

# A Concurrency Control Protocol for Multiversion RDF Datastores (Discussion Paper<sup>1</sup>)

Andrae Muys  
*andrae@netymon.com*

Netymon Pty Ltd  
and  
School of ITEE  
The University of Queensland

## RDF and Traditional Concurrency Control

XA1 uses multiversion concurrency control with a single global write-lock to ensure serialisability of transactions. One of the goals of XA2 is to support concurrent writes, which therefore requires that we address the issue of multiversion serialisability. There is a substantial literature discussing serialisability, traditional approaches include 2-phase-locking, and timestamp-ordering[Bernstein87]. While these techniques have been extended to support multiversion databases[Bober92,Phatah99], the databases addressed in the literature have been either Relational or Object DBMS's. An RDF datastore is fundamentally a graph, and therefore violates the assumptions underlying much of the pre-existing work on concurrency control.

Traditional concurrency control assumes the existence of 'cells' that are subject to read, write, create and delete operations — in the relational literature create is handled implicitly as a write to a previously unused cell id, and delete as just another write. The critical assumption traditionally made is that the cells are independent — that a write to  $y$  does not invalidate a read from  $x$ . With an RDF store a common fundamental read-primitive is what SPARQL refers to as 'Basic Graph Pattern' (BGP) and iTQL refers to as a 'Match Pattern', which is unified with the RDF graph to produce a result. As unification applies to the entire graph, any modification of graph potentially alters the result of a read operation and therefore the smallest 'cell' we can potentially consider in RDF datastore concurrency control is the graph itself. In fact because a graph name returned by one pattern can be used to specify the target graph of another pattern, our theory must consider all the accessible graphs a single 'cell' when considering traditional techniques. It is worth noting here that any set of named 3-graphs can be represented as a single 4-graph by incorporating the graph-name as an extra vertex in each edge of each graph. So for the purposes of our discussion we will ignore the distinction between sets of 3-graphs, and 4-graphs.

## Traditional Relational Concurrency Control

As long as we are forced to treat an RDF datastore as a single cell the traditional relational concurrency control protocols devolve to pathological cases. With only one cell multiversion 2-phase-locking (MV2PL) is operating with only one lock, and consequently devolves to a snapshot isolation with a single global write-lock (which is what XA1 supports); and multiversion timestamp ordering (MVTO) devolves to aborting any transaction that would have blocked under MV2PL. The traditional protocols widely used by relational databases are therefore provide very little concurrency when applied to an RDF datastore.

---

<sup>1</sup> Presented at the Topaz / Public Library of Science, Mulgara Workshop, San Francisco, March 2008

## Traditional Object Concurrency Control

Linearizability[lin], often used to reason about correctness in concurrent object systems, requires us to define a *representation invariant*, and an *abstraction function*. For RDF these are defined by the Abstract Syntax and Semantics but in practice the representation invariant is trivial — any set of 3-tuples that meet RDF's restrictions on subject, and predicate resources is a valid representation. In practice the abstraction function simply maps the 2nd element of each 2-tuple to a predicate: P; each 3-tuple to a simple logical proposition: P(subject, object); and the set of 3-tuples to the disjunctive sum of the elements of the set. With only a single object (cell), linearization has similar issues of aborting writes as MVTO, although it has the advantage of permitting guarantee blind-writes will not abort. The underlying concept of linearizability, that operations are locally deemed to have occurred at a single point in time, fits well with the concept of a multiversion store.

## XA2 and Predicate Locking

Clearly we have to find some way to avoid treating the entire graph as a single entity for concurrency control purposes. We can achieve this if we introduce the concept of a *predicate-lock*[Eswaran76]. A predicate lock allows a transaction to lock a portion of the potential table defined by a given predicate. This lock needs to cover both the data actually present in the table as well as any data that may be inserted that would also satisfy the predicate (referred to as phantom data). In the case of RDF an obvious candidate for treatment as a predicate is the Basic Graph Pattern. Under this conception, a BGP would only be affected by an update if the update changed the result of the BGP under unification. Formally we can express this as: given a graph  $G$  accessed by a transaction  $T_0$ , and an updated graph  $G'$  generated by  $T_1$  then  $BGP(G) \neq BGP(G') \vdash T_0 \ll T_1$ .

RDF semantics specify an infinite external vertex set, therefore updates are strictly in the form of extensions or subsetting of the graph's edge set. BGP unification is also defined solely in terms of the edge set, and therefore graph pattern unification is monotonic with respect to either insertion and deletion of edges when treated in isolation. This means that determining if an update produces a graph that alters the patterns result is equivalent to determining if the pattern unifies against the update in isolation. So formally, given an update  $g$  to a graph  $G$  generating an updated graph  $G'$  then  $BGP(g) \Rightarrow \emptyset \vdash BGP(G) =$

$BGP(G')$ . This test allows us to ignore the NP-complete issue of identifying arbitrary conflicting phantom data, and instead to implement predicate-locks as tests against the actual updates that need to be serialised. Given a  $O(\log n)$  unification algorithm for basic patterns[kow] this makes identifying conflicting transactions tractable.

## XA2 and Deadlock Resolution

Regardless of what concurrency control protocol we use, we still face the issue of deciding how to resolve the inevitable conflicts that are detected. As a multiversion store we can exploit the implicit snapshot isolation[Berensan95] to linearize all read-only operations at their inception which ensures they can never be involved in a serialization conflict. The remaining usecases in the standard workload are:

- A moderate number of very small read/delete/insert updates to a single graph, analogous to a row/field update in an rdbms.
- A moderate number of small blind-inserts into a non-empty graph, analogous to adding an entity in an ER-model.

- A small number of very small read/delete updates to a single graph, analogous to deleting an entity in an ER-model
- A small number of very large blind inserts into newly created graphs, which are a result of bulk-loads of data into a graph.

Ultimately of two conflicting transactions one of them will have to abort and restart. We must therefore define an algorithm that inflicts the least pain on the user. Specifically we should aim to minimise the amount of wasted work resulting from an abort. This leads us to prefer to abort smaller transactions over larger ones. As both the usecases that involve reads are also small, we should prefer to invert the traditional protocols and prefer to abort transactions with conflicting reads over conflicting writes.

### **XA2 and Optimistic Concurrency Control**

Unfortunately even with access to efficient predicate locking, the cost of performing a lock validation at each write will be prohibitively expensive for all but the smallest stores. It is worth noting that in a large store the average query rarely contains single BGPs, or update operations that apply to more than a single 3-graph. In addition with snapshot isolation reads do not need to be validated unless and until their transaction has attempted a write. This suggests that we can substantially reduce the overhead of serialization by adopting an optimistic protocol. We therefore propose to develop a multiversion optimistic concurrency control (MVOCC) based on the single-version protocol proposed by Kung and Robinson[Kung81].

They suggest splitting each transaction into three phases *Read*, *Validate*, and *Write*. In the read-phase the transaction is permitted to perform any read/write combination it desires, but all writes take place on local copies. During validation the query is validated against all intervening updates that belong to transactions that have previously finished their read-phases. For this purpose, in SVOCC the scheduler must keep track of the order in which transactions enter the validation phase in order to identify which transactions each transaction must validate against. Any transaction that successfully completes validation is permitted to enter the write-phase, where any updates are made persistent. Transactions that fail validation are aborted and retried. This protocol has no risk of deadlock, although there is an obvious risk of starvation, specifically high volume reads can stave a large write.

### **Adapting Single-Version Optimistic Concurrency Control to a Multiversion Store**

In adapting SVOCC to a multiversion protocol we make a number of changes.

First, as mentioned previously, by moving to a multiversion protocol we remove the need to include read-only queries from the validation protocol. They can continue to read their original version of the graph, and therefore can be serialised at their commencement[Agrawal91]. This alone substantially reduces the risk of starvation.

Second, as we are a multiversion store we already assign version numbers to every transaction. Consequently we replace most of the transaction numbers used by SVOCC by version numbers. It is worth noting here that, although it may be counter intuitive, it is worth adopting the numbering optimisation described in Kung where transactions do not obtain a version when they are started, but rather at the first read or (in the case of blind-writes) at commit. This minimises the number of intervening writes that must be serialized against.

Third, as discussed above, we prefer to abort read-write over write-only transactions. The validation phase is therefore defined in terms of identifying any reads within a transaction that conflict with intervening committed writes.

### **The Multiversion Optimistic Concurrency Control Protocol**

When a transaction starts it does not obtain a phase, but only registers itself with the scheduler.

If the first operation of a transaction is a read, it:

1. Obtains a phase
2. Registers with the scheduler as Read-Only
3. Remembers the Basic Graph Patterns used in the query.

Subsequent queries simply result in extending the remembered BGP-set.

If the first operation of a transaction is a write, it:

1. Performs the write outside of the store as a delta
2. Registers with the scheduler as Write-Only

Subsequent writes can be merged with the existing write, ultimately any series of inserts or deletes can, provided there are no intervening reads, be collapsed into a single insert followed by a single delete or visa versa.

If a read-only transaction performs a write, it:

1. Performs the write outside the store as a delta
2. Registers with the scheduler as Read-Write

If a write-only transaction performs a read, it:

1. Obtains a phase
2. Registers with the scheduler as Read-Write
3. Remembers the BGP's used in the query.

In both cases subsequent queries must return the merge of the unification of the pattern against the queried phase, and the unification of the pattern against the delta.

An attempt to validate a read-only transaction is a no-op.

When a read-only transaction commits, it:

1. Releases the phase
2. Notifies the scheduler it is complete
3. Releases any resources held, specifically the BGP-set.

An attempt to validate a write-only transaction is likewise a no-op:

When a write-only transaction commits, it:

1. Obtains a new phase number
2. Registers the deltas with the scheduler as a validation target
3. Merges deltas with the store

All subsequent attempts to obtain a phase will return the new merged phase until it is itself superseded.

A read-write transaction is validated, by:

1. Foreach committed delta  $\delta$  newer than the originating read
  - 1.1. Foreach BGP B in remembered BGP-set
    - 1.1.1. If  $B(\delta) \neq \emptyset$  abort transaction
2. Proceed as a write-only commit

### **Additional MVOCC Optimisations**

The validation optimisation envisaged by Kung and Robinson also applies to MVOCC. It is possible to run the validation in parallel provided only one transaction is permitted to leave validation at a time, and any remaining transactions must subsequently validate against the leaving transaction before leaving. Additionally it is possible to provisionally validate a transaction against any other incomplete transaction, and more importantly to validate the existent part of an incomplete transaction against a complete transaction. This latter is important as it permits transactions that are invalidated while still in their read-phase to be restarted while wasting minimal work.

If we are willing to accept a delay to write-only transactions it maybe possible to reduce the number of conflicts by serializing the write-only transactions after any existent read-write transactions[Agrawal91]. It therefore may be worth considering having write-only transactions participate in the validation phase by delaying their commit to give currently completing read-write transactions a chance to complete without the risk of being invalidated by the update.

### **Alternative levels of Consistency**

A benefit of the above changes is that it is localised to each transaction. Each transaction is provided with sufficient information to permit it to determine for itself if it can complete serialisably. However its decision remains independent of any other transaction, and therefore it becomes possible to support concurrent execution of transactions with differing consistency guarantees. The only overhead imposed on a transaction that declines to participate in the protocol is that it maintain update deltas, which is required by the use of multi-version isolation anyway.

The standard treatment of alternative levels of isolation and consistency is Berenson et al.[Berenson95] who classify them in terms of prohibited phenomena and anomalies. Phenomena are defined as sequences the *might* lead to an inconsistency; Anomalies as se-

quences that *have* lead to an inconsistency. Also referred to as the *broad-interpretation* of a prohibition and a *strict-interpretation* respectively. The phenomena and anomalies identified by Berenson are summarised below:

**A1 (Dirty Read)**

w1[x]...r2[x]...a1 read is invalid as T1 has aborted

**P1 (Dirty Read)**

w1[x]...r2[x]... read might be invalid if T1 aborts

**A2 (Non-Repeatable Read)**

r1[x]...w2[x]...c2...r1[x]... write has violated read-isolation of T1

**P2 (Non-Repeatable Read)**

r1[x]...w2[x]... write may have violated read-isolation of T1

**A3 (Phantom Read)**

r1[P]...w2[y in P]...c2...r1[P] write has violated query-isolation of T1

**P3 (Phantom Read)**

r1[P]...w2[y in P]... write may have violated query isolation in T1

**P4 (Lost Update)**

r1[x]...w2[x]...w1[x]...c1... T2's write has been lost, violating write-isolation

**A5A (Read Skew)**

r1[x]...w2[x]...w2[y]...c2...r1[y]... write has violated any constraint C(x,y) seen by T1

**A5B (Write Skew)**

r1[x]...r2[y]...w1[y]...w2[x]... writes have violated any constraint C(x,y)

It is worth considering the level of isolation provided by a multiversion store that ignores concurrency control. As all reads are performed on committed versions, and all writes are performed on branches, **A1**, **P1**, **A2**, **P2**, **A3**, **P3** and **A5A** are automatically prohibited. However **P4** can still occur if two transactions attempt to insert/delete the same triple concurrently. To address this we can add a trivial reconciliation algorithm, simply by giving precedence to inserts over deletes we can insure we never lose data – and for many applications that is sufficient. Alternatively we could validate inserts against deletes using the identity  $A \bowtie B = \emptyset \vdash A|B \cong B|A$ . This would be equivalent to implementing multiversion linearizability.

As an object-based concurrency protocol, linearization assumes operations independently transform the object from one valid state to another. Consequently it provides no way to address inter-object constraints. In other words, it remains subject to anomaly **A5B**. The fact that when applied to RDF **A5B** corresponds to an attempt to update a field of an RDF-serialized means that the reconciliation algorithm used to address **P4** can leave the system violating application cardinality constraints. Consequently applications requiring the prohibition of **A5B** are a primary motivation for providing support for full serializability in the transaction scheduler. “Linearizability ... is intended for applications ... where programmers are willing to apply special-purpose synchronization protocols, and to reason explicitly about the effects of concurrency” [Herlincy90]

## **Additional issues for Mulgara**

What interface (if any) should be provided to this protocol to other non-XA2 resolvers?

What does it mean for an update to an RDF graph to be 'inconsistent'? Inconsistent with-respect-to RDF? RDFS? OWL-DL? OWL-FULL? Cardinality? Other?

How do we reconcile the idea of a blind-insert into a blind-dropped model? (this is related to the previous question, as this is an inconsistency at a higher level than pure-RDF)

Do we want to add either a simple or compound pattern delete operation as a primitive to the store? (This might allow us to treat many delete/select statements as write-only, rather than read-write — with obvious advantages under MVOCC)

Extending/Subsetting the edge-set can imply extending the stored vertex-set. As discussed, semantically this isn't an issue as the vertex-set is defined by the RDF Semantics and is external to the store, however internally we will need to consider how we handle node-allocations in a multi-writer StringPool.

Does the concept of a blind-delete even make sense? Don't we have to implicitly query the graph before we can de-assert triples in that graph?

Do we need to push the concept of 'reresolve' from the join-optimisation down to the store to permit transactions to 'forget' superseded BGPs? (Note this may significantly affect the interaction of queries that contain `<?s ?p ?o>` as a BGP which various elements restricted to single elements via a `<mulgara:is>` constraint or an application enforced `[0,1]` cardinality restriction)

## **Bibliography**

Agrawal, D. and Krishnaswamy, V., "Using Multiversion Data for Non-Interfering Execution of Write-only Transactions", Proc. ACM SIGMOD 1991  
Berensan, H et al., "A Critique of ANSI SQL Isolation Levels", Proc. ACM SIGMOD 1995

Bernstein, P. et al., "Concurrency Control and Recovery in Database Systems", Addison Wesley, 1987, Chapter 2: Serializability Theory

ibid, Chapter 5: Multiversion Concurrency Control

Bober, P. and Carey, M., "Multiversion Query Locking", Proc. 18th International Conference on Very Large Data Bases, 497-510, 1992

Eswaran, K. et al., "The Notions of Consistency and Predicate Locks in a Database System", Communications of the ACM, Vol 19, No 11, 624-633, 1976

Herlincy, M. and Wing, J., "Linearizability: A Correctness Condition for Concurrent Objects", ACM Transactions on Programming Languages and Systems, Vol 12, No 3, 1990

Kung, H. and Robinson, J., "On Optimistic Methods for Concurrency Control", ACM Transactions on Database Systems, Vol 6, No 2, 1981

Phatah, S and Badrinath, B., "Multiversion Reconciliation for Mobile Databases", Proc. 15th International Conference on Data Engineering, 582-589, 1999