The VLDB Journal (2010) 19:91–113
DOI 10.1007/s00778-009-0165-y

SPECIAL ISSUE PAPER

The RDF-3X engine for scalable management of RDF data

Thomas Neumann · Gerhard Weikum

Received: 13 January 2009 / Revised: 19 June 2009 / Accepted: 7 August 2009 / Published online: 1 September 2009

Abstract RDF is a data model for schema-free structured information that is gaining momentum in the context of Semantic-Web data, life sciences, and also Web 2.0 platforms. The "pay-as-you-go" nature of RDF and the flexible pattern-matching capabilities of its query language SPARQL entail efficiency and scalability challenges for complex queries including long join paths. This paper presents the RDF-3X engine, an implementation of SPARQL that achieves excellent performance by pursuing a RISC-style architecture with streamlined indexing and query processing. The physical design is identical for all RDF-3X databases regardless of their workloads, and completely eliminates the need for index tuning by exhaustive indexes for all permutations of subject-property-object triples and their binary and unary projections. These indexes are highly compressed, and the query processor can aggressively leverage fast merge joins with excellent performance of processor caches. The query optimizer is able to choose optimal join orders even for complex queries, with a cost model that includes statistical synopses for entire join paths. Although RDF-3X is optimized for queries, it also provides good support for efficient online updates by means of a staging architecture: direct updates to the main database indexes are deferred, and instead applied to compact differential indexes which are later merged into the main indexes in a batched manner. Experimental studies with several large-scale datasets with more than 50 million RDF triples and benchmark queries that include pattern matching, manyway star-joins, and long path-joins demonstrate that RDF-3X can outperform the previously best alternatives by one or two orders of magnitude.

Keywords RDF · Batched updates · Indexing · Query processing · Query optimization · SPARQL · Database engine

T. Neumann (✉) · G. Weikum
Max-Planck-Institut für Informatik, Saarbrücken, Germany
e-mail: neumann@mpi-inf.mpg.de

G. Weikum
e-mail: weikum@mpi-inf.mpg.de

1 Introduction

1.1 Motivation and problem

The resource description framework (RDF) data model has been around for a decade. It has been designed as a flexible representation of schema-relaxable or even schema-free information for the Semantic Web [61]. In the commercial IT world, RDF has not received much attention until recently, but now it seems that RDF is building up a strong momentum. Semantic-Web-style ontologies and knowledge bases with millions of facts from Wikipedia and other sources have been created and are available online [4,55,65]. E-science data repositories support RDF as an import/export format and also for selective (thus, query-driven) data extractions, most notably, in the area of life sciences (e.g., [8,58]). Finally, Web 2.0 platforms for online communities are considering RDF as a non-proprietary exchange format and as an instrument for the construction of information mashups [25,26,44].

In RDF, all data items are represented in the form of $(subject, predicate, object)$ triples, also known as $(subject, property, value)$ triples. For example, information about the song "Changing of the Guards" performed by Patti Smith could include the following triples:
$(id1, hasType, "song")$,
$(id1, hasTitle, "Changing\ of\ the\ Guards")$,
$(id1, performedBy, id2)$,

(*id*2, *hasName*, "*Patti Smith*)",
(*id*2, *bornOn*, "*December* 30, 1946)",
(*id*2, *bornIn*, *id*3),
(*id*3, *hasName*, "*Chicago*)",
(*id*3, *locatedIn*, *id*4),
(*id*4, *hasName*, "*Illinois*)",
(*id*1, *composedBy*, *id*5),
(*id*5, *hasName*, "*Bob Dylan*)",
and so on.

Note that, although predicate names such as "performedBy" or "composedBy" resemble attributes, there is no database schema; the same database may contain triples about songs with predicates "performer", "hasPerformed", "sungBy", "composer", "creator", etc. A schema may emerge in the long run (and can then be described by the RDF Vocabulary Description Language). In this sense, the notion of RDF triples fits well with the modern notion of data spaces and its "pay as you go" philosophy [21]. Compared to an entity-relationship model, both an entity's attributes and its relationships to other entities are represented by predicates. All RDF triples together can be viewed as a large (instance-level) graph.

The SPARQL query language is the official standard for searching over RDF repositories. It supports conjunctions (and also disjunctions) of triple patterns, the counterpart to select-project-join queries in a relational engine. For example, we can retrieve all performers of songs composed by Bob Dylan by the following SPARQL query:

```
Select ?n Where {
  ?p <hasName> ?n. ?s <performedBy> ?p.
  ?s <composedBy> ?c. ?c <hasName> "Bob Dylan"
}
```

Here each of the conjunctions, denoted by a dot, corresponds to a join. The whole query can also be seen as graph pattern that needs to be matched in the RDF data graph. In SPARQL, predicates can also be variables or wildcards, thus allowing schema-agnostic queries.

RDF engines for storing, indexing, and querying have been around for quite a few years; especially, the Jena framework by HP Labs has gained significant popularity [64], and Oracle also provides RDF support for semantic data integration in life sciences and enterprises [13,42]. However, with the exception of the VLDB 2007 paper by Abadi et al. [1] (and very recent work presented at VLDB 2008 [39,50,60]), none of the prior implementations could demonstrate convincing efficiency, failing to scale up towards large datasets and high load. [1] achieves good performance by grouping triples with the same property name into property tables, mapping these onto a column store, and creating materialized views for frequent joins.

Managing large-scale RDF data includes technical challenges for the storage layout, indexing, and query processing:

1. The absence of a global schema and the diversity of predicate names pose major problems for the *physical database design*. In principle, one could rely on an auto-tuning "wizard" to materialize frequent join paths; however, in practice, the evolving structure of the data and the variance and dynamics of the workload turn this problem into a complex sisyphus task.
2. By the fine-grained modeling of RDF data—triples instead of entire records or entities—queries with a large number of joins will inherently form a large part of the workload, but the join attributes are much less predictable than in a relational setting. This calls for specific choices of *query processing algorithms*, and for careful *optimization of complex join queries*; but RDF is meant for on-the-fly applications over data spaces, so the optimization takes place at query run-time.
3. As join-order and other execution-plan optimizations require data statistics for selectivity estimation, an RDF engine faces the problem that a suitable granularity of *statistics gathering* is all but obvious in the absence of a schema. For example, single-dimensional histograms on all attributes that occur in the workload's where clauses—the state-of-the-art approach in relational systems—is unsuitable for RDF, as it misses the effects of long join chains or large join stars over many-to-many relationships.
4. Although RDF uses XML syntax and SPARQL involves search patterns that resemble XML path expressions, the fact that RDF triples form a graph rather than a collection of trees is a major difference to the more intensively researched settings for XML.

1.2 Contribution and outline

This paper gives a comprehensive, scalable solution to the above problems. It presents a complete system, coined *RDF-3X* (for RDF Triple eXpress), designed and implemented from scratch specifically for the management and querying of RDF data. RDF-3X follows the rationale advocated in [30,54] that data-management systems that are designed for and customized to specific application domains can outperform generic mainstream systems by two orders of magnitude. The factors in this argument include (1) tailored choices of data structures and algorithms rather than supporting a wide variety of methods, (2) much lighter software footprint and overhead, and as a result, (3) simplified optimization of system internals and easier configuration and self-adaptation to changing environments (e.g., data and workload characteristics).

RDF-3X follows such a RISC-style design philosophy [12], with "reduced instruction set" designed to support RDF. RDF-3X is based on the following three key principles:

- *Physical design* is *workload-independent* by creating appropriate indexes over a single "giant triples table". RDF-3X does not rely on the success (or limitation) of an auto-tuning wizard, but has effectively eliminated the need for physical-design tuning. It does so by building indexes over all 6 permutations of the three dimensions that constitute an RDF triple, and additionally, indexes over count-aggregated variants for all three two-dimensional and all three one-dimensional projections. Each of these indexes can be compressed very well; the total storage space for all indexes together is less than the size of the primary data.
- The *query processor* is *RISC-style* by relying mostly on merge joins over sorted index lists. This is made possible by the "exhaustive" indexing of the triples table. In fact, all processing is index-only, and the triples table exists merely virtually. Operator trees are constructed so as to preserve *interesting orders* [20] for subsequent joins to the largest possible extent; only when this is no longer possible, RDF-3X switches to hash-based join processing. This approach can be highly optimized at the code level, and has much lower overhead than traditional query processors. At the same time, it is sufficiently versatile to support also the various duplicate-elimination options of SPARQL, disjunctive patterns in queries, and all other features that SPARQL requires.
- The *query optimizer* mostly focuses on *join order* in its generation of execution plans. It employs dynamic programming for plan enumeration, with a cost model based on RDF-specific statistical synopses. These statistics include counters of frequent predicate-sequences in paths of the data graph; such paths are potential join patterns. Compared to the query optimizer in a universal database system, the RDF-3X optimizer is simpler but much more accurate in its selectivity estimations and decisions about execution plans.

Although it is reasonable to assume that most RDF databases are query-intensive if not read-only (e.g., large reference repositories in life sciences), there has to be support for at least incremental loading and ideally even online updates such as inserting a new triple for annotating existing data. The challenge in doing this efficiently is to deal with the aggressive indexing that RDF-3X employs for fast querying. We have developed a *staging architecture* that defers index updates. Updates are collected in workspaces and differential indexes, and are later merged into the main database indexes in a very efficient batched manner. This staging is transparent to programs; the query processor is appropriately extended to provide this convenience with low overhead. While this approach does not provide full-fledged ACID transactions, it includes simple but effective ways of concurrency control and recovery and supports the read-committed isolation level.

The scientific contributions of this work are

1. A novel architecture for RDF indexing and querying, eliminating the need for physical database design;
2. an optimizer for large join queries over non-schematic RDF triples, driven by a new kind of selectivity estimation for RDF paths;
3. New staging architecture for efficiently handling updates, with very good performance of batched updates;
4. A comprehensive performance evaluation, based on real measurements with three large datasets, demonstrating large gains over the previously best engine [1] (by a typical factor of 10 and up to 100 or more for some queries).

The source code of RDF-3X and the experimental data are available for non-commercial purposes from the Web site [43].

A previous version of this work has been presented in [39]. The current paper extends this prior publication in two major ways: (1) we describe the algorithms for query compilation and query optimization, which are merely sketched in [39]; (2) we present a novel way of efficiently supporting updates to RDF-3X databases, and give performance measurements for multi-user workloads.

The rest of the paper is organized as follows: Sect. 2 provides background on RDF, SPARQL, and prior work on indexing and query processing. Section 3 presents our solution for the physical design of RDF repositories. Section 4 describes the architecture and algorithms of the RDF-3X query compiler and query processor. Section 5 covers the query optimization techniques used in RDF-3X. Section 6 discusses our solutions to selectivity estimation for SPARQL expressions. Section 7 shows how to efficiently support online updates and incremental loading in RDF-3X. Section 8 presents the results of our comprehensive performance evaluation and comparison to alternative approaches.

2 Background and state of the art

2.1 SPARQL

SPARQL queries [62] are pattern matching queries on triples that constitute an RDF data graph. Syntactic sugar aside, a basic SPARQL query has the form

```
select ?variable1 ?variable2 ...
 where { pattern1. pattern2. ... }
```

where each *pattern* consists of $subject$, $predicate$, $object$, and each of these is either a variable or a literal. The query model is query-by-example style: the query specifies the known literals and leaves the unknowns as variables. Variables can occur in multiple patterns and thus imply joins. The query processor needs to find all possible variable bindings that satisfy the given patterns and return the bindings from the projection clause to the application. Note that not all variables are necessarily bound (e.g., if a variable only occurs in the projection and not in a pattern), which results in NULL values.

This pattern-matching approach restricts the freedom of the query engine with regard to possible execution plans, as shown in the following example:

```
select ?a ?c where { ?a label1 ?b. ?b label2 ?c }
```

The user is interested in all ?*a* and ?*c* that are reachable with certain edges via ?*b*. The value of ?*b* itself is not part of the projection clause. Unfortunately, the pattern-matching semantics requires that nevertheless all bindings of ?*b* need to be computed. There might be multiple ways from ?*a* to ?*c*, resulting in duplicates in the output. As this is usually not what the user/application intends, SPARQL introduced two query modifiers: the *distinct* keyword specifies that duplicates must be eliminated, and the *reduced* keyword specifies that duplicates may but need not be eliminated. The goal of the *reduced* keyword was obviously to help RDF query engines by allowing optimizations, but with the *reduced* option the query output has a nondeterministic nature.

Nevertheless, even the default mode of creating all duplicates allows some optimizations. The query processor must not ignore variables that are projected away due to their effect on duplicates, but it does not have to create the explicit bindings. As long as we can guarantee that the correct number of duplicates is produced, the bindings themselves are not relevant. We will use this observation later by counting the number of duplicates rather than producing the duplicates themselves.

2.2 Related work

Most publicly accessible RDF systems have mapped RDF triples onto relational tables (e.g., RDFSuite [2,27], Sesame [9,41], Jena [28,64], the C-Store-based RDF engine of [1], and also Oracle's RDF_MATCH implementation [13]). There are two extreme ways of doing this: (1) All triples are stored in a single, giant *triples table* with generic attributes *subject, predicate, object*. (2) Triples are grouped by their predicate name, and all triples with the same predicate name are stored in the same *property table*. The extreme form of property tables with a separate table for each predicate name can be made more flexible, leading to a hybrid approach; (3) Triples are clustered by predicate names, based on predicates for the same entity class or co-occurrence in the workload; each *cluster-property table* contains the values for a small number of correlated predicates, and there may additionally be a "left-over" table for triples with infrequent predicates. A cluster-property table has a class-specific schema with attributes named after the corresponding RDF predicates, and its width can range from a single predicate (attribute) to all predicates of the same entity type.

Early open-source systems like Jena [28,64] and Sesame [9,41] use clustered-property tables, but left the physical design to an application tuning expert. Neither of these systems has reported any performance benchmarks with large-scale data in the Gigabytes range with more than 10 million triples. Oracle [42] has reported very good performance results in [13], but seems to heavily rely on good tuning by making the right choice of materialized join views (coined subject-property matrix) in addition to its basic triples table. The previously fastest RDF engine by [1] uses minimum-width property tables (i.e., binary relations), but maps them onto a column-store system. [56] gives a nice taxonomy of different storage layouts and presents systematic performance comparisons for medium-sized synthetic data and synthetic workload.

The scalable system of [1], published in the 2007 VLDB conference, kindled great interest in RDF performance issues and new architectures. In contrast to the arguments that [1] gives against the "giant-triples-table" approach, both RDF-3X [39] and HexaStore [60] recently showed how to successfully employ a triples table with excellent performance. The work of [50] systematically studied the impact of column-stores (MonetDB) and row-stores (PostgreSQL) on different physical designs and identified strengths and weaknesses under different data and workload characteristics. The quest for scalable performance has also led to new benchmark proposals [45] and the so-called billion triples challenge [49].

The previously best performing systems, Oracle and the C-Store-based engine [1], rely on materialized join paths and indexes on these views. The indexes themselves are standard indexes as supported by the underlying RDBMS and column store, respectively. The native YARS2 system [23] proposes exhaustive indexes of triples and all their embedded sub-triples (pairs and single values) in six separate B^+-tree or hash indexes. This resembles our approach, but YARS2 misses the need for indexing triples in collation orders other than the canonical order by subject, predicate, object (as primary, secondary, and tertiary sort criterion). A very similar approach is presented for the HPRD system [7] and available as an option (coined "triple-indexes") in the Sesame system [41]. Both YARS2 and HPRD seem to be primarily geared for simple lookup operations with limited support for joins; they lack DBMS-style query optimization (e.g., do not consider any join-order optimizations, although [7] recognizes the issue). Baolin and Bo [6] proposes to index entire join

paths using suffix arrays, but does not discuss optimizing queries over this physical design. Udrea [57] introduces a new kind of path indexing based on judiciously chosen "center nodes"; this index, coined GRIN, shows good performance on small- to medium-sized data and for hand-specified execution plans. Physical design for schema-agnostic "wide and sparse tables" is also discussed in [14], without specific consideration to RDF. All these methods for RDF indexing and materialized views incur some form of physical design problem, and none of them addresses the resulting query optimization issues over these physical-design features.

As for query optimization, [1,13,42] utilize the state-of-the-art techniques that come with the SQL engines on which these solutions are layered. To our knowledge, none of them employs any RDF-native optimizations. Stocker et al. [53] outlines a framework for algebraic rewriting, but it seems that the main rule for performance gains is pushing selections below joins (i.e., applying selections as early as possible); there is no consideration of join ordering. Hartig and Heese [24] has a similar flavor, and likewise disregards the key problem of finding good join orderings.

Recently, selectivity estimation for SPARQL patterns over graphs have been addressed by [33,53]. The method by [53] gathers separate frequency statistics for each subject, each predicate, and each object (label or value); the frequency of an entire triple pattern is estimated by assuming that subject, predicate, and object distributions are probabilistically independent. The method by [33,34] is much more sophisticated by building statistics over a selected set of arbitrarily shaped graph patterns. It casts the selection of patterns into an optimization problem and uses greedy heuristics. The cardinality estimation of a query pattern identifies maximal subpatterns for which statistics exist, and combines them with uniformity assumptions about super-patterns without statistics. While [53] seems to be too simple for producing accurate estimates, the method by [33] is based on a complex optimization problem and relies on simple heuristics to select a good set of patterns for the summary construction. The method that we employ in RDF-3X captures path-label frequencies, thus going beyond [53] but avoiding the computational complexity of [33].

3 Storage and indexing

3.1 Triples store and dictionary

Although most of the prior, and especially the recent, literature favors a storage schema with property tables, we decided to pursue the conceptually simpler approach with a single, potentially huge *triples table*, with our own storage implementation underneath (as opposed to using an RDBMS). This reflects our RISC-style and "no-knobs" design rationale. We overcome the previous criticism that a triples table incurs too many expensive self-joins by creating the "right" set of indexes (see below) and by very fast processing of merge joins (see Sect. 4).

We store all triples in a (compressed) clustered B^+-tree. The triples are sorted lexicographically in the B^+-tree, which allows the conversion of SPARQL patterns into range scans. In the pattern (literal1,literal2,?x) the literals specify the common prefix and thus effectively a range scan. Each possible binding of ?x is found during a single scan over a moderate number of leaf pages.

As triples may contain long string literals, we adopt the natural approach (see, e.g., [13]) of replacing all literals by ids using a *mapping dictionary*. This has two benefits: (1) it compresses the triple store, now containing only id triples, and (2) it is a great simplification for the query processor, allowing for fast, simple, RISC-style operators (see Sect. 4.4). The small cost for these gains is two additional *dictionary indexes*. During query translation, the literals occurring in the query are translated into their dictionary ids, which can be done with a standard B^+-tree from strings to ids. After processing the query the resulting ids have to be transformed back into literals as output to the application/user. We could have used a B^+-tree for this direction, too, but instead we implemented a direct mapping index [17]. Direct mapping is tuned for id lookups and results in a better cache hit ratio, as it is more compact than a B^+-tree and accesses only two pages per lookup. Note that this is only an issue when the query produces many results. Usually the prior steps (joins etc.) dominate the costs, but for simple queries with many results dictionary lookups are non-negligible.

3.2 Compressed indexes

In the index-range-scan example given above we rely on the fact that the variables are a suffix (i.e., the *object* or the *predicate* and *object*). To guarantee that we can answer every possible pattern with variables in any position of the pattern triple by merely performing a single index scan, we maintain all six possible permutations of *subject (S)*, *predicate (P)* and *object (O)* in six separate indexes. We can afford this level of redundancy because we compress the id triples (discussed below). On all our experimental datasets, the total size for all indexes together is less than the original data.

As the collation order in each of the six indexes is different (SPO, SOP, OSP, OPS, PSO, POS), we use the generic terminology $value_1$, $value_2$, $value_3$ instead of *subject*, *predicate*, *object* for referring to the different columns. The triples in an index are sorted lexicographically by ($value_1$, $value_2$, $value_3$) (for each of the six different permutations) and are directly stored in the leaf pages of the clustered B^+-tree.

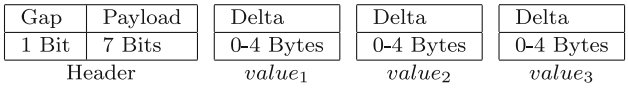

Fig. 1 Structure of a compressed triple

```
compress((v1, v2, v3),(prev1, prev2, prev3))
// writes (v1, v2, v3) relative to (prev1, prev2, prev3)
if v1 = prev1 ∧ v2 = prev2
   if v3 − prev3 < 128
      write v3 − prev3
   else encode(0,0,v3 − prev3 − 128)
else if v1 = prev1
   encode(0,v2 − prev2,v3)
else
   encode(v1 − prev1,v2,v3)

encode(δ1, δ2, δ3)
// writes the compressed tuple corresponding to the deltas
write 128+bytes(δ1)*25+bytes(δ2)*5+bytes(δ3)
write the non-zero tail bytes of δ1
write the non-zero tail bytes of δ2
write the non-zero tail bytes of δ3
```

Fig. 2 Pseudo-code of triple compression

The collation order causes neighboring triples to be very similar: most neighboring triples have the same values in $value_1$ and $value_2$, and the increases in $value_3$ tend to be very small. This observation naturally leads to a compression scheme for triples. Instead of storing full triples we only store the changes between triples. This compression scheme is inspired by methods for inverted lists in text retrieval systems [68], but we generalize it to id triples rather than simple ids. For reasons discussed below, we apply the compression only within individual leaf pages and never across pages.

For the compression scheme itself, there is a clear trade-off between space savings and CPU consumption for decompression or interpretation of compressed items [63]. We noticed that CPU time starts to become an issue when compressing too aggressively, and therefore settled for a byte-level (rather than bit-level) compression scheme. We compute the delta for each value, and then use the minimum number of bytes to encode just the delta. A header byte denotes the number of bytes used by the following values (Fig. 1). Each value consumes between 0 bytes (unchanged) and 4 bytes (delta needs the full 4 bytes), which means that we have 5 possible sizes per value. For three values these are $5 * 5 * 5 = 125$ different size combinations, which fits into the payload of the header byte. The remaining *gap* bit is used to indicate a small gap: When only $value_3$ changes, and the delta is less than 128, it can be directly included in the payload of the header byte. This kind of small delta is very common, and can be encoded by a single byte in our scheme.

The details of the compression are shown in Fig. 2. The algorithm computes the delta to the previous tuple. If it is small it is directly encoded in the header byte; otherwise, it computes the δ_i values for each of the tree values and calls *encode*. *encode* writes the header byte with the size information and then writes the non-zero tail of the δ_i (i.e., it writes δ_i byte-wise but skips leading zero bytes). This results in compressed tuples with varying sizes, but during decompression the sizes can be reconstructed easily from the header byte. As all operations are byte-wise, decompression involves only a few cheap operations and is very fast.

We tested the compression rate and the decompression time (in seconds) of our byte-wise compression against a number of bit-wise compression schemes proposed in the literature [46]. The results for the Barton dataset (see Sect. 8) are shown in Table 1. Our byte-wise scheme compresses nearly as good as the best bit-wise compression scheme, while providing much better decompression speed. The Gamma and Golomb compression methods, which are popular for inverted lists in IR systems, performed worse because, in our setting, gaps can be large whenever there is a change in the triple prefix.

We also experimented with the more powerful LZ77 compression on top of our compression scheme. Interestingly, our compression scheme compresses better with LZ77 than the original data, as the delta encoding exhibits common patterns in the triples. The additional LZ77 compression decreases the index size roughly by a factor of two, but increases CPU time significantly, which would become critical for complex queries. Thus, the RDF-3X engine does not employ LZ77.

An important consideration for the compressed index is that each leaf page is compressed individually. Compressing larger chunks of data leads to better compression (in particular in combination with the LZ77 compression), but page-wise compression has several advantages. First, it allows us to seek (via B^+-tree traversal) to any leaf page and directly start

Table 1 Comparison of byte-wise compression versus bit-wise compression for the Barton dataset

Barton data	Byte-wise	Bit-wise		
	RDF-3X	Gamma	Delta	Golomb
6 indexes (GBytes)	1.10	1.21	1.06	2.03
Decompression (s)	3.2	44.7	42.5	82.6

reading triples. If we compressed larger chunks we would often have to decompress preceding pages. Second, the compressed index behaves just like a normal B^+-tree (with a special leaf encoding). Thus, updates can be easily done like in a standard B^+-tree. This greatly simplifies the integration of the compressed index into the rest of the engine and preserves its RISC nature. In particular, we can adopt advanced concurrency control and recovery methods for index management without any changes.

3.3 Aggregated indices

For many SPARQL patterns, indexing partial triples rather than full triples would be sufficient, as demonstrated by the following SPARQL query:

```
select ?a ?c where { ?a ?b ?c }
```

It computes all ?*a* and ?*c* that are connected through any predicate, the actual bindings of ?*b* are not relevant. We therefore additionally build *aggregated indexes*, each of which stores only two out of the three columns of a triple. More precisely, they store two entries (e.g., *subject* and *object*), and an aggregated *count*, namely, the number of occurrences of this pair in the full set of triples. This is done for each of the three possible pairs out of a triple and in each collation order (SP, PS, SO, OS, PO, OP), thus adding another six indexes.

The count is necessary because of the SPARQL semantics. The bindings of ?*b* do not occur in the output, but the right number of duplicates needs to be produced. Note that the aggregated indexes are much smaller than the full-triple indexes; the increase of the total database size caused by the six additional indexes is negligible.

Instead of $(value_1, value_2, value_3)$, the aggregated indexes store $(value_1, value_2, count)$, but otherwise they are organized in B^+-trees just like the full-triple compressed indexes. The leaf encoding is slightly different, as now most changes involve a gap in $value_2$ and a low *count* value. The pseudo-code is shown in Fig. 3.

Finally, in addition to these indexes for pairs in triples, we also build all three one-value indexes containing just $(value_1, count)$ entries (the encoding is analogous). While

$compressAggregated((v_1, v_2), count, (prev_1, prev_2))$
// writes $(v_1, v_2) * count$ relative to $(prev_1, prev_2)$
if $v_1 = prev_1$
  if $v_2 - prev_2 < 32 \wedge count < 5$
    write $(count - 1) * 32 + (v_2 - prev_2)$
  else
    $encode(0, v_2 - prev_2, count)$
else
  $encode(v_1 - prev_1, v_2, count)$

Fig. 3 Aggregated triple compression

triple patterns using only one variable are probably rare, the one-value indexes are very small, and having them available simplifies query translation.

4 Query translation and processing

4.1 Translating SPARQL queries

The first step in compiling a SPARQL query is to transform it into a calculus representation suitable for later optimizations. We construct a query graph representation that can be interpreted as relational tuple calculus. It would be simpler to derive domain calculus from SPARQL, but tuple calculus is better suited for the query optimizer.

The translation of SPARQL queries is illustrated by an example in Fig. 4. While SPARQL allows many syntax shortcuts to simplify query formulation (Fig. 4a), each (conjunctive) query can be parsed and expanded into a set of triple patterns (Fig. 4b). Each component of a triple is either a literal or a variable. The parser already performs dictionary lookups, i.e., the literals are mapped into ids. Similar to domain calculus, SPARQL specifies that variable bindings must be produced for every occurrence of the resulting literals-only triple in the data. When a query consists of a single triple pattern, we can use our index structures from Sect. 3 and answer the query with a single range scan. When a query consist of multiple triple patterns, we must join the results of the individual patterns (Fig. 4d). We thus employ join ordering algorithms on the query graph (Fig. 4c) representation, as discussed in Sect. 5.

Each triple pattern corresponds to one node in the query graph. Conceptually, each node entails a scan of the whole database with appropriate variable bindings and selections induced by the literals. While each of these scans could be implemented as a single index range scan, the optimizer might choose a different strategy (see below). The edges in the query graph reflect joint variable occurrences: two nodes are connected if and only if they have a (query) variable in common.

Using the query graph, we can construct an (unoptimized) execution plan as follows:

1. Create an index scan for each triple pattern. The literals in the pattern determine the range of the scan.
2. Add a join for each edge in the query graph. If the join is not possible (i.e., if the triple patterns are already joined via other edges) add a selection.
3. If the query graph is disconnected, add cross-products as necessary to obtain a single join tree.
4. Add a selection containing all *FILTER* predicates.

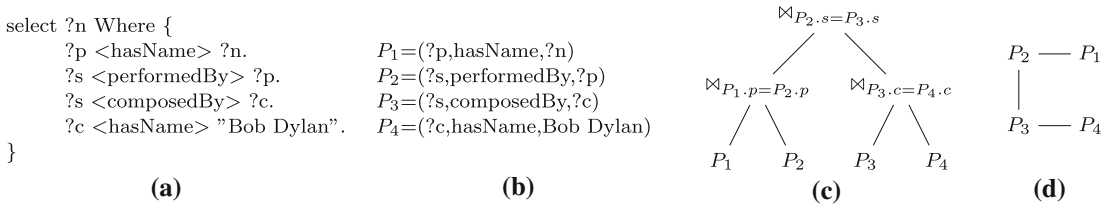

Fig. 4 SPARQL translation example. **a** SPARQL, **b** Triples form, **c** Possible join tree, **d** Query graph

5. If the projection clause of a query includes the *distinct* option, add an aggregation operator that eliminates duplicates in the result.
6. Finally add a dictionary lookup operator that converts the resulting ids back into strings.

This gives us a "canonical" execution plan for any conjunctive SPARQL query, i.e., a plan that is valid but potentially inefficient. In the RDF-3X system the steps 1 through 4 are actually performed by the query optimizer (see Sect. 5), which finds the optimal scan strategy and the optimal join order for a given query.

4.2 Handling disjunctive queries

While conjunctive queries are more commonly used, SPARQL also allows certain forms of disjunctions. The *UNION* expression returns the union of the bindings produced by two or more pattern groups. The *OPTIONAL* expressions returns the bindings of a pattern group if there are any results, and NULL values otherwise. In this context, pattern groups are sets of triple patterns, potentially containing *UNION* and *OPTIONAL* expressions themselves.

During query translation and optimization, we treat pattern groups in *UNION* and *OPTIONAL* as nested subqueries. That is, we translate and optimize the nested pattern groups first (potentially recursively) and then treat them as base relations with special costs and cardinalities during the translation and optimization of the outer query. For *UNION* we add the union of the results of the pattern groups as if it were a base relation, for *OPTIONAL* we add the result as a base relation using an outer join.

In principle, it would be possible to optimize these queries more aggressively, but most interesting optimizations require the usage of bypass plans [52] or other non tree-structured execution plans, which is beyond the scope of this work. And these optimizations would only pay off for complex queries; when the disjunctive elements are simple, our nested translation and optimization scheme produces the optimal solution.

4.3 Preserving result cardinality

The standard SPARQL semantic requires that the right number of variable bindings are produced, even if many of them are duplicates. However, from a processing point of view, one should avoid the additional work for producing and keeping duplicates.

We solve this issue by tracking the multiplicity of each tuple during query processing. Scans over unaggregated indexes always produce a multiplicity of 1, while aggregated indexes report the number of duplicates as multiplicity. Join operators multiply the multiplicities to get the number of duplicates of each output tuple. Note that we can optionally switch off the multiplicity tracking if we can statically derive that it has to be 1 in a subplan. When the result is presented to the application/user, the output operator interprets the multiplicities according to the specified query semantics (*distinct*, *reduced*, or standard).

4.4 Implementation issues

Our run-time system includes the typical set of algebraic operators (merge-join, hash-join, filter, aggregation, etc.). One notable difference to other systems is that our run-time system is very RISC-style: most operators merely process integer-encoded ids, consume and produce streams of id tuples, compare ids, etc. Besides simplifying the code, this reduced complexity allows neat implementation tricks.

For example, consider an index-scan operator that uses a B^+-tree iterator to access the physical data, comparing a triple pattern against the data. Each entry in the triple is either an id attribute that must be produced or bound to a literal, which affects start/stop condition if it is in the prefix or implies a selection if unbound entries come first. Instead of checking for these different conditions at run-time, we can handle them at query compilation time. Each entry is either an id attribute or a literal. There are three entries in a triple, which means there are eight possible combinations. With a single method interface that has eight different implementations of the index scan operator, we can greatly reduce the number of conditional branches in the system code. Besides being faster, each specialized operator is much simpler as it now implements just the logic required for its setting. Note that we only need to specialize the step logic, which is less than ten lines of code for each specialization.

This RISC-style combination of simplified type system and simple, fast operators leads to very good CPU

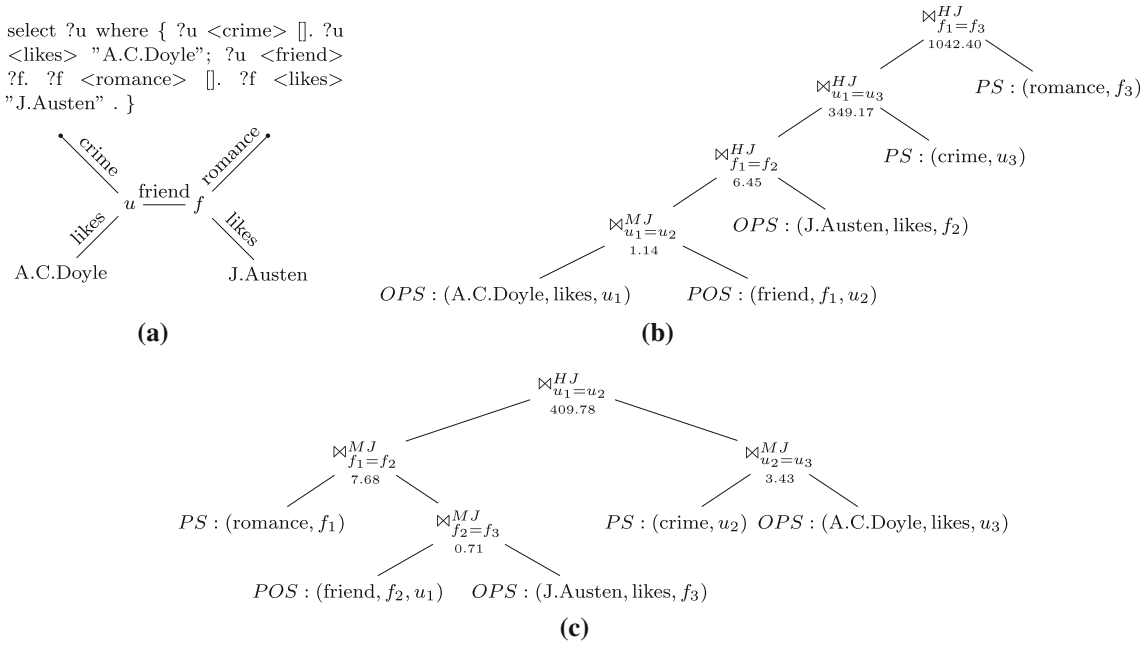

Fig. 5 Execution plans with estimated CPU costs for LibraryThing query B1. **a** Original query, **b** Plan constructed by minimum-selectivity-ordering, **c** Fully optimized plan

performance. In our evaluation in Sect. 8 we include warm-cache times to demonstrate these effects. We realize that these kinds of code-tuning issues are often underappreciated, but are crucial for high performance on modern hardware.

5 Query optimization

5.1 Requirements and example plans

The key issue for optimizing SPARQL execution plans is join ordering. There is a rich body of literature on this problem, with solutions typically based on various forms of dynamic programming (DP) or randomization (e.g., [15,18,35,48]). However, the intrinsic characteristics of RDF and SPARQL create join queries with particularly demanding properties, which are not directly addressed by prior work:

1. SPARQL queries tend to contain multiple star-shaped subqueries, for combining several attribute-like properties of the same entity, that are joined together. Thus, it is essential to use a strategy that can create bushy join trees (rather than focusing on left-deep or right-deep trees).
2. These star joins occur at various nodes of long join paths, often at the start and end of a path. SPARQL queries can easily lead to ten or more joins between triples (see, for example, our benchmark queries in Sect. 8). So, exact optimization either requires very fast plan enumeration and cost estimation or needs to resort to heuristic approximations.
3. We would like to leverage the particular strengths of our triple indexes, which encourage extensive use of merge joins (rather than hash or nested-loop joins), but this entails being very careful about preserving interesting orders in the generation of join plans.

The first requirement rules out methods that cannot generate all possible star-chain combinations. The second requirement strongly suggests a fast bottom-up method rather than transformation-based top-down enumeration. The third requirement rules out sampling-based plan enumeration (or randomized optimization methods), as these are unlikely to generate all order-preserving plans for queries with more than 10 joins. In fact, we expect that the most competitive execution plans have a particular form: they would use order-preserving merge-joins as long as possible and switch to hash-joins only for the last few operators.

These requirements are illustrated in Fig. 5, which shows the (slightly simplified) query B1 from Sect. 8.4, its query graph (a), and two possible execution plans (b and c). The query consists of two small stars centered around the entities u and f which are joined together. The two operator trees show the estimated CPU costs (in the cost units of our estimator) for each node, using the actual (i.e., measured) cardinalities of intermediate results.

A seemingly natural way of constructing a suitable execution plan would be based on a greedy minimum-selectivity heuristics: pick joins in ascending order of selectivity (while avoiding Cartesian products). This leads to the left-deep operator tree shown in Fig. 5b. This execution plan

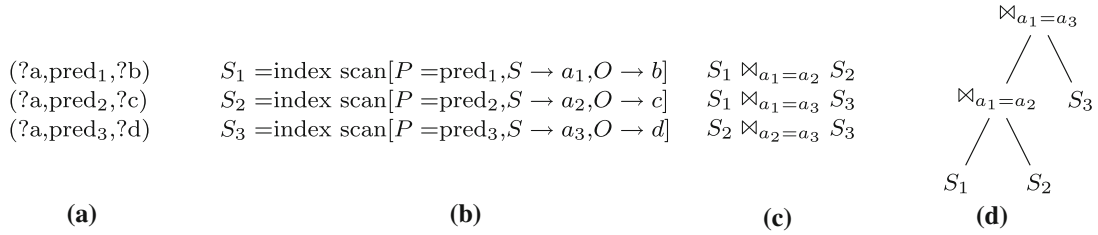

Fig. 6 Example for reasoning using attribute equivalence classes. **a** Triple patterns, **b** Scan operators, **c** Possible joins, **d** Possible join tree

includes fairly expensive hash-table build and probe operations. In contrast, the most efficient way to execute this query (using the RDF-3X operators) is to join each star individually using merge-joins, and then joining the two stars together using a hash-join, resulting in a bushy execution plan (Fig. 5c). The patterns in a star are semantically related and are selective only in combination, which mandates the bushy plans. Furthermore, the triple patterns within each star join on the same subject, which makes the merge-joins extremely efficient. This greatly reduces the CPU overhead compared to hash-joins. During an actual execution the greedy plan causes CPU costs of 3,325 hash-table operations, while the optimal plan requires just 258 hash-table operations and the costs of merge-joins are negligible relative to those of the hash-table operations. The actually measured warm-cache run-times were 1 and 2 ms, respectively. This query structure with multiple stars (connected by other relationships) is very common in SPARQL queries.

5.2 DP-based plan enumeration

Our solution to finding optimal (or near-optimal) execution plans is based on the *bottom-up dynamic-programming* (DP) framework of [35]. It organizes a DP table by subgraphs of the query graph, maintaining for each subgraph the optimal plan and the resulting output order. If there are multiple plans for a subgraph none of which dominates the other, all of them are kept in the DP table. This happens when good plans produce different output orders.

The optimizer seeds its DP table with scans for the base relations, in our case the triple patterns. The seeding is a two-step process. First, the optimizer analyzes the query to check which variable bindings are used in other parts of the query. If a variable is unused, it can be projected away by using an aggregated index (see Sect. 3.3). Note that this projection conceptually preserves the cardinality through the count information in the index (see Subsect. 4.3). In the second step the optimizer decides which of the applicable indexes to use. There are two factors that affect the index selection. When the literals in the triple pattern form the prefix of the index key, they are automatically handled by the range scan. Otherwise, too many tuples are read and additional selections are required. On the other hand, a different index might produce the triples in an order suitable for a subsequent merge-join later on, which may have lower overall costs than reading too many tuples. The optimizer therefore generates plans for all indexes and uses the plan pruning mechanism to decide if some of them can be discarded early on.

Pruning the search space of possible execution plans is based on estimated execution costs. The optimizer calls the cost model for each generated plan and prunes equivalent plans that are dominated by cheaper alternatives. This pruning mechanism relies on order optimization [51] to decide if a plan is dominated by another plan. As the optimizer can use indexes on all triple permutations, it can produce tuples in an arbitrary order, which makes merge joins very attractive. Thus, some plans are kept even if they are more expensive but produce an interesting ordering that can be used later on. Note that orderings are not only created by index scans but also by functional dependencies induced by selections; therefore, the order optimization component is non-trivial [51]. We utilize the techniques of [38] for this purpose.

Starting with the seeds, larger plans are created by joining optimal solutions of smaller problems. During this process, all attribute-handling logic for comparing values is implemented as reasoning over equivalence classes of variables instead of individual variable bindings. Variables that appear in different triple patterns are equivalent if they must have the same bindings to values in the final result of the query. These equivalence classes are determined at compile-time. It is sufficient that an execution plan produces at most one binding for each equivalence class (and no bindings for variables that do not appear in the query output). This simplifies implicit projections that precede pipeline breakers (e.g., hash-table build for a hash-join) and also allows for automatic detection of transitive join conditions (e.g., $a = b \wedge b = c \Rightarrow a = c$).

An example for this reasoning is shown in Fig. 6. It shows a query with three triple patterns (Fig. 6a) and joins on the common attribute a between the patterns. The first translation step creates index scans for the three patterns (Fig. 6b), producing attributes a_1, b, a_2 ,c, a_3, and d. Note that the a_i are really different from each other in terms of their value bindings; we named them similarly to show the connection to the original query variable. We can derive three join operators from the query graph (Fig. 6c), but in the final join tree (Fig. 6d) we use only two of them. The canonical translation

Table 2 Optimization times for queries with x patterns in two stars connected by a path of length y

	6-triples Star patterns (ms)	8-triples Star patterns (ms)	10-triples Star patterns (ms)
Path length 5	0.7	2.4	6.4
Path length 10	2.0	4.6	17.3
Path length 20	3.5	13.0	56.6

would add another $\sigma_{a_2=a_3}$, but we can deduce that a_1 and a_2 are in the same equivalence class, thus the $\bowtie_{a_1=a_3}$ is sufficient. Note that equivalence classes are derived from equality conditions, not only from join-variable names. We could perform the same reasoning if the three patterns had no variable in common but instead joined via *FILTER* conditions.

Starting with the index scans seeds, larger plans are created by joining optimal solutions of smaller problems that are adjacent in the query graph [35]. When the query contains additional selections due to *FILTER* predicates, they are placed greedily as soon as possible, as they are usually inexpensive to evaluate. If a selection is really expensive, it could be better to integrate it into the DP operator placement as proposed in [11], but we did not investigate this further. The DP method that we implemented along these lines is very fast and is able to compute the exact cost-optimal solution for join queries with up to 20 triple patterns. We measured optimization times (in milliseconds) for a typical SPARQL scenario with two entities selected by star-shaped subqueries and connected by a chain of join patterns. The results are shown in Table 2. Note that the resulting plans are always optimal relative to the estimates by the cost model.

6 Selectivity estimates

The query optimizer relies upon its cost model in finding the lowest-cost execution plan. In particular, estimated cardinalities (and thus selectivities) have a huge impact on plan generation. While this is a standard problem in database systems, the schema-free nature of RDF data complicates statistics generation. We propose two kinds of statistics. The first one, specialized histograms, is generic and can handle any kind of triple patterns and joins. Its disadvantage is that it assumes independence between predicates, which frequently does not hold in tightly coupled triple patterns. The second statistics therefore computes frequent join paths in the data, and gives more accurate predictions on these paths for large joins. During query optimization, we use the join-path cardinalities when available and otherwise assume independence and use the histograms.

6.1 Selectivity histograms

While triples conceptually form a single table with three columns, histograms over the individual columns are not very useful as most query patterns touch at least two attributes of a triple. Instead, we harness our aggregated indexes, which are perfectly suited for the calculation of triple-pattern selectivities: for each literal or literal pair, we can get the exact number of matching triples with one index lookup. Unfortunately, this is not sufficient for estimating join selectivities. Also, we would like to keep all auxiliary structures for the cost model in main memory. Therefore, we aggregate the indexes even further such that each index fits into a single database page and includes information about join selectivity.

Just like the aggregated indexes we build six different statistics, one for each order of the entries in the triples. Starting from the aggregated indexes, we place all triples with the same prefix of length two in one bucket and then merge the smallest two neighboring buckets until the total histogram is small enough. This approximates an equi-depth histogram, but avoids placing bucket boundaries between triples with the same prefix (which are expected to be similar).

For each bucket we then compute the statistics shown in Fig. 7. The first three values—the number of triples, number of distinct 2-prefixes, and number of distinct 1-prefixes—are used to estimate the cardinality of a single triple pattern. Note that this only gives the scan cardinality, i.e., the number of scanned triples, which determines the costs of an index scan. The true result cardinality, which affects subsequent operators, could actually be lower when literals are not part of the index prefix and are tested by selections later on. In this case, we derive the result cardinality (and obtain exact predictions) by reordering the literals such that all literals are in the prefix.

start (s,p,o)		end (s,p,o)	
number of triples			
number of distinct 2-prefixes			
number of distinct 1-prefixes			
join partners on subject			
	s=s	s=p	s=o
join partners on predicate			
	p=s	p=p	p=o
join partners on object			
	o=s	o=p	o=o

Fig. 7 Structure of a histogram bucket

The next values are the numbers of join partners (i.e., the result cardinality) if the triples in the bucket were joined to all other triples in the database according to the specified join condition. As there are nine ways to combine attributes from two triples, we precompute nine cardinalities. For example, the entry o=s is effectively

$$|\{b|b \in \text{current bucket}\} \bowtie_{b.\text{object}=t.\text{subject}} \{t|t \in \text{all triples}\}|.$$

These values give a perfect join-size prediction when joining a pattern that exactly matches the bucket with a pattern without literals. Usually, this is not the case; we therefore assume independence between query conditions and multiply the selectivities of the involved predicates. Such independence assumptions are standard in state-of-the-art query optimizers for tractability. Note that these values can be computed efficiently by two merge-joins with the one-value indexes (S, P, O), as these directly return the number of triples with a certain subject, predicate, or object.

6.2 Frequent paths

The histograms discussed above have decent accuracy, and are applicable for all kinds of predicates. Their main weakness is that they assume independence between predicates. Two kinds of correlated predicates commonly occur in SPARQL queries. First, *"stars"* of triple patterns, where a number of triple patterns with different predicates share the same subject. These are used to select specific subjects (i.e., entities based on different attributes of the same entities). Second, *"chains"* of triple patterns, where the object of the first pattern is the subject of the next pattern, again with given predicates. These chains correspond to long join paths (across different entities). As both of these two cases are common, we additionally build specialized statistics to have more accurate estimators for such queries.

To this end, we precompute the frequent paths in the data graph and keep exact join statistics for them. Frequency here refers to paths with the same label sequence. Note that we use the term path both for chains and stars as the constructions are similar in the two cases. We characterize a path P by the sequence of predicates $p_1, \ldots, p_n$ seen in its traversal. Using SPARQL syntax, we define a (chain) path $P_{p_1,\ldots,p_n}$ as

$$P_{p_1,\ldots,p_n} := \text{select } r_1 r_{n+1} \text{ where } \{(r_1\, p_1 r_2).\ (r_2\, p_2 r_3). \ldots (r_n\, p_n r_{n+1})\}$$

Star paths are defined analogous; the $p_1, \ldots, p_n$ are unsorted in this case. We compute the most frequent paths, i.e., the paths with the largest cardinalities, and materialize their result cardinalities and path descriptions $p_1, \ldots, p_n$. Using this information we can exactly predict the join cardinality for the frequent paths that occur in a query. Again, we want to keep these statistics in main memory and therefore compute the most frequent paths such that they still fit on a single database page. In our experiments we could store about 1,000 paths on one 16 KB page.

```
FrequentPath(k)
// Computes the k most frequent paths
C_1 = {P_p | p is a predicate in the database}
sort C_1, keep the k most frequent
C = C_1, i = 1
do
  C_{i+1} = ∅
  for each p' ∈ C_i, p predicate in the database
    if top k of C ∪ C_{i+1} ∪ {P_{p'p}} includes all subpaths of p'p
      C_{i+1} = C_{i+1} ∪ {P_{p'p}}
    if top k of C ∪ C_{i+1} ∪ {P_{pp'}} includes all subpaths of pp'
      C_{i+1} = C_{i+1} ∪ {P_{pp'}}
  C = C ∪ C_{i+1}, sort C, keep the k most frequent
  C_{i+1} = C_{i+1} ∩ C, i = i + 1
while C_i ≠ ∅
return C
```

Fig. 8 Frequent path mining algorithm

Finding the most frequent paths requires some care. While it may seem that this is a standard graph-mining issue, the prevalent methods in that line of research [19,59,67], e.g., based on the well-known Apriori frequent-itemset mining algorithm, are not directly applicable.

Unlike the Apriori setting, a frequent path in our RDF-path sense does not necessarily consist of frequent subpaths. Consider a graph with two star-shaped link clusters where all end-nodes are connected to their respective star centers by predicates (edge labels) p_1 and p_2, respectively. Now consider a single edge with predicate p_3 between the two star centers. In this scenario, the path P_{p_3} will be infrequent, while the path P_{p_1,p_3,p_2} will be frequent. Therefore, we cannot simply use the Apriori algorithm.

Another problem in our RDF setting are cycles, which could lead to seemingly infinitely long, infinitely frequent paths. We solve this problem by two means. First, we require that if a frequent path P is to be kept, all of its subpaths have to be kept, too. This is required for query optimization purposes anyway, as we may have to break a long join-path into smaller joins, and it simplifies the frequent-path computation. Second, we rank the frequent paths not by their result cardinalities but by their number of distinct nodes. In a tree these two are identical, but in the presence of cycles we do not count nodes twice. This approach is similar in spirit to computing maximal patterns in data mining [5].

The pseudo-code of the path mining algorithm is shown in Fig. 8. It starts from frequent paths of length one and enlarges them by appending or prepending predicates. When a new path is itself frequent and all of its subpaths are still kept, we add it. We stop when no new paths can be added. Note that, although the pseudo-code shows a nested loop for ease of presentation, we actually use a join and a group-by operator in the implementation. For the datasets we considered

in our experiments the 1,000 most frequent paths could be determined in a few minutes.

6.3 Estimates for composite queries

For estimating the overall selectivity of an entire composite query, we combine the histograms with the frequent paths statistics. A long join chain with intermediate nodes that have triple patterns with object literals is decomposed into subchains of maximal lengths such that only their end nodes have triple patterns with literals. For example, a query like

$?x_1\ a_1\ v_1.\ ?x_1\ p_1\ ?x_2.\ ?x_2\ p_2\ ?x_3.\ ?x_3\ p_3\ ?x_4.$
$?x_4\ a_4\ v_4.\ ?x_4\ p_4\ ?x_5.\ ?x_5\ p_5\ ?x_6.\ ?x_6\ a_6\ v_6$

with attribute-flavored predicates a_1, a_4, a_6, literals v_1, v_4, v_6, and relationship-flavored predicates p_1 through p_5 will be broken down into the subchains for p_1-p_2-p_3 and for p_4-p_5 and the per-subject selections a_1-v_1, a_4-v_4, and a_6-v_6. We use the frequent paths statistics to estimate the selectivity of the two join subchains, and the histograms for selections. Then, in the absence of any other statistics, we assume that the different estimators are probabilistically independent, leading to a product formula with the per-subchain and per-selection estimates as factors. If instead of a simple attribute-value selection like $?x_6\ a_6\ v_6$ we had a star pattern such as $?x_6\ a_6\ u_6.\ ?x_6\ b_6\ v_6.\ ?x_6\ c_6\ w_6$ with properties a_6, b_6, c_6 and corresponding object literals u_6, v_6, w_6, we would first invoke the estimator for the star pattern, using the frequent paths statistics for stars, and then combine them with the other estimates in the product form.

6.4 Estimation accuracy

To assess the accuracy of our estimation techniques, we computed the estimates for the benchmark queries that we will use in Sect. 8 for comprehensive performance evaluation. We compare three techniques: a) a baseline which uses standard single-dimensional histograms on each of the three attributes S, P, and O, b) our notion of RDF-specific selectivity histograms developed in Subsect. 6.1 but without any frequent-paths information, and c) the full-fledged combination of the new type of histograms and the frequent-paths statistics of Subsect. 6.2.

Table 3 gives the relative estimation errors $\frac{|\text{actual}-\text{estimated}|}{\text{actual}}$ of pattern cardinalities, averaged over all queries in each of the three query benchmarks. The queries over the Barton data set are relatively simple, and accordingly the selectivity estimates are very accurate here. The single-dimensional histograms give the best accuracy on this dataset as they require less space per bucket and thus have more fine-grained buckets, compared to the RDF histograms. However, our techniques perform only slightly worse, still yielding very good accuracy. This is very different for the other two datasets, where both the data and the queries are more complex. In these two cases, the single-dimensional histograms perform very poorly, while our RDF synopses are much more accurate. The frequent-paths statistics further improve accuracy, in particular for the Librarything dataset.

7 Managing updates

So far, we have solely considered static RDF databases that are bulk-loaded once and then repeatedly queried. The current RDF and SPARQ standards [61,62] do not discuss any update operations, but an RDF database should support data changes and incremental loading as well.

In designing the management of updates for RDF-3X, we make the following assumptions about typical usage patterns:

1. Queries are far more frequent than updates, and also much more resource consuming.
2. Updates are mostly insertions of new triples. Overwriting existing triples (updates-in-place) are rarely needed; rather new versions and annotations would be created.
3. Updates can usually be batched, boiling down to incremental loading.
4. It is desirable that (batched) updates can be performed concurrently with queries, but there is no need for full-fledged ACID transactions.

Based on these assumptions, we have designed and implemented a *differential indexing* method for RDF-3X that supports both individual update operations and entire batches. The key idea is that updates are initially applied to small differential indexes, deferring the actual updates to the main indexes. The differential indexes can be easily integrated into the query processor, and are thus transparent to application programs. Periodically, or after reaching a certain size, differential indexes are merged into the main indexes in a batched manner. This overall architecture for managing updates in RDF-3X is illustrated in Fig. 9.

Table 3 Average relative estimation errors for query patterns from Sect. 8

	Barton	Yago	Librarything
Single-dimensional histograms	0.44	2.18	15.74
RDF selectivity histograms	0.73	0.86	0.88
+ Frequent paths	0.57	0.78	0.54

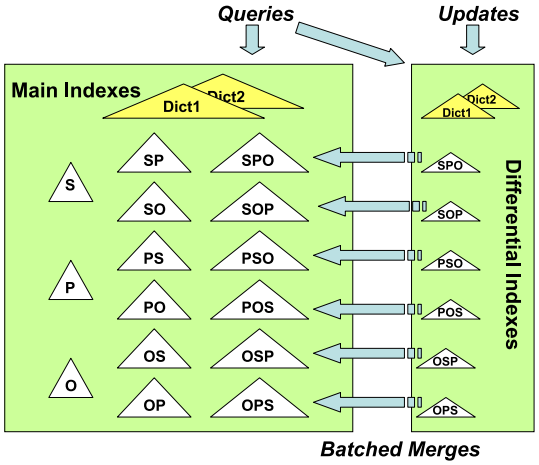

Fig. 9 Differential indexes in RDF-3X

In Subsects. 7.1 and 7.2 we first discuss the insertion of new triples, introducing our additional data structures, and the deletion of existing triples. Updates to existing triples, if ever needed, must be expressed as a pairs of deletions and insertions. Although full ACID transactions are beyond our current scope, we do provide rudimentary support for atomic batches of operations. This will be discussed in Subsect. 7.3.

7.1 Insert operations

Inserting new triples into a database is a performance challenge because of the aggressive indexing on which RDF-3X is based. In principle, the indexes can be easily updated as they are standard B$^+$-trees, but each new triple would incur access to 15 indexes. Furthermore, the updates are computationally more expensive than usual due to the compressed leaf pages. This makes direct updates to indexes unattractive.

To avoid these costs, we use a *staging architecture* with deferred index updates. All updates are first collected in *workspaces*, and all their triples are indexed separately from the main database in *differential indexes*. Periodically, or when the size of the differential indexes exceeds a specified limit, all staged changes are integrated into the main indexes described in Sect. 3. During query processing, the query compiler adds additional merge joins with the, usually very small, differential indexes to transparently integrate deferred updates if they satisfy query conditions. The principles of this architecture resemble the rationale of the multi-component LSM-tree [40] and similar data structures with batched updates such as [16,37,29,47]. In the following, we present more details on (1) workspace management, (2) differential indexes, and (3) extended query processing.

Workspace management. Each application program that interacts with the RDF-3X engine has a private workspace in memory. We expect that the following is the most typical update pattern: a program loads a new RDF file into the database, generating a large number of new triples without any queries. Additionally, but much less frequently, programs can mix updates and queries, deriving new triples from existing ones. But in this case, we expect that the number of generated triples is much smaller.

With consideration to these two usage patterns, we index the new triples lazily. Initially, we just store them in an in-memory heap as they arrive. Only when we need a certain triple ordering for query processing, we generate the desired index by sorting the triples in the workspace. For programs that interleave queries and updates, we expect bursts of insertions (i.e., many new triples derived from a query result). Therefore, we do not maintain the indexes but simply invalidate the indexes after a new update and rebuild them from scratch when needed for a later query. When the program terminates (or at a program-specified "commit point", see below), we build indexes for all six triple orderings and merge them into the corresponding differential indexes that are shared across all programs. In addition, the workspace manager for each program also maintains a string dictionary for new strings. We generate temporary string ids for each program, which are later resolved into final ids during the merge with the differential index. This technique avoids the need for concurrency control on the globally shared string dictionary, and is safe as the ids are not visible at the application level.

Differential indexes. The differential indexes, which are shared by all programs, are an intermediate layer between the programs' private workspaces and the full indexes of the main database. This intermediate indexing allows us to improve update throughput by aggregating many new triples into bulk operations against the main indexes. We keep differential indexes in main memory in our implementation, which simplifies their maintenance. When they exhaust a given memory budget, we initiate a merge process into the main indexes to release memory. For recoverability after soft crashes that lose memory content, newly created triples are written to a disk-resident log file in their plain form (i.e., only once, not in six different orders). Alternatively, we could store the differential indexes in flash RAM.

The differential indexes are B$^+$-trees for the different triple orderings, similar to the main indexes of Sect. 3. However, there are some differences. First, we only build the six full indexes for the different attribute permutations; there is no need for building the binary or unary aggregation indexes. As all data are in main memory, we can inexpensively construct aggregated indexes on the fly when needed, by using sort-based group-by operators. Furthermore, the indexes are not compressed, as we want to be able to modify them as fast as possible. Finally, we use a read-copy-update (RCU) mechanism [22] to allow queries to perform scans on the

differential indexes without any concurrency control. This is similar to B^{link}-tree-style techniques [32] but simpler and not as powerful. When a tree page is modified, we first copy the page, modify the page, and then change all pointers to the old page to point to the new page. Note that this only maximizes concurrency for read operations; write operations on the tree have to latch pages to protect the tree from becoming inconsistent because of concurrent writers. Further note that while the RCU mechanism allows for concurrent operations, it does not provide transaction isolation. Similar to the workspace management, the differential index management also includes maintaining a string dictionary for new strings, but here the string ids are already the final ones that will later be used in the disk-resident main indexes. The string dictionary also uses an RCU mechanism to allow lock-free reads, so queries are never blocked.

Merging the six differential indexes into the corresponding main indexes of the database is I/O-intensive and thus expensive, but relatively simple. Some care is needed because of the compression in the main indexes, but as compressed pages are self-contained and never need any other pages for uncompression, we can perform the merge page by page. Each of the six index merges can be performed independently. Overall, the merge process works pretty much like a standard bulk insertion of sorted data into a B^+-tree.

The string dictionary of the main database is updated as well, but here the merging is even simpler because of the absence of compression and the monotonicity of string ids (which means that without deletions new entries are always appended).

Query processing. All queries are run on the union of the main indexes, the differential indexes, and the workspace of the program that invokes the following query:

$$\text{SPO}^{\text{query}} = \text{SPO} \cup \text{SPO}^{\text{diff}} \cup \text{SPO}^{\text{prog}}$$

As all three indexes are sorted in the same way, we use merge-joins with union semantics ($\bowtie_{\cup}^{merge}$) to combine the indexes. For example, the triple pattern (90, ?a, ?b) would be translated into the following expression:

$$\begin{aligned}(\sigma_{S=90}(\text{SPO})) \; &\bowtie_{\cup}^{\text{merge}} \; (\sigma_{S=90}(\text{SPO}^{\text{diff}})) \\ &\bowtie_{\cup}^{\text{merge}} \; (\sigma_{S=90}(\text{SPO}^{\text{prog}}))\end{aligned}$$

These join operators are very cheap and require no additional memory, but cause some small CPU overhead due to additional comparisons. This is slightly troublesome, as each triple pattern in a query generates two of these extra joins. Most of these joins will turn out to be unnecessary, namely, whenever there are no new triples in the differential indexes or workspace that match a given triple pattern. The RDF-3X system therefore tries to eliminate unnecessary joins of this kind. One way to detect unnecessary scans of the differential indexes is to check for relevant triples during query compilation, as the differential indexes are in main memory anyway. Unfortunately, this approach faces complications when queries contain selections that are pushed down below joins by the query optimizer. Instead, we implemented the $\bowtie_{\cup}^{\text{merge}}$ operator such that at query execution time it removes itself from the execution plan once one of its inputs (differential index or workspace) is or becomes empty.

7.2 Delete operations

The algorithms for handling deletions and for running queries in the presence of deletions are very similar to the described case of insertions. When deleting a triple, we "insert" the triple into the workspace and differential indexes with a deletion flag. During queries, the $\bowtie_{\cup}^{\text{merge}}$ operator interprets this flag and behaves like an anti-join when encountering deleted triples, eliminating matching triples on the other side. When merging the differential indexes into the main indexes, triples marked for deletion cause the deletion of their counterparts in the main indexes and are discarded after the entire merge completes.

While deleting the triples themselves is simple, maintaining a compact string dictionary is more challenging. Ideally we want to eliminate strings that are no longer referenced by any triple, to avoid monotonic growth of the dictionary. When deleting a single triple we do not know a priori if the strings used by the triple occur in any other triples. We can find this out by checking the fully aggregated indexes of RDF-3X, which implicitly give the number of occurrences for each string, but checking these for each triple would be expensive. Fortunately, we do not have to check for each triple, but only when we eliminate an entry from an aggregated unary index. Assume that we want to delete the SPO triples (1, 2, 3) and (1, 4, 5). When merging the corresponding differential index entries into the main indexes, the aggregated unary index for S will be looked up with value $S = 1$. The *count* entry for 1 is decreased by two to reflect two eliminated triples. Only if the count reaches zero the whole entry is eliminated, and only in this case do we check if the value 1 occurs in the other two unary indexes for P and O. If none of the unary indexes signals the presence of the value 1, we can remove 1 from the dictionary.

Alternatively to the immediate checking of all unary indexes, we could periodically run a garbage collector to eliminate strings with zero references. This can be implemented by a single merge scan over the three unary aggregation indexes and the string dictionary. It depends on the data and the update behavior whether this is more efficient or not. If we assume that deletions are very infrequent, it is cheaper to perform the checks immediately as described above.

7.3 Support for atomicity and weak isolation

The above procedures provide proper isolation between individual operations, but no transactional ACID guarantees for entire blocks of operations. While full-fledged transactions are beyond the scope of this paper, we have successfully added support for the atomicity of programs and a weak form of isolation level among concurrent program executions.

In the first stage of our staging architecture, programs that contain only update operations (and no queries) are automatically isolated by using their private workspaces. A program's termination can thus be easily viewed as a "commit point". Alternatively, a program can specify multiple commit points during its run-time; each commit point prompts a merge with the globally shared differential indexes. During merges with the differential indexes and with the main indexes, the consistency of the index trees is guaranteed by latching and the RCU mechanism. When a program merges its workspace into a differential index, it latches index leaf pages in a lock-coupling manner, always holding latches for two successive leaf pages at the same time. As all write programs have the very same page access order during such a merge, the lock-coupling technique guarantees serializability between such write-only programs. The merges are naturally idempotent, so we obtain crash-resilience for free: an interrupted merge can simply be restarted and would then ignore inserted triples that are already present in the merge target.

Programs that issue only queries can run against both the differential indexes and the main indexes without any locking or latching (even the latter is avoided because of the RCU mechanism for concurrent writers). Of course, it is impossible to guarantee transactional serializability between readers and writers this way, and even snapshot isolation is not feasible this way. However, no query will ever see a non-committed update, as merges from workspaces into differential indexes start only after commit points. Thus, our staging architecture provides the *read-committed isolation level*, which seems sufficient for many applications outside enterprise-business IT. This property also holds for programs that mix queries and updates: all their queries see committed data and their updates are properly serialized against concurrent read-write programs.

Overall, the deferred update handling has the following salient properties:

1. Query execution times are unaffected if there are no updates that satisfy any triple patterns in the query, and have very small overhead otherwise.
2. Updates to the main indexes are aggregated into efficient bulk operations.
3. We provide support for atomic batches of update operations. Queries never have to wait; they run at the read-committed isolation level (which is weaker than serializability or snapshot isolation).

We will report on performance measurements for updates in Sect. 8.5.

8 Evaluation

8.1 General setup

For evaluating the performance of RDF-3X, we used three large datasets with different characteristics and compared the query run-times to other approaches (discussed below). All experiments were conducted on a Dell D620 PC with a 2 Ghz Core 2 Duo processor, 2 GBytes of memory, and running a 64-bit Linux 2.6.24 kernel. For the cold-cache experiments we used the `/proc/sys/vm/drop_caches` kernel interface to drop all filesystem caches before restarting the various systems under test. We repeated all queries five times (including the dropping of caches and the system restart) and took the best result to avoid artifacts caused by OS activity. For warm caches we ran the queries five times without dropping caches, again taking the best run-time.

Our primary comparison is against the *column-store-based approach* presented in [1], which has already been shown to be highly superior to all other DBMS-based approaches in that paper. We implemented the approach as described in [1], but used MonetDB 5.2.0 [36] as a backend instead of C-Store, because C-Store is no longer maintained and does not run on our hardware/OS platform. The C-Store web page [10] suggests using MonetDB instead, and MonetDB worked fine. Note that our setup uses substantially weaker hardware than [1]; in particular the hard disk is about a factor of 6 slower than the very fast RAID used in [1], transferring ca. 32 MB/s in sequential reads. Taking this factor of 6 into account, the performance numbers we got for MonetDB are comparable to the C-Store numbers from [1]. For one query (Q6) MonetDB was significantly faster than a factor of 6 (14s vs. 10s), while for another (Q7) significantly slower (61s vs. 1.4s), but overall MonetDB performed as expected given the slower hard disk.

As a second opponent to RDF-3X, we used *PostgreSQL 8.3 as a triple store* with indexes on the string dictionary and on (subject, predicate, object), (predicate, subject, object), and (predicate, object, subject). This emulates a Sesame-style [9] storage system. We also tried out the current release of a leading commercial database system with built-in RDF support, but could not obtain acceptable performance anywhere near the run-times of the other systems. When using its own RDF query language and despite trying several of its auto-tuning options, it performed significantly slower than the PostgreSQL triple store even for simple queries, and failed to

Table 4 Database load after triple construction

	Barton		Yago		Librarything	
	Load time (min)	DB size (GB)	Load time (min)	DB size (GB)	Load time (min)	DB size (GB)
RDF-3X	13	2.8	25	2.7	20	1.6
MonetDB	11	1.6	21	1.1	4	0.7
PostgreSQL	30	8.7	25	7.5	20	5.7

Table 5 Query run-times in seconds for the Barton dataset

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Geom. mean
Cold caches								
RDF-3X	0.14	3.10	31.49	11.57	18.82	2.75	32.61	5.9
MonetDB	5.66	11.70	54.66	34.77	80.96	14.14	61.52	26.4
PostgreSQL	28.32	181.00	291.04	224.61	199.07	207.72	271.20	167.8
Warm caches								
RDF-3X	0.001	1.17	2.22	1.58	0.49	1.20	1.26	0.4
MonetDB	0.65	1.41	3.63	9.59	77.53	1.86	2.48	3.8
PostgreSQL	8.15	174.41	286.76	26.80	8.77	206.46	231.79	64.3

execute more complex queries in reasonable time. We therefore omitted it from the presentation.

In addition to these DBMS-based opponents, we tried several systems from the semantic web community that are available as open-source code. Unfortunately, none of them scaled to the dataset sizes that we used. We first tried the popular Jena2 system [64] which came out of the HP Labs Semantic Web Programme. We used *Jena version 2.5.5* with the SDB 1.0 wrapper and Apache Derby 10.3.2.1, but were unable to import any of our three datasets in 24 h. Judging from the file growth, the system became continuously slower and did not seem to terminate in a reasonable time. We also tried *Yars2* [23,66], but again were unable to import any of our datasets in 24 h. Finally, we tried *Sesame 2.0* [9,41], which is supposed to be one of the fastest semantic web systems. Sesame 2.0 was able to import the Barton dataset in 13 h, but then needed ca. 15 min for each of the first two queries and crashed due to excessive memory usage for the more complex queries.

Note that both MonetDB and RDF-3X could import the data sets in less than half an hour, and could run the queries in the order of seconds. Other semantic web approaches usually assume that the RDF data fits into main memory, which is not the case here. All experiments below therefore only consider RDF-3X, the column-stored-based approach on top of MonetDB, and the PostgreSQL-based triples store.

Independently of the database system, each of the datasets discussed below is first brought into a factorized form: one file with RDF triples represented as integer triples and one dictionary file mapping from integers to literals. All three systems use the same files as inputs, loading them into fact table(s) and dictionary. The load times of this second phase and the database sizes are shown in Table 4. The MonetDB sizes are the initial sizes after loading. After running the benchmark the sizes were 2.0/2.4/6.9 GB. Apparently, MonetDB builds some index structures on demand.

8.2 Barton dataset

For the first experiment we used the Barton Library dataset and the queries proposed as a benchmark in [1]. We processed the data as described in [1], converting it into triple form using the Redland parser and then imported the triples into our RDF-3X system. In [1] the authors pruned the data due to C-Store limitations (they dropped all triples containing strings longer than 127 bytes and some triples with a huge number of join partners). We left the complete data as it was and imported it directly into all three systems. Overall the data consists of 51,598,328 distinct triples, and 19,344,638 distinct strings. The original data was 4.1 GB in RDF (XML) format, 7.7 GB in triple form, and 2.8 GB in our RDF-3X store including all indexes and the string dictionary.

We used the queries from [1] for our experiment, but as they were given in SQL we had to reformulate them in SPARQL for RDF-3X. Appendix A shows all queries. The results of our measurements are shown in Table 5. We include also the geometric mean of the query set, which is often used as a workload-average measure in benchmarks (e.g., TPC) and is more resilient to extreme outliers than the arithmetic average.

Table 6 Query run-times in seconds for the Yago dataset

	A1	A2	A3	B1	B2	B3	C1	C2	Geom. mean
Cold caches									
RDF-3X	0.29	0.28	1.20	0.28	0.99	0.33	2.23	4.23	0.73
MonetDB	43.55	44.13	55.49	62.94	182.39	72.22	101.66	157.11	78.29
PostgreSQL	1.62	6.31	5.46	3.04	117.51	4.71	29.84	59.64	10.66
Warm caches									
RDF-3X	0.02	0.02	0.02	0.01	0.05	0.01	0.61	1.44	0.04
MonetDB	36.92	32.96	34.72	49.95	64.84	52.22	84.41	131.35	54.62
PostgreSQL	0.08	0.43	0.20	0.11	7.33	0.12	0.31	50.37	0.56

The first observation is that RDF-3X performs much better than MonetDB for all queries, and MonetDB itself performs much better than PostgreSQL (as reported in [1]). We first discuss the results for RDF-3X versus MonetDB. When comparing the cold-cache times and the warm-cache times, it becomes clear that disk I/O has a large impact on the overall run-times. RDF-3X simply reads less data due to its highly compressed index structures, therefore outperforming MonetDB in the cold-cache case by a typical factor of 2–5, and sometimes by more than 10. In the warm-cache case, the differences are typically smaller but still substantial (factor of 2, sometimes much higher). An interesting data point is query Q4, which is relatively expensive in terms of constructed join pairs, and where RDF-3X performs very well even in CPU-dominated warm-cache case. Furthermore, we observe that a third critical aspect besides I/O and CPU usage is memory consumption. Query Q5 has a very large intermediate result. MonetDB apparently materializes parts of these intermediate results in main memory. As a consequence only few database pages can be buffered, which significantly hurts warm-cache behavior.

PostgreSQL has problems with this dataset due to the nature of the queries for this benchmark. Nearly all queries are aggregations queries (usually aggregating by predicate), and the result cardinality is large which entails expensive dictionary lookups. For other, more natural, kinds of RDF queries, PostgreSQL performs much better, as we will see in the next two subsections.

To get an idea how a Yars2-style system could scale, we experimentally disabled all aggregated indices. This increased the geometric means to 9.52 s (cold) and 1.04 s (warm), which is significantly slower than RDF-3X. This is still much faster than the other systems, though, in particular due to our runtime system and query optimizer.

8.3 Yago dataset

The Barton dataset is relatively homogeneous, as it describes library data. As a second dataset we therefore used Yago [55] which consists of facts extracted from Wikipedia (exploiting the infoboxes and category system of Wikipedia) and integrated with the WordNet thesaurus. The Yago dataset contains 40,114,899 distinct triples and 33,951,636 distinct strings, consuming 3.1 GB as (factorized) triple dump. RDF-3X needs 2.7 GB for all indexes and the string dictionary. As queries we considered three different application scenarios–entity-oriented, relationship-oriented, and queries with unknown predicates—and derived eight benchmark queries, shown in Appendix A. These queries are more "natural" than the Barton queries, as they are standard SPARQL without any aggregations and with explicitly given predicates. On the other hand, the queries are much larger (requiring more many-to-many joins) and thus more difficult to optimize and execute.

The results are shown in Table 6. Again, RDF-3X clearly outperforms the other two systems for both cold and warm caches, by a typical factor of 5–10. Here PostgreSQL performed much better than MonetDB. This is most likely caused by the poor join orderings in MonetDB. The warm-cache run-times are nearly as high as the cold-cache times, which indicates that MonetDB creates large intermediate results.

In general, this dataset is much more challenging for the query optimizer, as queries are more complex and selectivity estimates are important. While testing our system, we noticed that selectivity mis-estimations can easily cause slowdown by a factor of 10-100 on this dataset. RDF-3X shows excellent performance regarding both the run-time execution and the choice of execution plans by the optimizer.

8.4 LibraryThing dataset

As a third dataset we used a partial crawl of the LibraryThing book-cataloging social network [31]. It consists of 9,989 users, 5,973,703 distinct books (personally owned by these users), and the tags that the users have assigned to these books. Overall, the dataset consists of 36,203,751 triples and 9,352,954 distinct strings, consuming 1.8 GB in its original

Table 7 Query run-times in seconds for the LibraryThing dataset

	A1	A2	A3	B1	B2	B3	C1	C2	Geom. mean
Cold caches									
RDF-3X	0.28	1.01	21.85	0.14	0.34	4.17	0.28	1.21	0.89
MonetDB	2.14	1.41	1220.09	1.63	2.20	*	1.66	>15 min/*	>8.16
PostgreSQL	20.78	1.43	715.64	0.88	2.13	>8 h	5108.01	1031.63	>93.91
Warm caches									
RDF-3X	0.05	0.15	0.95	0.01	0.12	1.61	0.03	0.26	0.13
MonetDB	0.82	0.77	1171.82	0.56	0.63	*	0.59	>15 min/*	>4.39
PostgreSQL	12.31	0.05	611.41	0.02	0.66	>8 h	5082.34	1013.01	>30.43

* System crashed, see description

form and 1.6 GB in RDF-3X. One particularity of this dataset is that is has a heterogeneous link structure. In our RDF representation, each tag is mapped to a predicate, linking the user to the book she tagged. As the number of different tags is very large, the dataset contains 338,824 distinct predicates, whereas the other two datasets contained only 285 and 93 distinct predicates, respectively. While other mappings onto RDF may be possible, we used this extremely non-schematic approach as a stress test for all competing systems.

These data make compression more difficult for RDF-3X, and causes serious problems for MonetDB. MonetDB was unable to handle 338,824 tables, creating millions of files in the file system and swapping all the time. We therefore used a hybrid storage scheme for MonetDB for this dataset. We partitioned the 1,000 most commonly used predicates as described in [1], and placed the remaining triples (ca. 12%) in one big triples table. We again constructed three kinds of queries: book-oriented, user-oriented, and navigating book and user chains (see Appendix A). In contrast to the Yago dataset, there were few predicates that occurred in millions of triples, which lowered the impact of join-ordering decisions. On the other hand, the data itself is very inhomogeneous so that selectivities are more difficult to predict.

The results are shown in Table 7. RDF-3X performs very well, outperforming the opponents by a typical factor of at least 5 and more than 30 in some cases. Between MonetDB and PostgreSQL there is no clear winner. Overall, MonetDB seems to perform better, but it crashed two times. It refused to execute query B3 ("too many variables"), probably because it included three patterns with variable predicates (and thus at least 3000 scans). In query C2 it crashed after 15 min due to lack of disk space, as it had materialized a 20 GB intermediate result (which is more than ten times the size of the whole database).

The query A3 stands out by its high run-times. It performs many joins with relatively unselective predicates (book authors, etc.), which are expensive. The other "difficult" queries (B3, C1, C2) are not that difficult per se, they just require the right choice of execution plans. B3, for example, finds all users with books tagged as English, French, and German. PostgreSQL starts this query by collecting all pairs of books a user has, which is prohibitively expensive. The optimizer of RDF-3X, on the other hand, chooses a plan that collects for each tag the users with such books and then joins the results, which is much more efficient.

8.5 Updates

We studied the update performance by "steady state" measurements as follows. Inserting new triples into an empty or very small database is clearly cheap as the merge is trivial and the database fits into main memory. We therefore took the full database for the Barton dataset, which does not fit into the available memory on our test machine, and initiated insert operations for all triples of the dataset. As these very same triples already exist in the main indexes, the merge process actually discards the new triples. But initially, the new triples are added to a workspace and the differential indexes. Moreover, the attempted merging into the main indexes touches the entire database, causes a recomputation of every index page, and we even force the writing of the "modified" pages back to disk. This way, we emulate a heavy upload load, while keeping the overall database size constant.

We ran this experiment with a single (single-threaded) program and varied two parameters: (1) the batch size, i.e., the number of inserted triples after which we periodically generate a "commit point" and merge the workspace into the differential indexes, and (2) the maximum differential index size at which we a force a merge into the main indexes (measured in numbers of indexed triples). Note that having a batch size larger than the maximum differential index hardly ever makes sense (and would only cause additional merges but no fundamental change of disk I/O behavior), so we ignore it here.

The throughput results of this experiment are shown in Table 8. We consistently achieved a sustained throughput of more than 20,000 inserts per second, which is remarkably high given that RDF-3X is primarily designed and optimized

Table 8 Insertion rates in triples/second for varying batch and differential index sizes

Batch size	Max. differential index size		
	100,000	500,000	1,000,000
100,000	23,059	25,391	24,701
500,000		25,972	23,407
1,000,000			23,490

Table 9 Effect of updates on warm-cache query performance

	No updates	Irrelevant updates	Relevant updates
1 thread	0.63 s	0.63 s	0.69 s
2 threads	–	0.64 s	0.70 s

for queries. The differences in insertion rates for different parameter settings are not that large, but exhibit trends that we confirmed by additional experiments on different machines. First, increasing the maximum differential index size obviously tends to improve the performance. Second and also unsurprisingly, inserting in larger batches tends to be faster than inserting in many smaller batches. But there are exceptions to these trends: in our specific setup, a maximum differential index size of 500,000 triples outperformed the index with 1,000,000 triples. The reason for this is memory pressure. Both the differential index and the page buffer manager require memory; by increasing the size of the differential index we reduce the available buffer space and thus incur more disk I/O's during index merges. The optimal size of the differential index depends on the machine resources and configuration. In experiments on a machine with more main memory, the larger index was indeed faster, but as the differences are not that large one can select a conservative size (e.g., 100,000 triples) and obtain good performance without affecting the rest of the system. Ideally the buffer manager would keep track of the I/O rate and adjust the space available for the differential index dynamically, but such auto-tuning is beyond the scope of this work.

Overall, the insertion rates in Table 8 are fairly good. For comparison, the insertion rates reached by our initial database bulk load are about 80,000 triples per second, which is a factor of 3–4 better. But our setting here is much more complex than the initial bulk load, as we have to integrate the changes into the existing database (which has to be read for the integration and does not fit into main memory). Therefore, a slowdown by 3× seems acceptable. Note that our experiment is a stress test, as every inserted triple already exists and we therefore have to read and touch everything. Inserting completely new data that are just appended and not combined with the existing triples is much faster and achieves more or less the same throughput as the initial bulk load procedure.

Next, we studied the effect of updates on query performance. The first observation is that the differential index has nearly no impact on the cold-cache case, as the queries are then dominated by I/O costs and the differential index is in main memory. We therefore concentrate on the warm-cache case. We ran the query *(?a,type,?b), (?a,type,?c)* (a simplified version of the query Q2 of the Barton benchmark) in different scenarios. We used a relatively simple query, as in more complex queries the CPU costs of joins and aggregations potentially hide the differences that we want to measure.

The first scenario is the static database, running the query without any updates. In the second scenario, we performed updates in parallel to the query, but ensured that the updates do not match any condition of the query. In the third and last scenarios, we inserted query-relevant triples (which satisfy at least one query condition). The new triples used ids that follow those of any existing triple in the sort order, thus forcing the merge joins with workspaces and differential indexes to continue to the end. For all three scenarios, we ran a single-threaded and a two-threaded experiment (on a two-cores machine with true parallelism). With a single thread, query and update operations are alternated over a long time period. With two threads, one thread keeps repeating the query while the other performs update operations in parallel. The single-thread experiment studies the overhead of having to consider workspaces and differential indexes by additional merge joins in query processing. The two-threads experiment additionally reflects the overhead of maintaining differential indexes in parallel to ongoing queries. The batch size and maximum differential index size were set to 500,000. We report the median of the measured response times.

The results are shown in Table 9. Our extended query processor discussed in Sect. 7.1 performs extremely well in the case of query-irrelevant updates. The first two scenarios have almost identical run-times. The third scenario, on the other hand, exhibits slightly increased query response times. However, the overall CPU overhead of merge joins with the differential indexes is small. Although queries never have to wait for locks or latches, there is some moderate amount of CPU and I/O contention that leads to query response times increasing by about 10 percent. This is a very moderate overhead for a fairly demanding stress-test experiment.

9 Conclusion

This paper has presented the RDF-3X engine, a RISC-style architecture for executing SPARQL queries over large repositories of RDF triples. As our experiments have shown, RDF-3X outperforms the previously best systems by a large margin. In particular, it addresses the challenge of schema-free data and, unlike its opponents, copes very well with data that exhibits large diversity of property names. The salient features of RDF-3X that lead to these performance gains are (1) exhaustive but very space-efficient triple indexes that

eliminate the need for physical-design tuning, (2) a streamlined execution engine centered around very fast merge joins, (3) a smart query optimizer that chooses cost-optimal join orderings and can do this efficiently even for long join paths (involving 10 to 20 joins), and (4) a selectivity estimator based on statistics for frequent paths that feeds into the optimizer's cost model. In addition, although RDF database are most likely query-intensive, RDF-3X provides decent support of updates, for both incremental loading in batched mode and individual insert or delete operations. This is achieved by a staging architecture with deferred index maintenance, provides good update throughput, and incurs only small overhead on concurrent queries.

Our future work includes various extensions and optimizations. First, we plan to further improve the query processor and optimizer (e.g., based on magic sets) and provide support for RDF search features that go beyond the current SPARQL standard. Along the latter lines, one direction is to allow more powerful wild-card patterns for entire paths, in the spirit of the XPath descendants axis but for graphs rather than trees. Proposals for extending SPARQL have been made [3], but there is no implementation yet. Second, we are interested in providing ranked query results, based on application-specific scoring models. This calls for top-k query processing and poses challenging issues for algorithms and query optimization. Third, full SPARQL support requires some additional information, in particular typing information. We feel that this can be included in the dictionary, but determining the best encoding relative to runtime performance and compression rate needs more work. Fourth and last, we believe that our current support for atomicity and read-committed isolation can be further extended towards transactional guarantees. In particular, we want to pursue adding versioning and snapshot isolation to the RDF-3X engine.

Appendix A: SPARQL queries

For completeness, we include the SPARQL queries used in our evaluation.

Barton Dataset. As the queries in [1] were given in SQL, we had to reformulate them in SPARQL. We abbreviate some constants here, the queries are discussed in [1]. We had to extend the SPARQL projection clause a bit to get equivalent queries. *count* is like *distinct* but includes the number of occurrences. *duplicates* is like *count* but only returns bindings that are produced at least twice.

Q1: select count ?c where { ?a a ?c }

Q2: select count ?bp where { ?as a <Text>; ?bp ?bo. filter (?bp in <predicate list>)}

Q3: select duplicates ?bp ?bo where { ?as a <Text>; ?bp ?bo. filter (?bp in <predicate list>) }

Q4: select duplicates ?bp ?bo where { ?as a <Text>; ?bp ?bo; <language> <iso639-2b/fre>. filter (?bp in <predicate list>) }

Q5: select ?as ?co where { ?as <origin> <marcorg/DLC>; <records> ?bo. ?bo a ?co. filter (?co != <Text>) }

Q6: select count ?ap where { {?as a <Text>} union {?as <records> []; a <Text>} ?as ?ap []. filter (?ap in <predicate list>)}

Q7: select ?as ?bo ?co where { ?as <point> "end"; <encoding> ?bo; a ?co }

Yago Dataset. We grouped the queries thematically into three groups. The first group consists of oriented facts, e.g., "scientists from Switzerland with a doctoral advisor from Germany" (A1). The second group is relationship oriented, e.g., "two actors from England playing together in the same movie" (B1). The third group examines relationships with unknown predicates, e.g., "two scientists related to the same city" (C1).

A1: select ?gn ?fn where { ?gn <givenNameOf> ?p. ?fn <familyNameOf> ?p. ?p <type> "scientist"; <bornInLocation> ?city; <hasDoctoralAdvisor> ?a. ?a <bornInLocation> ?city2. ?city <locatedIn> "Switzerland". ?city2 <locatedIn> "Germany". }

A2: select ?n where { ?a <isCalled> ?n; <type> "actor"; <livesIn> ?city; <actedIn> ?m1; <directed> ?m2. ?city <locatedIn> ?s. ?s <locatedIn> "United_States". ?m1 <type> "movie"; <producedInCountry> "Germany". ?m2 <type> "movie"; <producedInCountry> "Canada". }

A3: select distinct ?n ?co where { ?p <isCalled> ?n. { ?p <type> "actor" } union { ?p <type> "athlete" } ?p <bornInLocation> ?c. ?c <locatedIn> ?s. ?s <locatedIn> ?co. ?p <type> ?t. filter(?t reaches "politician" via <subClassOf>) }

B1: select distinct ?n1 ?n2 where { ?a1 <isCalled> ?n1; <livesIn> ?c1; <actedIn> ?movie. ?a2 <isCalled> ?n2; <livesIn> ?c2; <actedIn> ?movie. ?c1 <locatedIn> "England". ?c2 <locatedIn> "England". filter (?a1 != ?a2) }

B2: select ?n1 ?n2 where { ?p1 <isCalled> ?n1; <bornInLocation> ?city; <isMarriedTo> ?p2. ?p2 <isCalled> ?n2; <bornInLocation> ?city. }

B3: select distinct ?n1 ?n2 where { ?n1 <familyNameOf> ?p1. ?n2 <familyNameOf> ?p2. ?p1 <type> "scientist"; <hasWonPrize> ?award; <bornInLocation> ?city. ?p2 <type> "scientist"; <hasWonPrize> ?award; <bornInLocation> ?city. filter (?p1 != ?p2) }

C1: select distinct ?n1 ?n2 where {?n1 <familyNameOf> ?p1. ?n2 <familyNameOf> ?p2. ?p1 <type> "scientist"; [] ?city. ?p2 <type> "scientist"; [] ?city. ?city <type> <site> filter (?p1 != ?p2) }

C2: select distinct ?n where { ?p <isCalled> ?n; [] ?c1. [] ?c2. ?c1 <type> <village>; <isCalled> "London". ?c2 <type> <site>; <isCalled> "Paris". }

LibraryThing Dataset. Similar to the Yago setting we used three query groups. First queries on books (e.g., A1 "books tagged with romance, love, suspense, mystery"), second queries on users (e.g., B1 "users who like crime novels and Arthur Conan Doyle and have friends who like romances and Jane Austen"), and third queries with chains over books and users (e.g., C1 "books tagged with romance by users who have friends or friends of friends who have tagged books with documentary which have also been tagged with thriller").

A1: select distinct ?title where { ?b <hasTitle> ?title. [] <romance> ?b. [] <love> ?b. [] <suspense> ?b. [] <mystery> ?b. }

A2: select distinct ?title where { ?b <hasTitle> ?title. ?u <romance> ?b; <love> ?b; <suspense> ?b. }

A3: select distinct ?title where { ?b <hasTitle> ?title; <hasAuthor> ?a. ?u <mystery> ?b; <romance> []. ?b2 <hasAuthor> ?a. [] <children> ?b2. }

B1: select distinct ?u where { ?u <crime> []; <hasFavoriteAuthor> "Arthur Conan Doyle"; <hasFriend> ?f. ?f <romance> []; <hasFavoriteAuthor> "Jane Austen". }

B2: select distinct ?u where { { ?u <documentary> ?b1; <suspense> ?b1 } union { ?u <biography> ?b2; <suspense> ?b2 } union { ?u <documentary> ?b3; <mystery> ?b3 } union { ?u <biography> ?b4; <mystery> ?b4 } }

B3: select distinct ?u where { ?u [] ?b1; [] ?b2; [] ?b3. [] <english> ?b1. [] <german> ?b2. [] <french> ?b3. }

C1: select distinct ?u where { { ?u <romance> ?b1; <hasFriend> ?f1. ?f1 <biography> ?b2. [] <thriller> ?b2. } union { ?u <romance> ?b1. <hasFriend> ?f1. ?f1 <hasFriend> ?f2. ?f2 <biography> ?b2. [] <thriller> ?b2. } }

C2: select distinct ?u ?u2 where { ?u <hasFavoriteAuthor> ?a1; <america> []; <hasInterestingLibrary> ?u2. ?b1 <hasAuthor> ?a1. [] <europe> ?b1. ?u2 <hasFavoriteAuthor> ?a2; <europe> []. ?b2 <hasAuthor> ?a2. [] <america> ?b2. }

References

1. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable semantic web data management using vertical partitioning. In: VLDB, pp. 411–422 (2007)
2. Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., Tolle, K.: The ICS-FORTH RDFSuite: Managing voluminous RDF description bases. In: SemWeb (2001)
3. Anyanwu, K., Maduko, A., Sheth, A.P.: SPARQ2L: towards support for subgraph extraction queries in rdf databases. In: WWW, pp. 797–806 (2007)
4. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.G.: Dbpedia: a nucleus for a web of open data. In: ISWC/ASWC, pp. 722–735 (2007)
5. Bayardo, R.J., Jr.: Efficiently mining long patterns from databases. In: SIGMOD, pp. 85–93 (1998)
6. Baolin, L., Bo, H.: Path queries based RDF index (2005)
7. Baolin, L., Bo, H.: HPRD: A high performance RDF database. In: NPC, pp. 364–374 (2007)
8. BioPAX: Biological Pathways Exchange. http://www.biopax.org/
9. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: an architecture for storing and querying RDF data and schema information. In: Spinning the Semantic Web, pp. 197–222 (2003)
10. C-Store. http://db.csail.mit.edu/projects/cstore/
11. Chaudhuri, F., Shim, K.: Optimization of queries with user-defined predicates. ACM Trans. Database Syst. **24**(2), 177–228 (1999)
12. Chaudhuri, S., Weikum, G.: Rethinking database system architecture: towards a self-tuning RISC-style database system. In: VLDB, pp. 1–10 (2000)
13. Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An efficient SQL-based RDF querying scheme. In: VLDB, pp. 1216–1227 (2005)
14. Chu, E., Beckmann, J.L., Naughton, J.F.: The case for a wide-table approach to manage sparse relational data sets. In: SIGMOD, pp. 821–832 (2007)
15. DeHaan, D., Tompa, F.W.: Optimal top-down join enumeration. In: SIGMOD, pp. 785–796 (2007)
16. den Bercken, J.V., Seeger, B.: An evaluation of generic bulk loading techniques. In: VLDB, pp. 461–470 (2001)
17. Eickler, A., Gerlhof, C.A., Kossmann, D.: A performance evaluation of OID mapping techniques. In: Dayal, U., Gray, P.M.D., Nishio, S. (eds.) VLDB, pp. 18–29. Morgan Kaufmann (1995)
18. Galindo-Legaria, C.A., Pellenkoft, A., Kersten, M.L.: Fast, randomized join-order selection—why use transformations? In: Bocca, J.B., Jarke, M., Zaniolo, C. (eds) VLDB, pp. 85–95. Morgan Kaufmann (1994)
19. Getoor, L., Diehl, C.P.: Link mining: a survey. SIGKDD Explor. **7**(2), 3–12 (2005)
20. Graefe, G.: Query evaluation techniques for large databases. ACM Comput. Surv. **25**(2), 73–170 (1993)
21. Halevy, A.Y., Franklin, M.J., Maier, D.: Principles of dataspace systems. In: PODS, pp. 1–9 (2006)
22. Hart, T.E., McKenney, P.E., Brown, A.D., Walpole, J.: Performance of memory reclamation for lockless synchronization. J. Parall. Distrib. Comput. **67**(12), 1270–1285 (2007)
23. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A federated repository for querying graph structured data from the web. In: ISWC/ASWC, pp. 211–224 (2007)
24. Hartig, O., Heese, R.: The SPARQL query graph model for query optimization. In: ESWC, pp. 564–578 (2007)
25. Hogan, A., Harth, A.: The ExpertFinder corpus 2007 for the benchmarking development of expert-finding systems. In: International ExpertFinder Workshop (2007)
26. Huynh, D., Mazzocchi, S., Karger, D.R.: Piggy bank: experience the semantic web inside your web browser. J. Web Sem. **5**(1), 16–27 (2007)
27. ICS-FORTH RDF suite. http://athena.ics.forth.gr:9090/RDF/
28. Jena: a Semantic Web Framework for Java. http://jena.sourceforge.net/
29. Jermaine, C.M., Omiecinski, E., Yee, W.G.: The partitioned exponential file for database storage management. VLDB J. **16**(4), 417–437 (2007)
30. Kersten, M., Siebes, A.P.: An organic database system. Technical report, CWI (1999)
31. LibraryThing. http://www.librarything.com
32. Lomet, D.B., Salzberg, B.: Concurrency and recovery for index trees. VLDB J. **6**(3), 224–240 (1997)
33. Maduko, A., Anyanwu, K., Sheth, A.P., Schliekelman, P.: Estimating the cardinality of RDF graph patterns. In: WWW, pp. 1233–1234 (2007)
34. Maduko, A., Anyanwu, K., Sheth, A.P., Schliekelman, P.: Graph summaries for subgraph frequency estimation. In: ESWC, pp. 508–523 (2008)
35. Moerkotte, G., Neumann, T.: Analysis of two existing and one new dynamic programming algorithm for the generation of optimal

bushy join trees without cross products. In: VLDB, pp. 930–941 (2006)
36. MonetDB. http://monetdb.cwi.nl/
37. Muth, P., O'Neil, P.E., Pick, A., Weikum, G.: The LHAM log-structured history data access method. VLDB J. **8**(3–4), 199–221 (2000)
38. Neumann, T., Moerkotte, G.: An efficient framework for order optimization. In: ICDE, pp. 461–472 (2004)
39. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. PVLDB **1**(1), 647–659 (2008)
40. O'Neil, P.E., Cheng, E., Gawlick, D., O'Neil, E.J.: The log-structured merge-tree (LSM-tree). Acta Inf. **33**(4), 351–385 (1996)
41. OpenRDF. http://www.openrdf.org/index.jsp
42. Oracle technical network, semantic technologies center. http://www.oracle.com/technology/tech/semantic_technologies/index.html
43. RDF-3X. http://www.mpi-inf.mpg.de/~neumann/rdf3x
44. RDFizers. http://simile.mit.edu/wiki/RDFizers
45. Schmidt, M., Hornung, T., Knchlin, N., Lausen, G., Pinkel, C.: An experimental comparison of RDF data management approaches in a SPARQL benchmark scenario. In: International Semantic Web Conference, pp. 82–97 (2008)
46. Scholer, F., Williams, H.E., Yiannis, J., Zobel, J.: Compression of inverted indexes for fast query evaluation. In: SIGIR, pp. 222–229 (2002)
47. Sears, R., Callaghan, M., Brewer, E.: Rose: compressed, log-structured replication. PVLDB **1**(1), 526–537 (2008)
48. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Bernstein, P.A. (ed.) SIGMOD, pp. 23–34. ACM (1979)
49. Semantic web challenge. http://challenge.semanticweb.org
50. Sidirourgos, L., Goncalves, R., Kersten, M.L., Nes, N., Manegold, S.: Column-store support for RDF data management: not all swans are white. PVLDB **1**(2), 1553–1563 (2008)
51. Simmen, D.E., Shekita, E.J., Malkemus, T.: Fundamental techniques for order optimization. In: SIGMOD, pp. 57–67 (1996)
52. Steinbrunn, M., Peithner, K., Moerkotte, G., Kemper, A.: Bypassing joins in disjunctive queries. In: Dayal, U., Gray, P.M.D., Nishio, S. (eds.) VLDB, pp. 228–238. Morgan Kaufmann (1995)
53. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: Sparql basic graph pattern optimization using selectivity estimation. In: WWW, New York, NY, USA, April 2008. ACM Press, to appear
54. Stonebraker, M., Bear, C., Çetintemel, U., Cherniack, M., Ge, T., Hachem, N., Harizopoulos, S., Lifter, J., Rogers, J., Zdonik, S.B.: One size fits all? part 2: Benchmarking studies. In: CIDR, pp. 173–184 (2007)
55. Suchanek, F.M., Kasneci, G., Weikum, G.: Yago: a large ontology from wikipedia and wordNet. J. Web Sem. **6**(3), 203–217 (2008)
56. Theoharis, Y., Christophides, V., Karvounarakis, G.: Benchmarking database representations of rdf/s stores. In: International Semantic Web Conference, pp. 685–701 (2005)
57. Udrea, O., Pugliese, A., Subrahmanian, V.S.: GRIN: A graph based RDF index. In: AAAI, pp. 1465–1470 (2007)
58. uniprot RDF. http://dev.isb-sib.ch/projects/uniprot-rdf/
59. Vanetik, N., Gudes, E.: Mining frequent labeled and partially labeled graph patterns. In: ICDE, pp. 91–102 (2004)
60. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. PVLDB **1**(1), 1008–1019 (2008)
61. W3C: Resource Description Framework (RDF). http://www.w3.org/RDF/.
62. W3C: SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/
63. Westmann, T., Kossmann, D., Helmer, S., Moerkotte, G.: The implementation and performance of compressed databases. SIGMOD Rec. **29**(3), 55–67 (2000)
64. Wilkinson, K., Sayers, C., Kuno, H.A., Reynolds, D.: Efficient RDF storage and retrieval in Jena2. In: SWDB, pp. 131–150 (2003)
65. W3C: RDF/OWL representation of WordNet. http://www.w3.org/TR/wordnet-rdf/
66. Yars2. http://sw.deri.org/svn/sw/2004/06/yars
67. Zhu, F., Yan, X., Han, J., Yu, P.S.: gPrune: A constraint pushing framework for graph pattern mining. In: PAKDD, pp. 388–400 (2007)
68. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. **38**(2). http://doi.acm.org/10.1145/1132956.1132959 (2006)