x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases

Thomas Neumann
Technische Universität München
Munich, Germany
neumann@in.tum.de

Gerhard Weikum
Max-Planck-Institut für Informatik
Saarbrücken, Germany
weikum@mpi-inf.mpg.de

ABSTRACT

The RDF data model is gaining importance for applications in computational biology, knowledge sharing, and social communities. Recent work on RDF engines has focused on scalable performance for querying, and has largely disregarded updates. In addition to incremental bulk loading, applications also require online updates with flexible control over multi-user isolation levels and data consistency. The challenge lies in meeting these requirements while retaining the capability for fast querying.

This paper presents a comprehensive solution that is based on an extended deferred-indexing method with integrated versioning. The version store enables time-travel queries that are efficiently processed without adversely affecting queries on the current data. For flexible consistency, transactional concurrency control is provided with options for either snapshot isolation or full serializability. All methods are integrated in an extension of the RDF-3X system, and their very good performance for both queries and updates is demonstrated by measurements of multi-user workloads with real-life data as well as stress-test synthetic loads.

1. INTRODUCTION

1.1 Motivation

In the last few years, there has been rekindled and steadily increasing interest in the Semantic-Web data model RDF [23]. Important application areas like computational biology (see, e.g., *uniprot.org*) and Social-Web knowledge-sharing (see, e.g., *dbpedia.org*) prefer RDF over other data models (e.g., relational or XML) for a number of reasons. 1) RDF is a natural representation for graph-structured data, by means of subject-property-object (SPO) triples, which can be viewed as constituting nodes (S and O) and edges (P) of labeled graphs. 2) RDF spans the whole spectrum (or evolutionary path) from schema-free or highly heterogeneous collections of triples to repositories with semantic constraints and (still flexible) schemas. 3) it is easy to attach annotations to primary data or other annotations, for capturing provenance or data-quality information.

Substantial research has gone into developing concepts, algorithms, and full-fledged systems for scalable management of RDF data [1, 3, 12, 14, 25, 27]. These projects have focused on speeding up complex join queries on large RDF collections, by means of novel approaches to indexing, query processing, and query optimization. However, biological or social-network data is not static at all; these appliations need updates, too. Incremental bulk loading has been addressed in [13], based on a deferred-indexing method, but this technique does not support online updates at high rates. Also, it provides only read-committed isolation of transactions and even lacks snapshot isolation [2]. It is clearly foreseeable that applications will require online updates as well, potentially with high throughput demands and non-trivial consistency needs. This is a major challenge for the heavily read-optimized RDF engines mentioned above. The aggressive indexing or view materialization employed by these systems would incur prohibitive costs for updates, unless new ways are found for reconciling fast querying with high-throughput updates.

Consider a social-tagging community such as *librarything.com*, where hundred thousands of users share their personal knowledge about books, with book metadata, annotations (tags), ratings, comments, recommendations, cross-language links, links to external sites, etc. Also, the community members have personal profiles and social relations like friendships, recommendations for "interesting users", thematic user groups, etc. Such communities exhibit very high dynamics: new users register, users join groups, add books, add tags or comments to existing books – all this at a high rate with occasional load bursts. Moreoever, there are cross-links between different social-community sites, in the spirit of the Linking-Open-Data movement (see *linkeddata.org*), which entails the need for keeping heavily updated graphs consistent.

Although this is not a classical OLTP application, these updates do pose mission-critical consistency requirements. Simply treating each individual insertion or change of an RDF triple as an atomic step would fail to meet these demands. For example, when a new book is added, it may have to be linked to already existing versions of the book in other languages and also to external sites (e.g., amazon.com). Likewise, when a user adds tags on a book, this may be in combination with posting a comment and, perhaps, linking to another user. Together, these steps should form an atomic transaction. If the new tags and comments are influenced, in the user's mind, by other users' postings, there is even a read-

Proceedings of the VLDB Endowment, Vol. 3, No. 1

write dependency that needs to be taken into account, to avoid confusing views of the data and irritated users. While transactional isolation is an obvious mechanism, applications may often be satisfied with relaxed guarantees like snapshot isolation, which comes with better cost/throughput ratio. But specific procedures of the application may still need full serializability, and it must be possible to mix different isolation levels across the workload.

1.2 Contribution

This paper develops new methods that reconcile fast querying with efficient online updates at high rates, versioning, time-travel access, and transactions on RDF databases. This is a difficult issue already for single-user mode: aggressive indexing as used by state-of-the-art RDF engines incurs a big penalty for updates, but fewer indexes would significantly slow down queries. For multi-user mode, the additional difficulty arises that updates may cause memory, disk, or lock contention with adverse impact on parallel queries.

The methods are fully implemented by extending the open-source system RDF-3X [12, 13]. We refer to our extension as *x-RDF-3X*; its source code will be available at [17]. Our approach is inspired by the deferred-indexing approach of [13], but makes a much bigger step forward by developing a comprehensive solution that efficiently supports the full suite of desiderata for updates, versioning, and transactions.

The paper's technical contributions are the following: 1) an extended deferred-indexing method that supports *versioning* and can sustain *high update rates*. 2) a new query processing techniques that enable *time-travel querying* for arbitrary joins of SPARQL triple patterns, without unduly degrading queries on the current data. 3) a new low-overhead way of version-based, fine-grained, *transactional concurrency control*, either with snapshot isolation or with full serializability, and 4) the x-RDF-3X implementation and *performance measurements* in comparison with the original RDF-3X and a PostgreSQL-based engine.

2. BACKGROUND

2.1 Related Work

RDF engines have wide differences in the way they map SPO triples onto storage-level tables or files, how they index triples, and how they process complex queries. Early RDF systems like Jena [9, 26] and many Semantic-Web-oriented engines were not designed for scalable performance or lack support for advanced join processing. Other systems such as [4] heavily rely on manual tuning by making the right choice of materialized join views. The work by [1] opened the path to a new generation of more scalable engines tailored to RDF workload characteristics. This specific work uses minimum-width property tables (i.e., binary relations) mapped onto a column-store. Follow-up work with novel indexing and query-processing techniques includes [25, 20, 12, 14, 3]. Among these, RDF-3X [14] has achieved the best query performance in all published measurements.

Storage designs and system support for versioning has a significant history in the database literature, ranging from the seminal work on *Postgres* [21] (and the even older operating-systems-level work by [18]) to the recent project on *ImmortalDB* [10]. These systems use an *append-only storage* model, possibly with some form of garbage collection for discarding old, unwanted versions. The units of versioning are database records (relational tuples), and versions are timestamped at transaction-commit time.

The state-of-the-art ImmortalDB system has developed *lazy-timestamping* techniques to avoid repeated access to record versions within the critical path of a transaction's user-perceived response time. For indexing versions, it enhances standard B^+-trees with a time dimension and time-split strategies following earlier work on the TSB-tree [11]. There is a large literature on *temporal and multiversion indexing structures* (including the TSB-tree); an excellent survey is given by [19].

An attractive way of implementing a versioned storage system is by means of a multi-stage architecture with deferred and batched maintenance between stages. This approach goes back to classic notions of differential files, and is best exemplified in the work on the *log-structured merge tree* or *LSM-tree* for short [15]. Our multi-stage architecture has been inspired by this work.

RDF-specific versioning is addressed in the tGRIN prototype [16]. This work develops a subgraph index structure for time-annotated RDF versions, with support for temporal querying but no consideration of online updates. tGRIN processes queries by identifying the smallest subgraphs that contain query answers. The paper presents performance comparisons against Jena, Sesame, and 3Store (but not against RDF-3X), and reports response times in the range of 10 to 30 seconds for datasets with 20 million triples.

Transaction support for RDF databases is vastly unexplored, because the need for mixed workloads with concurrent queries and updates is arising only now. SQL engines support a suite of transaction isolation levels, ranging from read-committed to full serializability (SR). But record-level concurrency control along these lines would cause disastrous overhead for RDF data because SPARQL operations often touch a huge number of fine-grained RDF triples. Snapshot isolation (SI) (see, e.g., [2]) is a version-based level that can boost the performance of read-only transactions by giving them wait-free access to consistent versions. However, it is well known that SI may exhibit anomalies that could destroy the consistency of a database, with small but non-zero probability. Thus, there is a need for systems to support multiple isolation levels.

2.2 RDF-3X Architecture

As we implemented our apporach in RDF-3X [17] we give a brief overview here. RDF-3X employs an *exhaustive-indexing* approach by building clustered B^+-trees on all six SPO permutations and also all permutations of six binary and three unary projections. This design rationale is similar to the methods of [7], [22], and [25]. For the projection indexes, the missing component(s) (S, P, or O) are replaced by count aggregates, for fast statistical lookups. In all indexes, all S, P, O components are implement as integer identifiers rather than the original literals (URLs or string constants). RDF-3X uses a *dictionary* with literal-to-identifier and identifier-to-literal mappings. For fast query processing, RDF-3X uses query optimization [13] and run-time methods for sideways information passing within the resulting join trees [14].

RDF-3X had previously been extended for incremental bulk loading and a weak form of online updates (with limited performance and without transactional support) [13]. It uses a *deferred-indexing* approach with three stages: 1) updates are initially performed in *workspaces* that are private

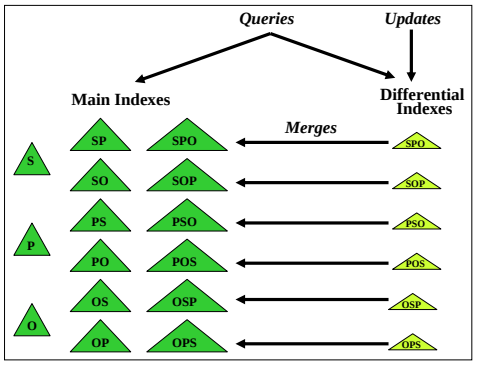

Figure 1: Differential indexes in RDF-3X

to program executions; 2) when a program terminates or issues a savepoint, the updates are merged into *differential indexes* shared by all running programs; 3) the differential indexes are periodically merged into the *main indexes* by a background process. There is one differential index for each of the main indexes: the six indexes on permutations SPO triples (SPO, SOP, ...), and the two mappings in the literal-identifier dictionary. The merge operations between storage stages are performed in a batch-oriented manner with efficient sequential access. Queries are executed against both main and differential indexes, at a moderately increased cost. This deferred-indexing architecture is illustrated in Figure 1. The current paper uses this design as a starting point, but adds major extensions for both enhanced functionality (versions, time-travel, transactions) and much better scalability and efficiency of online updates.

3. VERSION STORAGE AND INDEXING

Transactions always create new versions rather than overwriting existing triples. This forms the basis for both an efficient update mechanism based with deferred indexing, and transaction isolation that provides read access to globally consistent database snapshots. Designing a suitable storage and indexing system for RDF versions entails a number of technical difficulties: 1) keeping the *space overhead of versions* as low as possible, to minimize the potential degradation on scan and merge-join performance, 2) assigning and managing *timestamps on a per-transaction basis*, but again with as little interference with the normal update mechanisms as possible, and 3) coping with *hotspot situations* where a few triples would produce a large number of versions.

Triple Versions. Any change to an RDF triple will modify it so significantly (either S or P or O changes) that it will end up at completely different index positions (in the indexes for SPO, SOP, OPS, etc.). We therefore consider updates as pairs of deletes and inserts, and maintain versions of individual triples. For this purpose, we augment the triples by two additional fields: the *created* and *deleted* timestamps, where the latter has a null value for versions that are presently alive. The [created,deleted[interval is the lifespan of the triple version. We can reconstruct the database state *as of* a given point in time t by returning all triples for which t falls into the corresponding lifespan intervals.

This is conceptually straightforward, but we need to be careful about space consumption and devise specific compression techniques. In RDF-3X, the leaf pages in the indexes use byte-wise compression [12], where a header byte for a triple encodes the lengths of the variable parts. Since there are values that cannot be valid header bytes, we use these to encode timestamp differences in the compressed triple stream within a page. This achieves great space savings whenever subsequent triples in an index have the same or similar timestamps, which is often the case in realistic insertion load patterns. Insertions in the middle of a run require additional encoding steps. For incremental loads by a single "transaction", the space overhead is nearly zero as timestamps rarely change. This is a workload pattern that we would expect for computational-biology repositories. In all cases, decompressing during index scans has very low overhead. For a query with a given as-of timestamp, the header bytes allow us to skip irrelevant triples.

Timestamping. Ideally, timestamps reflect the commit order of transactions. Unfortunately the commit order is not known when inserting new data. This complication has led systems like Immortal DB [10] to write temporary timestamps and then update the written data again once the real timestamp is known. These timestamp adjustments can be performed in a lazy manner by a background process after the transaction commits, but still incur non-negligible costs.

We use a different approach, utilizing the fact that transactions initially perform all updates in private workspaces and merge their new triples into the differential indexes only at occasional savepoints. Each transaction is assigned a write timestamp once it starts updating the differential index, and this timestamp is then used for all subsequent operations. Triples that have not yet migrated into the differential index need no timestamp, as they are only visible to the current transaction. Ideally the migration is performed at transaction commit only, which means that the timestamps perfectly reflect the commit order and need no further updates.

In the general case where workspace updates merged into the differential indexes at savepoints before the transactions reaches its commit point (i.e., due to memory pressure), we still assign a write timestamp as outlined above and use the same timestamp for the rest of the transaction's updates, but then lose the connection to the transaction commit order. To rectify these situations, we introduce a *transaction inventory* that tracks transaction ids, their begin and commit times (BOT and EOT), the version number used for each transaction, and the largest version number of all committed transactions ($highCV\#$) at the commit time of a transaction:

transId	version #	BOT	EOT	highCV#
T_{101}	100	2009-03-20 16:51:12	2009-03-20 16:55:01	300
T_{102}	200	2009-03-20 16:53:25	2009-03-20 16:54:15	200
T_{103}	300	2009-03-20 16:54:01	2009-03-20 16:54:42	300
...	...	...	...	...

This inventory serves to efficiently decide if a transaction committed before another one. Also, it relates the relative time of transaction ordering and version numbering to wall-clock times. This is needed for supporting time-travel queries and snapshot isolation, explained in Section 4. Note that the transaction id does not need to be stored, it is mainly useful for debugging. The version number offers an implicit transaction id.

Handling Hotspots. The presented versioning techniques have very low overhead. However, there are potential problems with hotspots: triples that are inserted and deleted very

frequently. Note that this refers to insertions and deletions of exactly the same triple, with identical values for all three components (S, P, O). After a large number of inserts (a few 1000 in RDF-3X), the versions for the same triple would span multiple pages. Our remedy is the following. When we can no longer store all versions of the same triple on a single page, we create a separate B^+-tree that is linked to from the original leaf page. The main index leaf only contains the current-time version and also the oldest created timestamp and the most recent deleted timestamp of all versions. This extra information allows us to skip the triple if none of its versions qualifies for a given time-travel query, without touching the separate B^+-tree at all.

Hotspot triples should be rare, but derived index entries could become hotspots more easily. RDF-3X maintains aggregated-triples indexes for the 2-projections (SP, PS, SO, etc.) and 1-projections (S,P,O) of all triples. The entries in these indexes change at a much higher rate. For example, whenever a new triple is inserted for an existing SP value pair but with a new O value, the corresponding SP index entry is updated and would trigger the creation of a new version of the index entry. We therefore decided that projection/aggregation indexes are not versioned at all; we only maintain their present-time part. The indexes are used for selectivity estimation, for which we can tolerate that we do not have as-of statistics and instead use present-time statistics, and for faster scans when a sub-query does not need all three SPO components. The latter entails some modification to the query optimizer. When an execution plan uses a particular projection/aggregation index, the plan is adjusted so as to scan the index in both main-indexing and differential-indexing stages (see Section 2.2) and merge these two streams on the fly. For the differential-indexing part, this is actually implemented by computing the count-aggregates via scanning the corresponding full-triples index. This is performed only for the actually required projection – for example, for SP* from SPO, but not PS* etc. unless needed at this point.

4. TIME-TRAVEL QUERIES

Query Processing. With the version storage of x-RDF-3X developed in this paper, time-travel queries, i.e., queries running on database states as they were in the past, are amazingly easy to implement. The main technique is to evaluate queries – by scanning indexes, merging triple streams, etc. – with a *timestamp test* based on the query transaction's *as-of timestamp* T_R. This way, we can identify all relevant versions on the fly. This way, we can support time-traveling for the full set of SPARQL triple patterns (manyway joins) and filter conditions offered by RDF-3X. The query optimizer first generates plans suitable for current-time queries and then adds version-number filters and other adjustments to ensure the proper timestamping of the query results. In particular, whenever the current-time plan makes use of projection/aggregation indexes, the optimizer adds scans on appropriately chosen full-triples indexes, aggregates the triples on the fly, and merges these streams with the ones from the current-time projection/aggregation indexes (see also Section 3).

Timestamps and Version Numbers. Recall from Section 3 that we create version numbers that are not necessarily in synch with the transaction commit order. This is a design decision for higher update throughput, as it avoids having to touch a stored version a second time at transaction time or later. So in contrast to the design of ImmortalDB [10], which had different design criteria and constraints, we minimize the version overhead during normal operation, but this now entails complications for time-travel queries by facing the following problem of *out-of-order version numbers*.

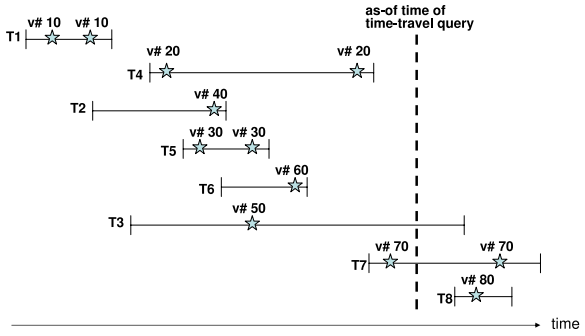

Figure 2: Out-of-order version numbers

Between two transactions, the one that committed earlier may have used higher (i.e., "younger") version numbers. So when scanning indexes and seeing a version number, the query execution has to look up the transaction inventory and perform a suitable test to check if the encountered version number belongs to a transaction that is a) in the time-travel scope of the query and b) indeed committed. This situation is illustrated in Figure 2. Here the time-travel query must not see the - then uncommitted - versions with version numbers 50 and 70. And for each accessed triple, it needs to determine among the versions numbered 10, 20, 30, 40, and 60 those that were committed last before the time-travel query's as-of timestamp. This could be version number 20 for a triple x, which also has versions numbered 10 and 40, and version number 30 for a triple y, which has another version numbered 40. When accessing triples in a time-travel transaction for timestamp T_R, we check if the versions fall into the range of transactions active at T_R. If yes, we maintain an in-memory cache of active transactions to see if have to ignore the triple or not. Otherwise, we can decide globally if a triple is visible (beeing either too new or old enough for sure).

5. CONCURRENCY CONTROL

5.1 Snapshot Isolation (SI)

The versioning of RDF triples provides snapshot isolation (SI) almost for free. We implemented SI via version numbers for reads and write locks to satisfy the write-set disjointness of concurrent transactions. The fine-grained nature of RDF raises issues regarding the overhead of lock management; we discuss them in Section 5.3 in the context of serializability which requires both read and write locks for update transactions (but no locks at all for read-only transactions). We use version numbers for determining the most recent committed version that a read access should see. We need handle the complication that version numbers do not necessarily reflect the commit order of transactions. We described our solution in Section 4.

5.2 Serializability (SR)

Usually SR on top of SI is achieved by using SI for read-only transactions, and adding locking for read-write transactions. This is also our solution. What complicates the situation for RDF is that the database may consist of billions of tiny

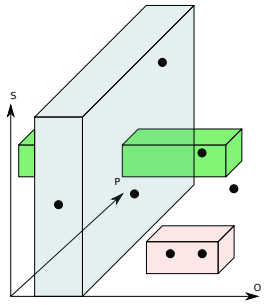

Figure 3: Geometric interpretation of RDF locks

triples. Locking individual triples is not practically viable because of the potentially huge memory demand for the lock table, and lock escalation is likely to lead to deadlocks due to the lack of natural ordering within the RDF graph. Next, we show how to bound the overhead of predicate locks without unduly restricting the degree of concurrency.

5.3 Predicate Locking

Internally, RDF-3X maps all SPO triples into integer triples by using a dictionary for literals. This means that all triples form points in $\mathbb{N}^3$. SPARQL queries consist of triple patterns, where each triple pattern uses constant values or variables for the S, P, O positions. Thus, each triple pattern denotes a 3-dimensional axis-aligned rectangular box in $\mathbb{N}^3$ data space, as illustrated in Figure 3. By keeping track of all boxes scanned or modified by a transaction and testing for overlaps, we can detect all conflicts between transactions. Our lock granularity are essentially these boxes, and we could use a standard locking protocol on this basis. To improve concurrency we use a conflict-ordering protocol (see Section 5.4), but for now it is sufficient to think of the boxes as regular locks. The locks are represented by projections of $\mathbb{N}^3$ such as $x_1 = (101, 11, *)$, $x_2 = (200, 11, *)$, $x_3 = (200, *, 5000)$, $x_4 = (*, *, 6000)$ where the conflicting pairs are (x_1, x_4), (x_2, x_3), and (x_2, x_4). Because transactions with many insert operations often write-lock sequences of consecutive projections by the nature of assigning new identifiers, we can also combine contiguous boxes into 3D range predicates such as $(101, 11, 1..7)$ or $(301..450, 11, *)$.

Tracking the locked boxes is light-weight for queries; a SPARQL query typically consists of a few or perhaps tens of patterns, much less than the number of examined triples. Update transactions are much more expensive, in particular if they perform (incremental) bulk insertions. In principle, one could lock a tiny box for each inserted or modified triple, but if a transaction loads millions of triples (in parallel to ongoing queries) this is no longer practically viable. Instead, we dynamically construct coarser-grained locking boxes on demand, using the following procedure.

Initially, all updates are stored in the per-transaction workspaces. At commit time, the transaction must request write locks for all its deferred updates before merging them into the shared differential indexes. We give each transaction a budget in terms of a maximum number of write locks it can use (e.g., 1000), and then construct locking boxes that cover all updated triples and as little non-modified space of $\mathbb{N}^3$ as possible.

In this covering of updated data with a limited number of boxes, we aim to minimize the unnecessarily locked space as this may cause "false conflicts" that degrade concurrency. We conjecture that this bounded-cover problem is combinatorially hard (related to NP-hard problems liket set cover – but we do not have any theorems); so we use heuristics. Similarly to the related but different problem of optimizing R-tree splits of bounding boxes, we aim to minimize the *volume* of the "falsely locked" space.

We define the volume of boxes in $\mathbb{N}^3$, by scaling the ranges of existing identifers in each of the S, P, and O dimensions to length 1 and assume a 3D grid of discrete cells on this uniformized space. As a result, the volume sub-cubes (or subplanes for projections) reflect the total number of distinct S, P, and O values in a data-adaptive manner. If a box comprises many consecutive P values and there are much fewer P values than S or O values overall, then the box is made more streched in the P dimension and thus given higher volume. This is exactly the desired effect of treating boxes with higher conflict potential in a finer-grained manner.

Ideally we would compute, for each transaction, the minimum-volume cover of its locked boxes with the allowed bound for the number of locks, but run-time techniques like this would unduly delay the response times of interactive transactions. Therefore, we settle for a faster heuristic approach, which resembles lock de-escalation techniques [6] (decomposing coarser lock granules into finer ones as conflicts arise) but operates on 3D range predicates. Initially, we place all modified data points in one big lock cube. Then, we repeatedly examine the lock cube with the largest volume and split it into two smaller sub-cubes using the algorithm shown in Figure 4. The algorithm examines the input boxes with regard to all three dimensions d (d is S, P, or O), finds the largest "falsely locked" gap along these dimensions, and then computes the resulting minimum-volume boxes if one would partition the lock along this dimension. It then picks the split dimension such that the resulting total volume for all locks is minimal. This process is repeated until the maximum number of allowed locks is reached. The algorithm has computational complexity $O(n \log n)$ where n is the number of a transaction's updated triples, or $O(n)$ if the triples are already sorted in each dimension. Our measurements confirm that this is a practically viable method.

The currently held locks can be organized efficiently in a k-d tree [5]. As all predicates form axis-aligned bounding boxes in the data space, we can test for lock collision by finding all intersecting boxes and checking for conflicting lock modes. Note that conflict checks are relative infrequent compared to record level locking. We only have to check the predicates in each query, which are few, and the covering boxes for update transactions, where we limit the number of boxes. The overall overhead for lock management is therefore low.

5.4 Conflict Ordering

When detecting a lock conflict, i.e., when one transaction tries to lock a region that intersects with an existing (and mode-wise conflicting) lock, there are different strategies to handle the situation. The simplest one would be to abort the lock-requesting transaction. Alternatively, one could use two-phase locking, where the second transaction waits until the first transaction releases its lock (at its commit time). This standard technique is unnecessarily restrictive in our setting. As we are operating on versioned data, there is no need to immediately block a transaction.

Instead, we ensure that the executed transaction schedule

splitLock($cube = [S_{min} : S_{max}]x[P_{min} : P_{max}]x[O_{min} : O_{max}]$)
for each $d \in \{S, P, O\}$
  $b_d = 0$, $s_d = \arg\min_{t \in cube} t.d$, $l = s_d$
  for each $t \in cube$ **ordered by** $t.d$
    if $|t.d - l.d| > b_d$
      $b_d = |t.d - l.d|$, $s_d = t$
    $l = t$
  C_d^1 =minimum bounding box for $\{t \in cube | t.d \leq s_d\}$
  C_d^2 =minimum bounding box for $\{t \in cube | t.d > s_d\}$
$d = \arg\min_{d \in \{S,P,O\}} |C_d^1| + |C_d^2|$
return (C_d^1, C_d^2)

Figure 4: Lock Splitting Algorithm

is serializable by dynamically building a *serialization graph (SG)* [24]. Initially, all transactions can be arbitrarily ordered. As we observe conflicts, we add order-constraining edges to the SG and check that the graph remains cycle-free:

- if a read lock of transaction T_1 conflicts with a previous write lock of transaction T_2, we order T_1 before T_2 (reading an earlier - already committed - version);
- if a write lock of transaction T_1 conflicts with a previous read or write lock of T_2, we order T_1 after T_2.

If this reordering is not possible (either because it creates a cycle in the SG or if T_1 would have to be ordered before a committed transaction because of other edges), we abort T_1. Note that before the commit point this reordering is purely conceptual: we ensure that the schedule is still serializable, but we do not cause any transaction waits. Only at commit point it may be necessary to wait. When a transaction T_1 has been ordered before T_2, T_1 must commit (or abort) before T_2 is allowed to commit.

This SG-testing protocol greatly improves throughput and the feasible concurrency, compared to a conservative, real-locking protocol. The only disadvantage is that we must possibly remember locks even after a transaction's commit (which is a common feature of graph-testing concurrency control [24]). We can drop locks from the lock table (and transactions from the SG) only when all concurrently running transactions are finished. Otherwise it could happen that during a long-running transaction a shorter transaction has already commited (and discarded its locks) before a conflict cycle could arise. But this addition to the lock-table and SG bookkeeping is straightforward and light-weight.

6. EVALUATION

We implemented all data structures and algorithms for versioning, time-travel queries, and transactional isolation in the RDF-3X engine [17]. Note that this software does not provide any update capabilities other than bulk-import. We denote our extended version as *x-RDF-3X*. We studied the efficiency of our methods by running different types of benchmarking workloads, comparing run-times averaged over a large number of transactions. 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 2.6.31 kernel.

Competitors. We ran the same workloads on two opponents that represent two classes of alternative architectures: 1) As a *triple store based on a relational engine*, we used *PostgreSQL 8.4* as described in [12]. PostgreSQL has built-in versioning and uses snapshot isolation for transactional concurrency control. 2) As a *native RDF system* we used *Jena* (SDB 1.3.0, Derby 10.5.3.0) [9], which also provides transaction support. Jena is widely considered the most mature system in this category. We also ran experiments with other RDF systems (Mulgara, Yars2, Sesame 2 – tGRIN is not publicly available), but encountered significant performance limitations.

Data. As there is no established benchmark for RDF updates, we constructed our own workloads based on data from a partial crawl of the LibraryThing book-tagging Web site (www.librarything.com). RDF triples capture the tagging activity of this community: when a user u annotates book b with tag t, this (u, t, b) combination forms an SPO triple [12]. This results in a large number of distinct P values and in skewed frequency distributions. Additional RDF triples are formed by user or book metadata, friendship links, etc.

Workload. For constructing a mixed read-write workload, we assume that when a user tags books, she first examines the existing tags, thinks a moment about her choice of tags, and then enters these tags - everything within a single transaction. (This interaction template follows the UI of the LibraryThing Web site). By running many of these transactions either sequentially or in parallel, we mimic the natural growth of the dataset and naturally introduce data access skew and hot spots. We used this transaction type as a building block for constructing a three-phase lifecycle: 1) bulk-loading, 2) growing the data by transactions, and 3) querying the existing data. Each phase is measured separately.

Bulk-load phase. We took one million random triples from the LibraryThing dataset and split it into two parts: the first part consists of half of the user-tag-book triples and all other triples with user or book metadata. We measured the time to complete the data import.

Online-update phase. We took the remaining half of the user-tag-book triples, grouped them by user-book pairs, and ran these user-book-specific updates as a single transaction as described above. We ran these transactions in multi-user mode, using an open system model [8] to reflect a realistic workload with many clients. Transactions arrive at the system by a Poisson process with rate λ. This load model creates occasional load bursts, which are more realistic than a closed system model with a fixed multiprogramming level. We did not exert any admission control, and left choosing the number of system-internal worker threads to the systems under test. We varied the arrival rates, and report numbers for the maximum sustained throughput and the average response time at particular arrival rates. The user think time inside the transactions was set to one second.

Query phase. We ran a series of time-travel queries on the complete one-million triples database as created by the first two phases. We adopted the queries of [12], but evaluated them as of a randomly chosen timepoint in the past. We measured these query run-times in single-user mode, with cold and with warm caches.

We also measured performance with all 35 million triples from the dataset instead of the one-million sample. This had little effect on the transaction rates of PostgreSQL and x-RDF-3X (preserving their relative performance ratio), but Jena could not load this large dataset. We thus report only on the one-million-sample results.

6.1 Bulk Load and Sequential Updates

The initial bulkload used the import functions provided by each of the competitors. We give the times for loading

system	total time [s]	triples/second	space [MB]
RDF-3X	36	23502	113
x-RDF-3X	55	15383	113
PostgreSQL	91	9228	373
Jena	929	910	375

Figure 5: Performance for the initial bulkload

system	total time [s]	triples/second
x-RDF-3X	10.34	14876
PostgreSQL	189.91	810
Jena	193.70	794

Figure 6: Performance for sequential updates

half of our test data in Figure 5, and also give the resulting database sizes. RDF-3X without update support performs best, with x-RDF-3X following in second place. Note that our performance loss compared to the original RDF-3X is somewhat misleading. We do have to pay some overhead by our modifications to the RDF-3X engine, but surprisingly most of the performance difference does not come from these but from input parsing. For larger datasets where the import is dominated by I/O costs, both systems perform nearly the same; we verified this by bulk-loading all 35 million triples into both systems. PostgreSQL performs slightly worse, although it did not have to do any complex input parsing; we used the RDF parser from RDF-3X to generate text data that could be directly imported into PostgreSQL, and did not count the RDF-to-text conversion (which took an additional 4s when using the RDF-3X parser). Jena is an order of magnitude slower than the other systems. The reasons are not fully clear to us.

We also studied the run-time of basic update operations. After completing the initial bulk-load, we added the remaining half of the one-million triples dataset by sequentially executing update operations. This measures elementary update costs without any concurrency. The results are shown in Figure 6.

6.2 Concurrent Read-Write Transactions

We employed an open system model with read-write transactions as explained above. We gradually increased the arrival rate λ of the transactions from low values where transactions run more or less sequentially (because of long $1/\lambda$ inter-arrival times) to high λ values until systems could no longer sustain the throughput. Beyond this point, response times grow dramatically and would approach infinity. This way we could determine the maximum sustained throughput of each system and the response times at different throughput levels. For each parameter setting we ran the experiment with 10,000 transactions and measured the response time of each transaction. We also included an extreme stress-test where all transactions arrived instantaneously at the very beginning of a run (denoted as "$\lambda = \infty$").

The results are shown in Figure 7. Initially x-RDF-3X, PostgreSQL, and Jena perform similarly, as all systems are still lightly utilized for low arrival rates up to 5. Average response times in this load range are around 2 seconds. As the load increases the differences become much larger. PostgreSQL becomes saturated at an arrival rate of 25 where the average response time goes up to 17 seconds, and degrades for $\lambda = 50$. x-RDF-3X handles this throughput of 50 transactions/second very well, and even at 200 transactions/second still has response times around 2 seconds. In terms of average response times at throughput $\lambda = 100$, x-RDF-3X outperformed PostgreSQL by a factor of 40. Jena exhibits severe problems with this multi-user workload. It performs well only for very low arrival rates, but saturates already at 5 transactions per second and crashed for higher λ.

Overall, x-RDF-3X could sustain a throughput of 200 transactions/second with hardly any slow-down of its near-sequential response times of 2 seconds. If we view 2 seconds average as a reasonable bound for user-tolerated response time on an interactive multi-user system, PostgreSQL could meet this standard only up to about 20 transactions/second, and Jena failed completely at around 5 transactions/second at best. The response time gains of x-RDF-3X at high throughput levels were one or two orders of magnitude.

6.3 Time-Travel Queries

In the previous workload, the queries are relatively simple lookups. To study the overhead induced by the tuple versioning, we also measured the more complex LibraryThing queries from [12] but made them time-travel queries that return the results as of a given timepoint in the past. As only x-RDF-3X supports time-travel querying, the competitor systems just ran regular queries (as of now). For x-RDF-3X, the overhead that we wanted to quantify is the extra cost incurred by reading but ignoring versions that are newer than the query timestamp.

The results are shown in Figure 8. Both PostgreSQL and Jena perform poorly on these complex queries. RDF-3X and x-RDF-3X have similar performance, with a mild slowdown of x-RDF-3X in the cold-cache case. This is mostly because the time-travel queries cannot use any projection/aggregation indexes and thus require more I/O. However, the overhead is small and the query run-times of x-RDF-3X are nearly as good as those of the original RDF-3X (despite the advantage of RDF-3X that it does not keep versions and runs regular instead of time-travel queries).

6.4 Locking Overhead

Finally, we studied the costs of serializability (SR). We ran the experiment from Figure 7 with different locking implementations, namely, without any locking (only short-term latching), with our predicate locking method from Section 5.2, and with traditional record locking. Up to an arrival rate of 20 transactions per second, there was no noticeable overhead induced by our locking scheme (i.e., by SR) compared to SI. Both average and median response times were unchanged, the 95% quantile increased by 3% due to lock waits. In an extreme stress test, we set the arrival rate to infinity so that all transactions arrived at once. In this case, the average response time did increase, but even then only by a negligible factor of 0.7%. Record locking, on the other hand, led to disastrous processing times for two reasons: first, because many thousands of locks have to be managed, and second because deadlocks (and thus transaction re-starts) are very frequent. Traditional locking schemes are not suited for the fine-grained nature of RDF data. Our predicate locking method works well, and SR is affordable with negligible impact on the overall system performance.

7. CONCLUSION

This paper has described how to extend a query-centric RDF engine into a system with full-fledged support for updates, versioning, and transactions. Although we started

$\lambda[s^{-1}]$	3.33	5	10	12.5	16.66	25	50	100	200	∞
x-RDF-3X	1995.3	2023.1	1998.9	2029.2	2027.9	2046.0	2037.5	1991.3	2008.9	16064.2
PostgreSQL	2026.2	2053.7	2045.5	2083.1	2075.4	2040.5	17455.5	63201.7	83270.3	102757.0
Jena	2119.5	2136.5	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Figure 7: Average response times at different arrival rates λ

	A1	A2	A3	B1	B2	B3	C1	C2	geom. mean
cold caches									
RDF-3X	0.03	0.05	0.12	0.06	0.03	0.25	0.06	0.11	0.06
x-RDF-3X	0.04	0.05	0.20	0.09	0.04	0.27	0.06	0.13	0.08
PostgreSQL	9.31	9.16	19.08	0.64	3.88	> 10min	6.95	8.12	>10.39
Jena	4.48	5,52	12.07	3.85	7.73	> 10min	>10min	385.91	>32.46
warm caches									
RDF-3X	<0.01	<0.01	0.01	<0.01	<0.01	0.03	<0.01	<0.01	<0.01
x-RDF-3X	<0.01	<0.01	0.01	<0.01	<0.01	0.04	<0.01	<0.01	<0.01
PostgreSQL	5.39	5.31	12.02	<0.01	0.92	> 10min	4.26	4.08	>3.67
Jena	2.31	2.25	10.90	2.11	6.44	> 10min	>10min	275.16	>22.92

Figure 8: Average run-times for time-travel queries on x-RDF-3X, regular queries on the other systems

with the "penalty" of an aggressive indexing scheme on the SPO triples, we managed to design and implement very efficient mechanisms for both incremental bulk loading and online updates. Our measurements have shown that the extended engine can sustain high throughput in multi-user mode for mixed read-write transactions, with both snapshot isolation and serializability. The opponents in our experimental comparison were outperformed by a very large margin, with orders-of-magnitude differences at high levels of resource contention. Our experiments also demonstrated that transactional isolation has low overhead when based on a judiciously designed versioning system and implemented with our RDF-specific predicate-lock management and a light-weight serialization-graph protocol. Thus, our approach successfully combines the virtues of fast querying, strong update performance, and the beauty of automatic, easy-to-program consistency guarantees by full-fledged transactions.

8. REFERENCES

[1] D. J. Abadi et al. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[2] A. Adya, B. Liskov, and P. E. O'Neil. Generalized isolation level definitions. In *ICDE*, pages 67–78, 2000.

[3] M. Bröcheler, A. Pugliese, and V. S. Subrahmanian. Dogma: A disk-oriented graph matching algorithm for rdf databases. In *International Semantic Web Conference*, pages 97–113, 2009.

[4] E. I. Chong et al. An efficient SQL-based RDF querying scheme. In *VLDB*, pages 1216–1227, 2005.

[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001. 2nd Edition.

[6] G. Graefe. Hierarchical locking in b-tree indexes. In *BTW*, pages 18–42, 2007.

[7] A. Harth et al. YARS2: A federated repository for querying graph structured data from the web. In *ISWC/ASWC*, pages 211–224, 2007.

[8] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley, 1991.

[9] Jena: a Semantic Web Framework for Java. http://jena.sourceforge.net/.

[10] D. B. Lomet et al. Transaction time support inside a database engine. In *ICDE*, page 35, 2006.

[11] D. B. Lomet and B. Salzberg. The performance of a multiversion access method. In *SIGMOD*, 1990.

[12] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.

[13] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 2009.

[14] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, 2009.

[15] P. E. O'Neil et al. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.

[16] A. Pugliese, O. Udrea, and V. S. Subrahmanian. Scaling rdf with time. In *WWW*, pages 605–614, 2008.

[17] RDF-3X. http://www.mpi-inf.mpg.de/~neumann/rdf3x.

[18] D. P. Reed. Implementing atomic actions on decentralized data. *TOCS*, 1(1):3–23, 1983.

[19] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2):158–221, 1999.

[20] L. Sidirourgos et al. Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.

[21] M. Stonebraker. The design of the postgres storage system. In *VLDB*, pages 289–300, 1987.

[22] O. Udrea, A. Pugliese, and V. S. Subrahmanian. GRIN: A graph based RDF index. In *AAAI*, 2007.

[23] W3C: Resource Description Framework (RDF). http://www.w3.org/RDF/.

[24] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[25] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[26] K. Wilkinson et al. Efficient RDF storage and retrieval in Jena2. In *SWDB*, pages 131–150, 2003.

[27] Yars2. http://sw.deri.org/svn/sw/2004/06/yars.