated query as a join of several subqueries, so as to minimize the evaluation cost through the RDBMS. Applying this technique to the TCP layout could presumably also improve its performance. A simplified version of TCP is briefly mentioned in [13], which studies generic SPARQL-to-SQL translation functions, as an example of possible relational layout. However, [13] does not consider the performance impact of this layout; nor do they consider RDFS entailment. Optimized storage layouts [7,20,25] or indexes [22,5] have been investigated to limit joins by storing e.g., the values of several properties for a given subject together. They allow translating several BGPQ triples into a single table (or index) access; they could also be applied on the TCP layout to further improve it. Finally, in [23], the storage layout based on binary tables is adapted to the graph topology, in order to speed up query evaluation.

TCP is a robust layout, which does not require query workload knowledge, and allows to significantly reduce BGPQ answering times. On queries well supported by the T, respectively, CP layout, TCP matches that performance; but most importantly, it *avoids all the performance (or plain unfeasibility) issues they encounter, in saturation- or reformulation-based query answering*. Summary-based pruning also importantly improves performance. We believe the TCP layout, and pruning, can be adopted with little effort, and can strongly consolidate and improve query answering performance in many RDF databases.

[7] E.g., `https://www.backblaze.com/blog/hard-drive-cost-per-gigabyte/`

References

1. RDF 1.1 Concepts, https://www.w3.org/TR/rdf11-concepts/
2. RDF 1.1 Semantics, https://www.w3.org/TR/rdf11-mt/#rdfs-entailment
3. SPARQL 1.1 Query Language, https://www.w3.org/TR/sparql11-query/
4. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable Semantic Web Data Management Using Vertical Partitioning. PVLDB (2007)
5. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix "bit" loaded: a scalable lightweight join query processor for RDF data. In: WWW (2010)
6. Bischof, S., Krötzsch, M., Polleres, A., Rudolph, S.: Schema-agnostic query rewriting in SPARQL 1.1. In: ISWC (2014)
7. Bornea, M.A., Dolby, J., Kementsietsidis, A., Srinivas, K., Dantressangle, P., Udrea, O., Bhattacharjee, B.: Building an efficient RDF store over a relational database. In: SIGMOD (2013)
8. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A generic architecture for storing and querying RDF and RDF schema. In: ISWC (2002)
9. Buron, M., Goasdoué, F., Manolescu, I., Mugnier, M.: Reformulation-based query answering for RDF graphs with RDFS ontologies. In: ESWC (2019)
10. Bursztyn, D., Goasdoué, F., Manolescu, I.: Optimizing reformulation-based query answering in RDF. In: EDBT (2015)
11. Bursztyn, D., Goasdoué, F., Manolescu, I.: Teaching an RDBMS about ontological constraints. PVLDB **9**(12) (2016)
12. Cebiric, S., Goasdoué, F., Kondylakis, H., Kotzinos, D., Manolescu, I., Troullinou, G., Zneika, M.: Summarizing Semantic Graphs: A Survey. VLDB Journal (2018)
13. Chebotko, A., Lu, S., Fotouhi, F.: Semantics preserving SPARQL-to-SQL translation. Data Knowl. Eng. **68**(10) (2009)
14. Goasdoué, F., Guzewicz, P., Manolescu, I.: RDF Graph Summarization for First-sight Structure Discovery. The VLDB Journal (Apr 2020)
15. Goasdoué, F., Karanasos, K., Leblay, J., Manolescu, I.: View selection in semantic web databases. PVLDB **5**(2) (2011)
16. Goasdoué, F., Manolescu, I., Roatis, A.: Efficient query answering against dynamic RDF databases. In: EDBT (2013)
17. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. Web Sem. **3**(2-3) (2005)
18. Kontchakov, R., Rezk, M., Rodriguez-Muro, M., Xiao, G., Zakharyaschev, M.: Answering SPARQL queries over databases under OWL 2 QL entailment regime. In: ISWC (2014)
19. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. VLDB J. (2010)
20. Pham, M., Passing, L., Erling, O., Boncz, P.A.: Deriving an emergent relational schema from RDF data. In: WWW (2015)
21. Sidirourgos, L., Goncalves, R., Kersten, M., Nes, N., Manegold, S.: Column-store support for RDF data management: not all swans are white. PVLDB **1**(2) (2008)
22. Udrea, O., Pugliese, A., Subrahmanian, V.S.: GRIN: A graph based RDF index. In: AAAI (2007)
23. Urbani, J., Jacobs, C.J.H.: Adaptive low-level storage of very large knowledge graphs. In: WWW (2020)
24. Weiss, C., Karras, P., Bernstein, A.: Hexastore: Sextuple indexing for Semantic Web data management. PVLDB (2008)
25. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D.: Efficient RDF storage and retrieval in Jena2. In: SWDB (2003)