# An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario

Michael Schmidt[1,⋆], Thomas Hornung[1], Norbert Küchlin[1], Georg Lausen[1], and Christoph Pinkel[2]

[1] Freiburg University, Georges-Köhler-Allee 51, 79106 Freiburg, Germany
{mschmidt,hornungt,kuechlin,lausen}@informatik.uni-freiburg.de
[2] MTC Infomedia OHG, Kaiserstr. 26, 66121 Saarbrücken, Germany
c.pinkel@mtc-infomedia.de

**Abstract.** Efficient RDF data management is one of the cornerstones in realizing the Semantic Web vision. In the past, different RDF storage strategies have been proposed, ranging from simple triple stores to more advanced techniques like clustering or vertical partitioning on the predicates. We present an experimental comparison of existing storage strategies on top of the SP$^2$Bench SPARQL performance benchmark suite and put the results into context by comparing them to a purely relational model of the benchmark scenario. We observe that (1) in terms of performance and scalability, a simple triple store built on top of a column-store DBMS is competitive to the vertically partitioned approach when choosing a physical (predicate, subject, object) sort order, (2) in our scenario with real-world queries, none of the approaches scales to documents containing tens of millions of RDF triples, and (3) none of the approaches can compete with a purely relational model. We conclude that future research is necessary to further bring forward RDF data management.

## 1 Introduction

The Resource Description Framework [1] (RDF) is a standard format for encoding machine-readable information in the Semantic Web. RDF databases are collections of so-called "triples of knowledge", where each triple is of the form (*subject*,*predicate*,*object*) and models the binary relation *predicate* between the *subject* and the *object*. For instance, the triple (*Journal1*,*issued*,"*1940*") might be used to encode that the entity *Journal1* has been *issued* in year *1940*. By interpreting each triple as a graph edge from a *subject* to an *object* node with label *predicate*, RDF databases can be seen as labeled directed graphs.

To facilitate RDF data access, the W3C has standardized the SPARQL [2] query language, which bases upon a powerful graph pattern matching facility. Its very basic construct are simple triple graph patterns, which, during query evaluation, are matched against components in the RDF graph. In addition, different SPARQL operators can be used to compose more advanced graph patterns.

⋆ The work of this author was funded by DFG, grant GRK 806/2.

A. Sheth et al. (Eds.): ISWC 2008, LNCS 5318, pp. 82–97, 2008.

An efficient RDF storage scheme should support fast evaluation of such graph patterns and scale to RDF databases comprising millions (or even billions) of triples, as they are commonly encountered in today's RDF application scenarios (e.g., [3,4]). The straightforward relational implementation, namely a single `Triples` relation with three columns *subject*, *predicate*, and *object* that holds all RDF triples, seems not very promising: The basic problem with this approach is that the evaluation of composed graph patterns typically requires a large amount of expensive self-joins on this (possibly large) table. For instance, the query "Return the year of publication of *Journal1 (1940)*" might be expressed in SQL as follows (for readability, we use shortened versions of the RDF URIs).

```
SELECT T3.object AS yr
FROM Triples T1 JOIN Triples T2 ON T1.subject=T2.subject
     JOIN Triples T3 ON T1.subject=T3.subject
WHERE T1.predicate='type' AND T1.object='Journal' AND T2.predicate='title'
      AND T2.object='Journal 1 (1940)' AND T3.predicate='issued'
```
(1)

The `Triples` table access `T1` and the associated WHERE-conditions extract all *Journal* entities, `T2` fixes the title, and `T3` extracts the year of publication. We observe that even this rather simple query requires two *subject-subject* self-joins over the `Triples` table. Practical queries may involve much more self-joins.

To overcome this deficiency, other physical organization techniques for RDF have been proposed [5,6,7,8,9,10,11]. One notable idea is to cluster RDF data, i.e. to group entities that are similar in structure [9,10] and store them in flattened tables that contain all the shared properties. While this may significantly reduce the amount of joins in queries, it works out only for well-structured data. However, one strength of RDF is that it offers excellent support for scenarios with poorly structured information, where clustering is not a feasible solution.

A conceptually simpler idea is to set up one table for each unique *predicate* in the data [5,11], which can be seen as full *vertical partitioning* on the predicates. Each such predicate table consists of two columns (*subject*, *object*) and contains all *subject-object* pairs linked through the respective predicate. Data is then distributed across several smaller tables and, when the predicate is fixed, joins do not involve the whole set of triples. By physically sorting data on the *subject* column, *subject-subject* joins between two tables, a very frequent operation, can be realized in linear time (w.r.t. the size of the tables) by merging their subject columns [11]. In such a scenario, the query from above might be formulated as,

```
SELECT DI.object AS yr
FROM type TY JOIN title TI ON TY.subject=DT.subject
     JOIN issued IS ON TY.subject=IS.subject
WHERE TY.object='bench:Journal' AND TI.object='Journal 1 (1940)'
```
(2)

where `type`, `title`, and `issued` denote the corresponding predicates tables. Predicate selection now is implicit by the choice of the predicate table (i.e., no longer encoded in the `WHERE`-clause) and, given that the *subject*-column is sorted, both joins might be efficiently implemented as linear merge joins.

In the experiments in [11] on top of the Barton library data [12], vertical partitioning turns out to be clearly favorable to the triple table scheme and always competitive to clustering. Although the scenario is a reasonable choice that illustrates many advantages of vertical partitioning, several issues remain open. One point is that, in the partitioned scenario, efficient *subject-subject* merge joins on the predicate tables (which are possible whenever predicates are fixed) are a key to performance. However, when physically sorting table `Triples` by (*predicate*, *subject*, *object*), linear merge joins might also apply in a triple store.

A study of the Barton benchmark shows that one query (out of seven) requires no join on the triple (resp., predicate) table(s), and each two involve (a) a single *subject-subject* join, (b) two *subject-subject* joins, and (c) one *subject-subject* plus one *subject-object* join. Thus, none involves more than two joins. The simplicity of these join patterns to a certain degree contrasts with the Introduction of [11], where the authors state that "almost all interesting queries involve many self-joins" and motivate vertical partitioning using a five-way self-join query. We agree that real-world queries often involve complex join-patterns and see an urgent need for reevaluating the vertical approach in a more challenging scenario.

To this end, we present an experimental comparison of the triple and vertically partitioned scheme on top of the the $SP^2$Bench SPARQL benchmark [13]. The $SP^2$Bench queries implement meaningful requests in the DBLP scenario [14] and have been designed to test challenging situations that may arise in the context of SPARQL and Semantic Web data. In contrast to the Barton queries, they contain no aggregation, due to missing SPARQL language support. But except for this construct, they cover a much wider range of operator constellations, RDF data access paths, join patterns, and advanced features (e.g., Optional clauses, solution modifiers). The queries for the vertical and the triple store are obtained from a methodical SPARQL-to-SQL translation and reflect these characteristics.

To put our analysis into context, we consider two more scenarios. First, we test the *Sesame* SPARQL engine [15] as a representative SPARQL processor that relies on a native RDF store. Second, we translate the $SP^2$Bench scenario into a purely relational scheme, thus comparing the current state-of-the-art in RDF data management against established relational database technologies.

**Contributions.** Among others, our experiments show that (1) when triple tables are physically sorted by (*predicate*, *subject*, *object*), efficient merge joins can be exploited (just like in the vertical scheme) and the triple table approach becomes more competitive, (2) for the challenging $SP^2$Bench queries neither the vertical nor the triple scheme shows a good overall performance, and (3) while both schemes typically outperform the *Sesame* SPARQL engine, the purely relational encoding is almost always at least one order of magnitude faster. We conclude that there is an urgent need for future research in this area.

**Related Work.** An experimental comparison of the triple table and a vertically partitioned scheme has been provided in [5]. Among others, the authors note the additional costs of predicate table unions in the vertical scenario, which will be discussed later in this paper. Nevertheless, the setting in [5] differs in several aspects, e.g. in the vertically partitioned scheme the RDF schema layer was stored in separate tables and physical sorting on the *subject*-column (to allow for *subject-subject* merge joins), a central topic in our analysis, was not tested.

We point the interested reader to the experimental comparison of the triple and vertical storage scheme in [16]. This work has been developed independently from us. It presents a reevaluation of the experiments from [11] and, in this line, identifies situations where vertical partitioning is an insufficient solution. Several findings there are similar to our results. While the latter experiments are carried out in the Barton scenario (like the original experiments in [11]), we go one step further, i.e. perform tests in a different scenario and put the results into context by comparing them to a purely relational scheme, as well as a SPARQL engine.

The Berlin SPARQL Benchmark [17] is settled in an e-commerce scenario and strictly use-case driven. In contrast, the language-specific SP$^2$Bench suite used in this work covers a broader range of SPARQL/RDF constructs and, for this reason, is preferable for testing the generality of RDF storage schemes.

**Structure.** In the next section we summarize important characteristics of the SP$^2$Bench SPARQL performance benchmark [13], to facilitate the interpretation of the benchmark results. In Section 3 we then sketch the tested storage schemes and the methodical query translation into these scenarios. Finally, Section 4 contains the in-depth discussion of our experiments and a conclusion. In the remainder, we assume the reader to be familiar with RDF [1] and SPARQL [2].

## 2 The SP$^2$Bench Scenario

SP$^2$Bench [13] is settled in the DBLP [14] bibliographic scenario. Central to the benchmark is a data generator for creating DBLP-like RDF documents, which mirror characteristics and relations found in the original DBLP data. It relies on natural function families to capture social-world aspects encountered in the DBLP data, e.g. the citation system is modeled by powerlaw distributions, while limited growth functions approximate the number of publications per year. Supplementary, the SP$^2$Bench suite provides a set of meaningful SPARQL queries, covering a variety of SPARQL operator constellations and data access patterns.

According to DBLP, the SP$^2$Bench generator creates nine distinct types of bibliographic entities, namely Article, Journal, Inproceedings, Proceedings, Book, Incollection, PhDThesis, MastersThesis, and WWW documents, where each document is represented by a unique URI. In addition, there are persons that act as authors or editors. They are modeled by blank nodes.

Each document (resp., person) is described by a set of properties, such as *dc:title*, *dc:creator* (i.e., the author), or *swrc:isbn*. Outgoing citations are expressed through predicate *dcterms:references*, which points to a blank node of type *rdf:Bag* (a standard RDF container class) that links to the set of all document URIs referenced by the respective document. Attribute *dcterms:partOf* links inproceedings to the proceedings they appeared in; similarly, *swrc:journal* connects articles to journals. Several properties (e.g., *dc:creator*) are multi-valued.

The first part of Table 1 lists the number of document class instances of type Inproceedings, Proceedings, Article, Journal, Incollection, and the

**Table 1.** Key characteristics of documents generated by the SP²Bench generator

| #triples | #Inpr. | #Proc. | #Art. | #Journ. | #Inc. | #Oth. | #auth./#dist. | #prop. | file size | year |
|---|---|---|---|---|---|---|---|---|---|---|
| $10k$ | 169 | 6 | 916 | 25 | 18 | 0 | 1.5k/0.9k | 23+34 | 1.0MB | 1955 |
| $50k$ | 1.4k | 37 | 4.0k | 104 | 56 | 0 | 6.8k/4.1k | 23+34 | 5.1MB | 1967 |
| $250k$ | 9.2k | 213 | 17.1k | 439 | 173 | 39 | 34.5k/20.0k | 23+43 | 26MB | 1979 |
| $1M$ | 43.5k | 903 | 56.9k | 1.4k | 442 | 551 | 151.0k/82.1k | 23+44 | 106MB | 1989 |
| $5M$ | 255.2k | 4.7k | 207.8k | 4.6k | 1.4k | 1.4k | 898.0k/429.6k | 23+52 | 533MB | 2001 |
| $25M$ | 1.5M | 24.4k | 642.8k | 11.7k | 4.5k | 2.4k | 5.4M/2.1M | 25+52 | 2.7GB | 2015 |

remaining types *#Oth.* (Book, Www, PhD- and MastersThesis) for generated documents up to $25M$ RDF triples. Article and Inproceedings documents clearly dominate. The total number of authors (i.e., triples with predicate *dc:creator*) increases slightly super-linear to the total number of documents. This reflects the increasing average number of authors per paper in DBLP over time.

The table also lists the number *#prop.* of distinct properties. This value $x+y$ splits into $x$ "standard" attribute properties and $y$ bag membership properties *rdf:_1*, ..., *rdf:_y*, where $y$ depends on the maximum-sized reference list in the data. We observe that larger documents contain larger reference lists, and hence more distinct properties. As discussed later, this might complicate data processing in the vertically partitioned scenario. Finally, we list the physical size of the RDF file (in NTriples format) and the year up to which data was generated.

To support queries that access an author with fixed characteristics, the documents contain a special author, named after the mathematician Paul Erdös, who gets assigned 10 publications and 2 editor activities in-between 1940–1996. As an example, $Q8$ (Appendix A) extracts all persons with *Erdös Number* 1 or 2.[1]

## 3 The Benchmark Scenarios

We now describe the four benchmark scenarios in detail. The first system under consideration is (1) the *Sesame* [15] SPARQL engine. *Sesame* constitutes a query engine that, like the other three scenarios, relies on a physical DB backend. It is among the fastest SPARQL engines that have been tested in the context of the SP²Bench benchmark (cf. [13]) and has been chosen as a representative for the class of SPARQL engines. The remaining scenarios are (2) the triple table approach, (3) the vertically partitioned approach as described in [11], and (4) a purely relational DBLP model. They are all implemented on top of a relational DBMS. Accordingly, a translation of the SP²*Bench* SPARQL queries into SQL is required. We will sketch the detailed settings and our methodical query translation approaches for scenarios (2)-(4) in the remainder of this section. The resulting SQL queries are available online[2]; still, to be self-contained we will summarize their key characteristics when discussing the results in Section 4.

According to [11], to reach best performance all relational schemes should be implemented on top of a column-store DBMS, which stores data physically

[1] See http://www.oakland.edu/enp/.
[2] http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B/translations.html

by column rather than row (see [11] for the advantages of column-oriented systems in the RDF scenario). The C-Store research prototype [18] used in [11] misses several SQL features that are essential for the SP$^2$Bench queries (e.g. left joins), so we fall back on the *MonetDB* [19] column-store, a complete, industrial-strength relational DBMS. We note that MonetDB differs from C-Store in several aspects. First, data processing in MonetDB is memory-based while it is disk-based in C-Store. Moreover, C-Store exhibits a carefully optimized merge-join implementation (on top of run-length encoded data) and makes heavy use of this operation. Although we observe that MonetDB uses merge joins less frequently (cf. Section 4), the system is known for its performance and has recently been shown to be competitive to C-Store in the Barton Library RDF scenario [16].

### 3.1 The Triple Table Storage Scheme

In the *triple table* scheme a single table `Triples`(*subject*, *predicate*,*object*) holds all RDF triples. Methodical translations of SPARQL into this scheme have been proposed in [20,21,22]. The idea is to evaluate triple patterns separately against table `Triples`, then combining them according to the SPARQL operators in the query. Typically, SPARQL operator AND is expressed by a relational join, UNION by a SQL union, FILTER clauses result in WHERE-conditions, and OPTIONAL is modeled by a left outer join. For instance, SPARQL query $Q1$ (Appendix A) translates into query (1) from the Introduction (prefixes and data types are omitted). Observe that $Q1$ connects three patterns through two AND operators (denoted as "."), resulting in two SQL joins. The patterns are connected through variable *?journal* in *subject* position, so both are *subject-subject* joins. We emphasize that, although queries were translated manually, the scheme is very close to the approaches used by SPARQL engines that build on the relational model.

**Dictionary Encoding.** URIs and Literals tend to be long strings; they might blow up relational tables and make joins expensive. Therefore, we store integer keys instead of the string value, while keeping the key-value mapping in a `Dictionary`(*ID*,*val*) table (cf. [15,23,24,11]). Note that dictionary encoding implies additional joins with the `Dictionary` table in the translated queries.

**Implementation.** We sort data physically by (*predicate*, *subject*, *object*) rather than (*subject*, *predicate*, *object*). While this contrasts with the experiments in [11], we will show that this sort order makes the triple approach more competitive, because fast linear merge joins across property tables in the vertical scenario can now be realized by corresponding merge joins in the triple scenario.

We note that indexing in *MonetDB* differs from conventional DBMS; it interprets `INDEX` statements as advices, feeling free to ignore them and create its own indices.[3] Though, we issue a secondary BTree index for all remaining permutations of the *subject*, *predicate*, and *object* columns. The `Dictionary` table is physically sorted by *ID* and we request a secondary index on column *val*.

[3] See http://monetdb.cwi.nl/projects/monetdb/SQL/Documentation/Indexes.html.

### 3.2 The Vertically Partitioned Storage Scheme

The vertically partitioned relational store maintains one two-column table with schema (*subject*, *object*) for each unique predicate in the data. The query translation for the vertical scenario is similar to the triple table translation. The translation of SPARQL query $Q1$ into this scenario is exemplarily shown in the Introduction, query (2). Here, data is extracted from the predicate tables, so predicate value restrictions in the WHERE-clause are no longer necessary.

One major problem in the vertical scheme arises when predicates in queries are not fixed (i.e., when SPARQL variables occur in predicate position). Then, information cannot be extracted from a single predicate table, but queries must compute the union over *all* these tables. As discussed in Section 2 (Table 1), in our scenario the number of distinct properties (and hence, predicate tables) increases with document size. Consequently, such queries require more unions on large documents. This illustrates a basic drawback of the vertical approach: Query translation depends on the structure of the data and, what is even more urgent, queries may require a large number of unions over the predicate tables.

**Implementation.** We sort the predicate tables physically on (*subject*, *object*) and issue an additional secondary BTree index on columns (*object*, *subject*). Dictionary encoding is implemented analogously to the triple scheme.

### 3.3 The Purely Relational Scheme

We started from scratch and developed an Entity Relationship Model (ERM) of DBLP. Using ERM translation techniques, we end up with the following tables, where primary keys are underlined and foreign keys are marked by prefix "*fk_*".

- `Document`(*ID*,*address*,*booktitle*,*isbn*,...,*stringid*,*title*,*volume*)
- `Document_homepage`(*fk_document*,*homepage*)
- `Document_seeAlso`(*fk_document*,*seeAlso*)
- `Venue`(*ID*,*fk_document*,*fk_venue_type*)
- `Publication`(*ID*,*chapter*,*fk_document*,*fk_publication_type*,*fk_venue*,*pages*)
- `Publication_cdrom`(*fk_publication*,*cdrom*)
- `Abstract`(*fk_publication*,*txt*)
- `PublicationType`(*ID*,*name*) and `VenueType`(*ID*,*name*)
- `Person`(*ID*,*name*,*stringid*)
- `Author`(*fk_person*,*fk_publication*) and `Editor`(*fk_document*,*fk_person*)
- `Reference`(*fk_from*,*fk_to*)

The scheme distinguishes between venues (i.e., JOURNAL and PROCEEDINGS) and publications (such as ARTICLE, INPROCEEDINGS, or BOOK). The dictionary tables `PublicationType` and `VenueType` contain integer *ID*s for the respective venue and publication classes. Table `Document` constitutes a base table for both document types, containing properties that are common to both venues and publications. Supplementary, `Venue` and `Publication` store the properties that are specific for the respective type. For instance, if a new BOOK document is inserted, its base properties are stored in table `Document`, while publication-type specific properties (e.g., *chapter*) are stored in table `Publication`. The entries are linked through foreign key `Publication`.*fk_document*; the type (in this case Book) is fixed by linking `Publication`.*fk_publication_type* to the Book ID in `PublicationType`. Properties *foaf:homepage*, *rdf:seeAlso*, and *bench:cdrom* are multi-valued in the $\mathrm{SP}^2$Bench scenario, so they are stored in the separate tables `Document_homepage`, `Document_seeAlso`, and `Publication_cdrom`. We use a distinguished `Abstract` table for the larger-than-average abstract strings.

Finally, there is one table `Person` that stores person information, two tables `Author` and `Editor` that store the author and editor activity of persons, and a table `Reference` that contains all references between documents.

**Implementation.** The scheme was implemented in *MonetDB* exactly as described above, using the specified `PRIMARY` and `FOREIGN KEY` constraints, without additional indices. In the sense of a relational schema we omit prefix definitions (such as "rdf:", "dc:"). The data was translated using a conversion script.

## 4 Experimental Results

**Setting.** The experiments were carried out on a Desktop PC running ubuntu v7.10 gutsy Linux, with Intel Core2 Duo E6400 2.13GHz CPU and 3GB DDR2 667 MHz nonECC physical memory. We used a 250GB Hitachi P7K500 SATA-II hard drive with 8MB Cache. The relational schemes were executed with *MonetDB* mserver v5.5.0, using the (more efficient) algebra frontend (flag "-G").

As discussed in Section 3, we tested (1) the *Sesame* v2.0 engine *SP* (coupled with its native storage layer, providing all possible combinations of indices) and three *MonetDB* scenarios, namely (2) the triple store *TR*, (3) the vertically partitioned store *VP*, and (4) the purely relational scheme *RS*. We report on user (`usr`), system (`sys`), and elapsed time (`total`). While `usr` and `sys` were extracted from the */proc* file system, elapsed time was measured through a timer. *MonetDB* follows a client-server architecture and we provide the sum of the `usr` and `sys` times of the client and server processes. Note that the experiments were run on a DuoCore CPU, where the linux kernel sums up `usr` and `sys` of the individual processor units, so `usr+sys` might be greater than `total`.

For all scenarios we carried out three runs over all queries on documents of $10k$, $50k$, $250k$, $1M$, $5M$, and $25M$ triples, setting a 30 minutes timeout and $2GB$ memory limit (using *ulimit*) per query. As our primary interest is the basic performance of the approaches (rather than caching or learning strategies), we performed *cold* runs, i.e. destroyed the database in-between each two consecutive runs and always restarted it before evaluating a query. We provide average times and omit the deviation from the average (which was always negligible).

**Discussion of the Benchmark Results.** All results were verified by comparing the outcome of the engines among each other (where possible). Table 2 summarizes the query result sizes and the physical DB sizes for each scenario on all documents. The *VP* scheme requires less disk space than *TR* for large documents, since predicates are not explicitly stored for each triple. For *Sesame*, indices

**Table 2.** Query result sizes on documents up to 25$M$ triples and physical DB size

| | Number of query results for individual queries | | | | | | | | | | | | | Phys. DB size (MB) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $Q1$ | $Q2$ | $Q3a$ | $Q3b$ | $Q3c$ | $Q4$ | $Q5a/b$ | $Q6$ | $Q7$ | $Q8$ | $Q9$ | $Q10$ | $Q11$ | $SP$ | $TR$ | $VP$ | $RS$ |
| 10$k$ | 1 | 147 | 846 | 9 | 0 | 23.2k | 155 | 229 | 0 | 184 | 4 | 166 | 10 | 3 | 3 | 6 | 4 |
| 50$k$ | 1 | 965 | 3.6k | 25 | 0 | 104.7k | 1.1k | 1.8k | 2 | 264 | 4 | 307 | 10 | 14 | 5 | 8 | 5 |
| 250$k$ | 1 | 6.2k | 15.9k | 127 | 0 | 542.8k | 6.9k | 12.1k | 62 | 332 | 4 | 452 | 10 | 69 | 18 | 20 | 13 |
| 1$M$ | 1 | 32.8k | 52.7k | 379 | 0 | 2.6M | 35.2k | 62.8k | 292 | 400 | 4 | 572 | 10 | 277 | 63 | 58 | 42 |
| 5$M$ | 1 | 248.7k | 192.4k | 1.3k | 0 | 18.4M | 210.7k | 417.6k | 1.2k | 493 | 4 | 656 | 10 | 1376 | 404 | 271 | 195 |
| 25$M$ | 1 | 1.9M | 594.9k | 4.1k | 0 | n/a | 696.7k | 1.9M | 5.1k | 493 | 4 | 656 | 10 | 6928 | 2395 | 1168 | 913 |

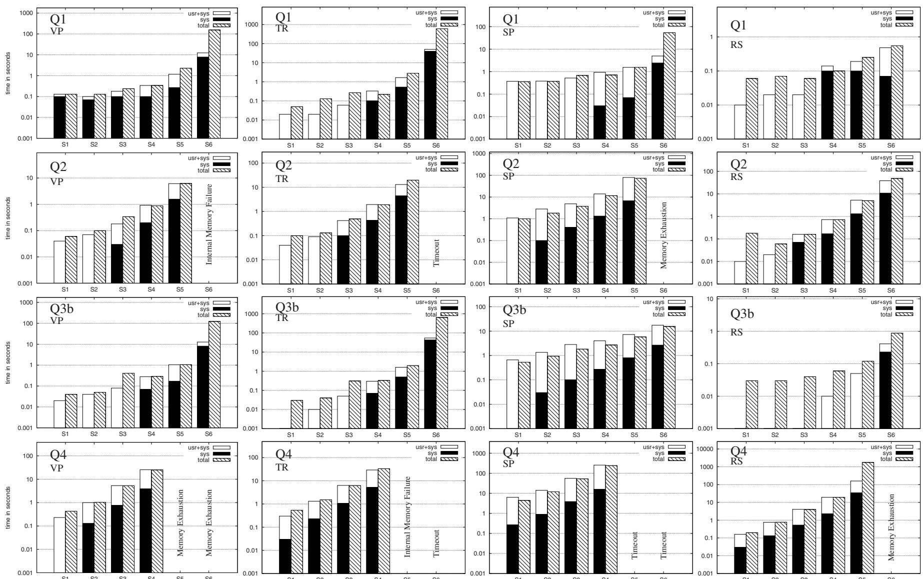


**Fig. 1.** Results on S1=10k, S2=50k, S3=250k, S4=1M, S5=5M, and S6=25M triples

occupy more than half of the required space. In *RS* there is no redundancy, no dictionary encoding, and no prefixes are stored, so least space is required.

The query execution times are shown in Figures 1, 2, and 3 (the $y$-axes are always in log scale). Please note that the individual plots scale differently.

**Q1. Return the year of publication of "Journal 1 (1940)".**

This simple query returns exactly one result on all documents. The *TR* and *VP* translations are shown in the Introduction. The *RS* query joins tables `Venue`, `Document`, and `VenueType` on the connecting foreign keys and then filters for `VenueType`.*name*="Journal" and `Document`.*title*="Journal 1 (1940)".

We observe that both the *TR* and *VP* scenario scale well for documents up to 5$M$ triples, but `total` time explodes for 25$M$ triples. The gap between `total` and `usr+sys` for 25$M$ indicates that much time is spent in waiting for data being read from or written to disk, which is caused by query execution plans (QEPs) that

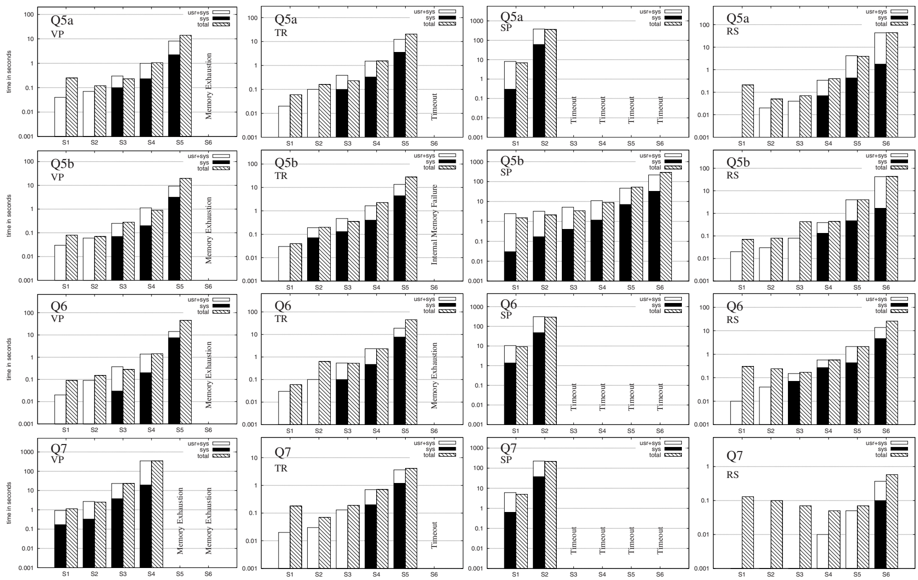


**Fig. 2.** Results on S1=10k, S2=50k, S3=250k, S4=1M, S5=5M, and S6=25M triples

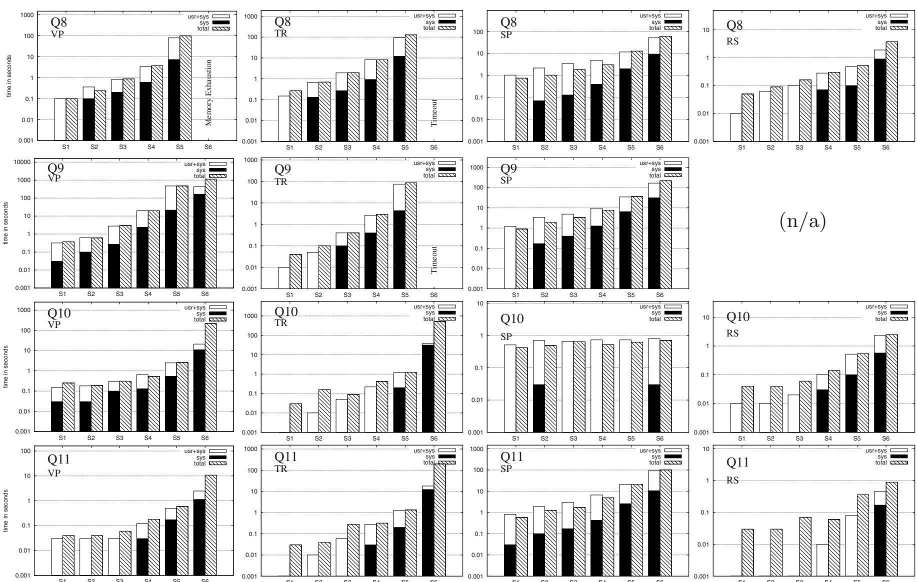


**Fig. 3.** Results on S1=10k, S2=50k, S3=250k, S4=1M, S5=5M, and S6=25M triples

involve expensive fetch joins, instead of efficient *subject-subject* merge joins. We claim that using merge joins would be more efficient here. Due to this deficiency, both *Sesame* and the *RS* scenario outperform the *TR* and *VP* schemes.

**Q2. Extract all inproceedings with properties** *dc:creator*, *bench:booktitle*, *dc:title*, *swrc:pages*, *dcterms:partOf*, *rdfs:seeAlso*, *foaf:homepage*, *dcterms:issued*, **and optionally** *bench:abstract*, **including these properties.**

$Q2$ implements a star-join-like graph pattern. Result size grows with document size (cf. Table 2) and the solution modifier ORDER BY forces result ordering. The nine outer SPARQL triple patterns translate into nine predicate (triple) table accesses in the *VP* (*TR*) scenario, connected through eight *subject-subject* joins, due to variable *?inproc*. The OPTIONAL clause causes an additional left outer join. The *RS* query gathers all relevant information from tables `Document`, `Publication`, `PublicationType`, `Author`, `Person`, `Document_seeAlso`, `Venue`, and `Document_homepage`, and also contains a left outer join with table `Abstract`.

Like for $Q1$, the *subject-subject* joins should be realized by merge joins in the *TR* and *VP* scenario, but MonetDB chooses QEPs that mostly use fetch joins, involving merge joins only in few cases. These fetch joins consume the major part of execution time. Lastly, none of both schemes succeeds for the $25M$ triples document. *Sesame* is about one order of magnitudes slower. The *RS* scheme requires less joins and is significantly faster than the other approaches.

**Q3abc. Select all articles with property (a)** *swrc:pages*, **(b)** *swrc:month*, **or (c)** *swrc:isbn*.

We restrict on a discussion of $Q3b$, as the results for $Q3a$ and $Q3c$ are similar. As explained in [13], the FILTER in $Q3b$ selects about 0.65% of all articles. The *TR* translation contains a *subject-subject* join on table `Triples` and a WHERE value-restrictions for predicate *swrc:month*. Although variable *?property* occurs in *predicate* position, we chose a *VP* translation that does not compute the union of all predicate tables, but operates directly on the table for predicate *swrc:month*, which is implicitly fixed by the FILTER. The *RS* translation is straightforward.

The *VP* approach is a little faster than *TR*, because it operates on top of the *swrc:month* predicate table, instead of the full triples table. The query contains only one *subject-subject* join, and we observe that the *VP* and *TR* approaches explode for the $25M$ document, again due to expensive fetch joins (cf. $Q1$, $Q2$). *Sesame* is competitive and scales even better, while *RS* shows best performance.

**Q4. Select all distinct pairs of article author names for authors that have published in the same journal.**

$Q4$ contains a long graph chain, i.e. variables *?name1* and *?name2* are linked through the articles that different authors have published in the same journal. When translated into *TR* and *VP*, the chain is mapped to a series of *subject-subject*, *subject-object*, and *object-object* joins. The *RS* query gathers all articles and their authors from the relevant tables twice and joins them on `Venue`.*ID*.

As apparent from Table 2, the query computes very large results. Due to the *subject-object* and *object-object* joins, the *TR* and *VP* scenarios have to compute many expensive (non-merge) joins, which makes the approaches scale poorly. *Sesame* is one order of magnitude slower. In contrast, *RS* involves simpler joins (e.g., efficient joins on foreign keys) and shows the best performance.

**Q5ab. Return the names of all persons that occur as author of at least one inproceeding and at least one article.**

*Q*5*a* joins authors implicitly on author names (through the Filter condition), while *Q*5*b* explicitly joins on variable *?person*. Although in general not equivalent, the one-to-one mapping between authors and their names in SP$^2$Bench implies equivalence of *Q*5*a* and *Q*5*b*. All translations share these join characteristics, i.e. all translations of *Q*5*a* model the join by an equality condition in the SQL Where-clause, whereas translations of *Q*5*b* contain an explicit SQL Join.

*Sesame* scales bad for *Q*5*a*, probably due to the implicit join (it performs much better for *Q*5*b*). In the SQL scenarios there are no big differences between implicit and explicit joins; such situations are resolved by relational optimizers.

**Q6. Return, for each year, the set of all publications authored by persons that have not published in years before.**

*Q*6 implements closed world negation (CWN), expressed through a combination of operators Optional, Filter, and bound. The block outside the Optional computes *all* publications and the inner one constitutes earlier publications from authors that appear outside. The outer Filter then retains all publications for which *?author2* is unbound, i.e. those from newcomers. In the *TR* and *VP* translation, a left outer join is used to connect the outer to the inner part. The *RS* query extracts, for each year, all publications and their authors, and uses a SQL NOT EXISTS clause to filter away authors without prior publications.

One problem in the *TR* and *VP* queries is the left join on top of a less-than comparison, which complicates the search for an efficient QEP. In addition, both queries contain each two *subject-object* joins on the left and on the right side of the left outer join. Ultimately, both scale poorly. Also *Sesame* scales very bad. In contrast, the purely relational encoding is elegant and much more efficient.

**Q7. Return the titles of all papers that have been cited at least once, but not by any paper that has not been cited itself.**

This query implements a double-CWN scenario. Due to the nested Optional clauses, the *TR* and *VP* translations involve two nested left outer joins with join-intensive subexpressions. The *VP* translation is complicated by three unions of all predicate tables, caused by the SPARQL variables *?member2*, *?member3*, and *?member4* in predicate position. When encoding them at the bottom of the evaluator tree, the whole query builds upon these unions and the benefit of sorted and indexed predicate tables gets lost. We tested different versions of the query and decided for the most performant (out of the tested variants), where we pulled off the outermost union, thus computing the union of subexpressions rather than individual tables. The *RS* query uses two nested SQL Not In-clauses to express double negation. We could have used nested Not Exists-clauses instead (cf. *Q*6), but decided to vary, to test the impact of both operators.

Due to the unbound predicates, the *VP* approach has severe problems in evaluating this query and behaves worse than the *TR* scheme. This illustrates the disadvantages of the vertical approach in scenarios where unbound predicates occur. *Sesame* also behaves very bad, while the nested NOT IN-clause in *RS*, a common construct in relational queries, constitutes the only practical solution.

**Q8. Compute authors that have published with Paul Erdoes or with an author that has published with Paul Erdoes.**

$Q8$ contains a SPARQL UNION operator, so all translations contain a SQL union. The *TR* and *VP* versions of this query are straightforward. The *RS* translation separately retrieves persons that have published with Paul Erdoes and persons that have published with one of its coauthors (each from the `Author` and the `Person` table), and afterwards computes the union of both person sets.

Again, the *TR* scenario turns out to be competitive to *VP*, but both schemes fail to find an efficient QEP for large documents, due to the *subject-object* and *object-object* joins and the additional non-equality WHERE-condition over the *subject* and *object* columns. The *Sesame* engine scales surprisingly well for this query, but is still one order of magnitude slower than the relational scheme.

**Q9. Return incoming and outgoing properties of persons.**

Both parts of the union in $Q9$ contain a fully unbound triple pattern, which selects all RDF database triples. The *TR* translation is straightforward. Concerning the unbound *?predicate* variable, we again pulled off the union of the predicate tables in the *VP* scenario, thus computing the same query separately for each predicate table and building the union of the results afterwards. As discussed in $Q7$, this was more efficient than the union at the bottom of the operator tree. The result size is always 4 (the first part constitutes properties *dc:creator* and *swrc:editor*, and the second one *rdf:type* and *foaf:name*). A meaningful *RS* translation of this query, which accesses schema information, is not possible: In *RS*, the properties are encoded as (fixed) table attributes names.[4]

Although a little bit slower than the *TR* approach for small documents, *VP* succeeds in evaluating the $25M$ triple document. Though, both approaches seem to have problems with the unbound triple pattern and scale poorly. *Sesame*'s native store offers better support, but is still far from being performant.

**Q10. Return all subjects that stand in any direct relation with Paul Erdoes. In our scenario the query can be reformulated as "Return publications and venues in which Paul Erdoes is involved as author or editor, respectively".**

$Q10$ implements an *object* bound-only RDF access path. The *TR* and *RS* translations are standard. Due to the unbound variable *?predicate*, the *VP* query involves a union of the predicate tables. As for $Q9$, the implementation of this union on top of the operator tree turned out to be the most performant solution.

Recalling that "Paul Erdoes" is active between 1940 and 1996, the result size has an upper bound (cf. Table 2 for the $5M$ and $25M$ documents). *VP* and *TR* show very similar behavior. As illustrated by the results of *Sesame*, this query can be realized in constant time (with an appropriate index). The index

[4] A lookup query for fixed values in the DBMS system catalog is not very interesting.

selection strategy of MonetDB in *TR* and *VP* is clearly suboptimal. *RS* scales much better, but (in contrast to *Sesame*) still depends on the document size.

**Q11. Return (up to) 10 electronic edition URLs starting from the $51^{st}$ publication, in lexicographical order.**

*Q*11 focuses on the combination of solution modifiers Order By, Limit, and Offset, which arguably remains the key challenge in all three translations.

The *VP* query operates solely on the predicate table for *rdfs:seeAlso* and, consequently, is a little faster than *TR*. *Sesame* scales superlinearly and is slower than both. Once more, *RS* dominates in terms of performance and scalability.

**Conclusion.** Our results bring many interesting findings. First, the MonetDB optimizer often produced suboptimal QEPs in the *VP* and *TR* scenario (e.g., for *Q*1, *Q*2, and *Q*3*b* not all *subject-subject* join patterns were realized by merge joins). This shows that relational optimizers may have problems to cope with the specific challenges that arise in the context of RDF. Developers should be aware of this when implementing RDF schemes on top of relational systems.

Using the $SP^2$Bench queries we have identified limitations of the vertical approach. We observe performance bottlenecks in complex scenarios with unbound predicates (e.g., *Q*7), for challenging operator constellations (e.g., CWN-queries *Q*6, *Q*7), and identified queries with many non-*subject-subject* joins as a serious weakness of the *VP* scheme. While the latter weakness has been noted before in [11], our experiments reveal the whole extent of this problem. The materialization of path expressions might improve the performance of such queries [11], but comes with additional costs (e.g., disk space), and is not a general solution.

Another finding is that a triple store with physical (*predicate*,*subject*,*object*) sort order is more competitive to the vertical scheme, and might even outperform it for queries (e.g., *Q*7) with unbound predicates (cf. [16]). This relativizes the results from [11], where the triple store was implemented with (*subject*, *predicate*, *object*) sort order and only tested in combination with a row-store DBMS.

Finally, none of the tested RDF schemes was competitive to a comparable purely relational encoding. Although relational schemata are domain-specific and, in this regard, optimized for the underlying scenario, we observed a gap of at least one order of magnitude for almost all queries already on small documents, typically increasing with document size. We therefore are convinced that there is still room for optimization in RDF storage schemes, to reduce the gap between RDF and relational data processing and bring forward the Semantic Web vision.

**Acknowledgment.** The authors thank the MonetDB team for its support in setting up MonetDB and interesting discussions on RDF storage technologies.

## References

1. W3C: Resource Description Framework (RDF), `http://www.w3.org/RDF/`
2. W3C: SPARQL Query Language, `http://www.w3.org/TR/rdf-sparql-query/`

3. Bizer, C., Cyganiak, R.: D2R Server – Publishing the DBLP Bibliography Database (2007), http://www4.wiwiss.fu-berlin.de/dblp/
4. Tauberer, J.: U.S. Census RDF Data, http://www.rdfabout.com/demo/census/
5. Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D.: On Storing Voluminous RDF Descriptions: The case of Web Portal Catalogs. In: WebDB (2001)
6. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 54–68. Springer, Heidelberg (2002)
7. Bonstrom, V., Hinze, A., Schweppe, H.: Storing RDF as a Graph. In: Web Congress, pp. 27–36 (2003)
8. Theoharis, Y., Christophides, V., Karvounarakis, G.: Benchmarking RDF Representations of RDF/S Stores. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) ISWC 2005. LNCS, vol. 3729, pp. 685–701. Springer, Heidelberg (2005)
9. Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An Efficient SQL-based RDF Querying Scheme. In: VLDB, pp. 1216–1227 (2005)
10. Wilkinson, K.: Jena Property Table Implementation. In: International Workshop on Scalable Semantic Web Knowledge Base, pp. 35–46 (2006)
11. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: VLDB, pp. 411–422 (2007)
12. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Using the Barton libraries dataset as an RDF benchmark. Technical report, MIT-CSAIL-TR-2007-036, MIT (2007)
13. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP$^2$Bench: A SPARQL Performance Benchmark. Technical report, arXiv:0806.4627v1 cs.DB (2008)
14. Ley, M.: DBLP Database, http://www.informatik.uni-trier.de/~ley/db/
15. openRDF.org: Home of Sesame, http://www.openrdf.org/documentation.jsp
16. Sidirourgos, L., Goncalves, R., Kersten, M., Nes, N., Manegold, S.: Column-store Support for RDF Data Management: not all swans are white. In: VLDB (2008)
17. Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark, http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/
18. Stonebraker, M., et al.: C-store: a Column-oriented DBMS. In: VLDB, pp. 553–564 (2005)
19. CWI Amsterdam: MonetDB, http://monetdb.cwi.nl/
20. Chebotko, A., Lu, S., Yamil, H.M., Fotouhi, F.: Semantics Preserving SPARQL-to-SQL Query Translation for Optional Graph Patterns. Technical report, TR-DB-052006-CLJF (2006)
21. Cyganiac, R.: A Relational Algebra for SPARQL. Technical report, HP Bristol
22. Harris, S.: SPARQL Query Processing with Conventional Relational Database Systems. In: SSWS (2005)
23. SourceForge: Jena2, http://jena.sourceforge.net/DB/index.html
24. Harris, S., Gibbins, N.: 3store: Efficient Bulk RDF Storage. In: PSSS (2003)

# A SP²Bench SPARQL Benchmark Queries

**Q1**
```
SELECT ?yr
WHERE {
  ?journal rdf:type bench:Journal.
  ?journal dc:title "Journal 1 (1940)"^^xsd:string.
  ?journal dcterms:issued ?yr }
```

**Q2**
```
SELECT ?inproc ?author ?booktitle ?title
       ?proc ?ee ?page ?url ?yr ?abstract
WHERE {
  ?inproc rdf:type bench:Inproceedings.
  ?inproc dc:creator ?author.
  ?inproc bench:booktitle ?booktitle.
  ?inproc dc:title ?title.
  ?inproc dcterms:partOf ?proc.
  ?inproc rdfs:seeAlso ?ee.
  ?inproc swrc:pages ?page.
  ?inproc foaf:homepage ?url.
  ?inproc dcterms:issued ?yr
  OPTIONAL { ?inproc bench:abstract ?abstract }
} ORDER BY ?yr
```

**Q3**
```
(a) SELECT ?article
    WHERE {
      ?article rdf:type bench:Article.
      ?article ?property ?value
      FILTER (?property=swrc:pages) }
(b) Q3a, but "swrc:month" instead of "swrc:pages"
(c) Q3a, but "swrc:isbn" instead of "swrc:pages"
```

**Q4**
```
SELECT DISTINCT ?name1 ?name2
WHERE {
  ?article1 rdf:type bench:Article.
  ?article2 rdf:type bench:Article.
  ?article1 dc:creator ?author1.
  ?author1 foaf:name ?name1.
  ?article2 dc:creator ?author2.
  ?author2 foaf:name ?name2.
  ?article1 swrc:journal ?journal.
  ?article2 swrc:journal ?journal
  FILTER (?name1<?name2) }
```

**Q5**
```
(a) SELECT DISTINCT ?person ?name
    WHERE {
      ?article rdf:type bench:Article.
      ?article dc:creator ?person.
      ?inproc rdf:type bench:Inproceedings.
      ?inproc dc:creator ?person2.
      ?person foaf:name ?name.
      ?person2 foaf:name ?name2
      FILTER(?name=?name2)
    }
(b) SELECT DISTINCT ?person ?name
    WHERE {
      ?article rdf:type bench:Article.
      ?article dc:creator ?person.
      ?inproc rdf:type bench:Inproceedings.
      ?inproc dc:creator ?person.
      ?person foaf:name ?name
    }
```

**Q6**
```
SELECT ?yr ?name ?doc
WHERE {
  ?class rdfs:subClassOf foaf:Document.
  ?doc rdf:type ?class.
  ?doc dcterms:issued ?yr.
  ?doc dc:creator ?author.
  ?author foaf:name ?name
  OPTIONAL {
    ?class2 rdfs:subClassOf foaf:Document.
    ?doc2 rdf:type ?class2.
    ?doc2 dcterms:issued ?yr2.
    ?doc2 dc:creator ?author2
    FILTER (?author=?author2 && ?yr2<?yr)
  } FILTER (!bound(?author2))
}
```

**Q7**
```
SELECT DISTINCT ?title
WHERE {
  ?class rdfs:subClassOf foaf:Document.
  ?doc rdf:type ?class.
  ?doc dc:title ?title.
  ?bag2 ?member2 ?doc.
  ?doc2 dcterms:references ?bag2
  OPTIONAL {
    ?class3 rdfs:subClassOf foaf:Document.
    ?doc3 rdf:type ?class3.
    ?doc3 dcterms:references ?bag3.
    ?bag3 ?member3 ?doc
    OPTIONAL {
      ?class4 rdfs:subClassOf foaf:Document.
      ?doc4 rdf:type ?class4.
      ?doc4 dcterms:references ?bag4.
      ?bag4 ?member4 ?doc3
    } FILTER (!bound(?doc4))
  } FILTER (!bound(?doc3))
}
```

**Q8**
```
SELECT DISTINCT ?name
WHERE {
  ?erdoes rdf:type foaf:Person.
  ?erdoes foaf:name "Paul Erdoes"^^xsd:string.
  {
    ?doc dc:creator ?erdoes.
    ?doc dc:creator ?author.
    ?doc2 dc:creator ?author.
    ?doc2 dc:creator ?author2.
    ?author2 foaf:name ?name
    FILTER (?author!=?erdoes &&
            ?doc2!=?doc &&
            ?author2!=?erdoes &&
            ?author2!=?author)
  } UNION {
    ?doc dc:creator ?erdoes.
    ?doc dc:creator ?author.
    ?author foaf:name ?name
    FILTER (?author!=?erdoes)
  }
}
```

**Q9**
```
SELECT DISTINCT ?predicate
WHERE {
  {
    ?person rdf:type foaf:Person.
    ?subject ?predicate ?person
  } UNION {
    ?person rdf:type foaf:Person.
    ?person ?predicate ?object
  }
}
```

**Q10**
```
SELECT ?subj ?pred
WHERE {
  ?subj ?pred person:Paul_Erdoes
}
```

**Q11**
```
SELECT ?ee
WHERE {
  ?publication rdfs:seeAlso ?ee
} ORDER BY ?ee
LIMIT 10
OFFSET 50
```