Scaling Jena in a commercial environment: The Ingenta MetaStore Project

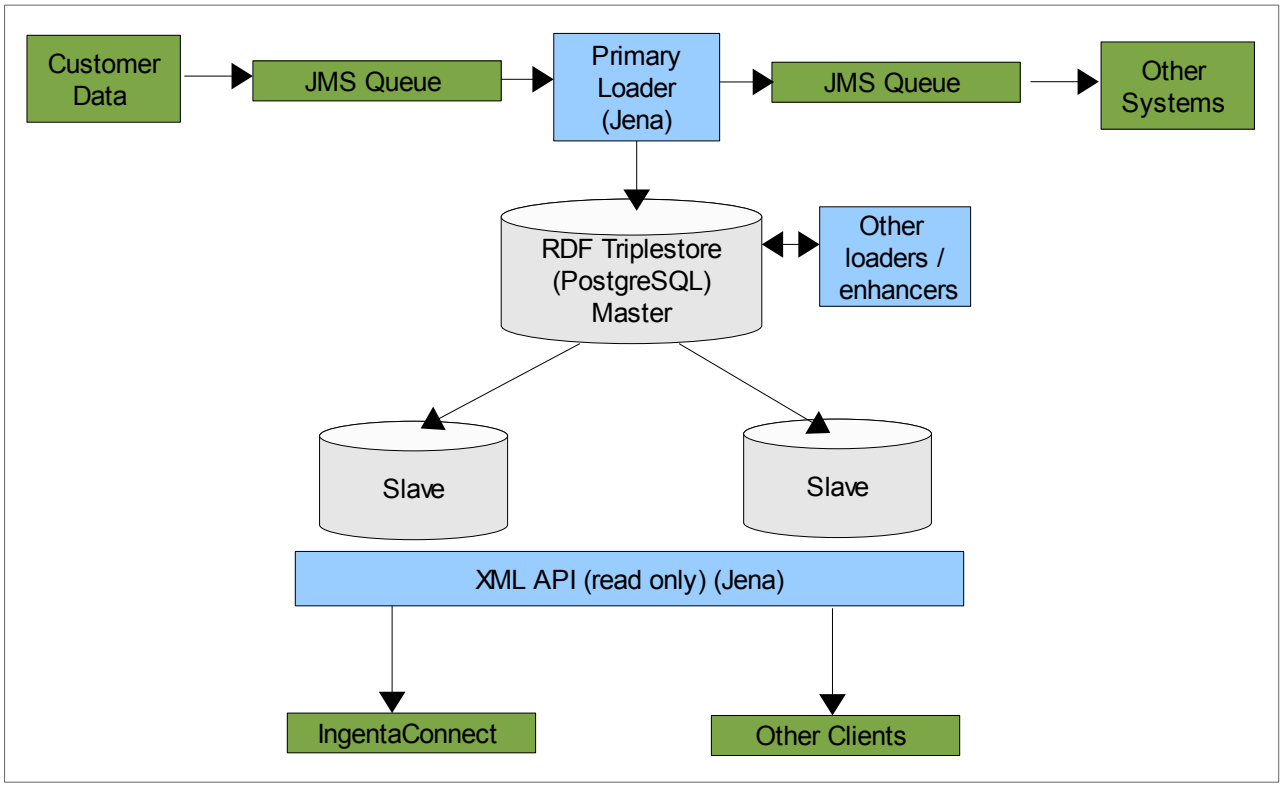
Priya.Parvatikar@ingenta.com and Katie.Portwin@ingenta.com

The Ingenta[1] MetaStore is a case study in using Jena API[2]. to implement a triplestore on a very large scale, within a commercial environment. This paper consists of two sections: one describes the MetaStore project itself, the other focuses on experiences with Jena. The first section explains what commercial and technical problems the project was designed to solve, and why developers chose to use an RDF triplestore to solve them. The focus is on system architecture and data modelling. The second section explains why developers chose Jena for the project, and then discusses some problems encountered while using Jena. In general, a Jena/PostgreSQL triplestore scaled up to our 200 million triple requirement. Quantitative results indicate that, given certain optimisations, the Jena implementation of SPARQL[3] does scale.

Keywords: RDF, Triplestore, Jena, Scalability, SPARQL

1. The Project

Figure 1: MetaStore Architecture Diagram



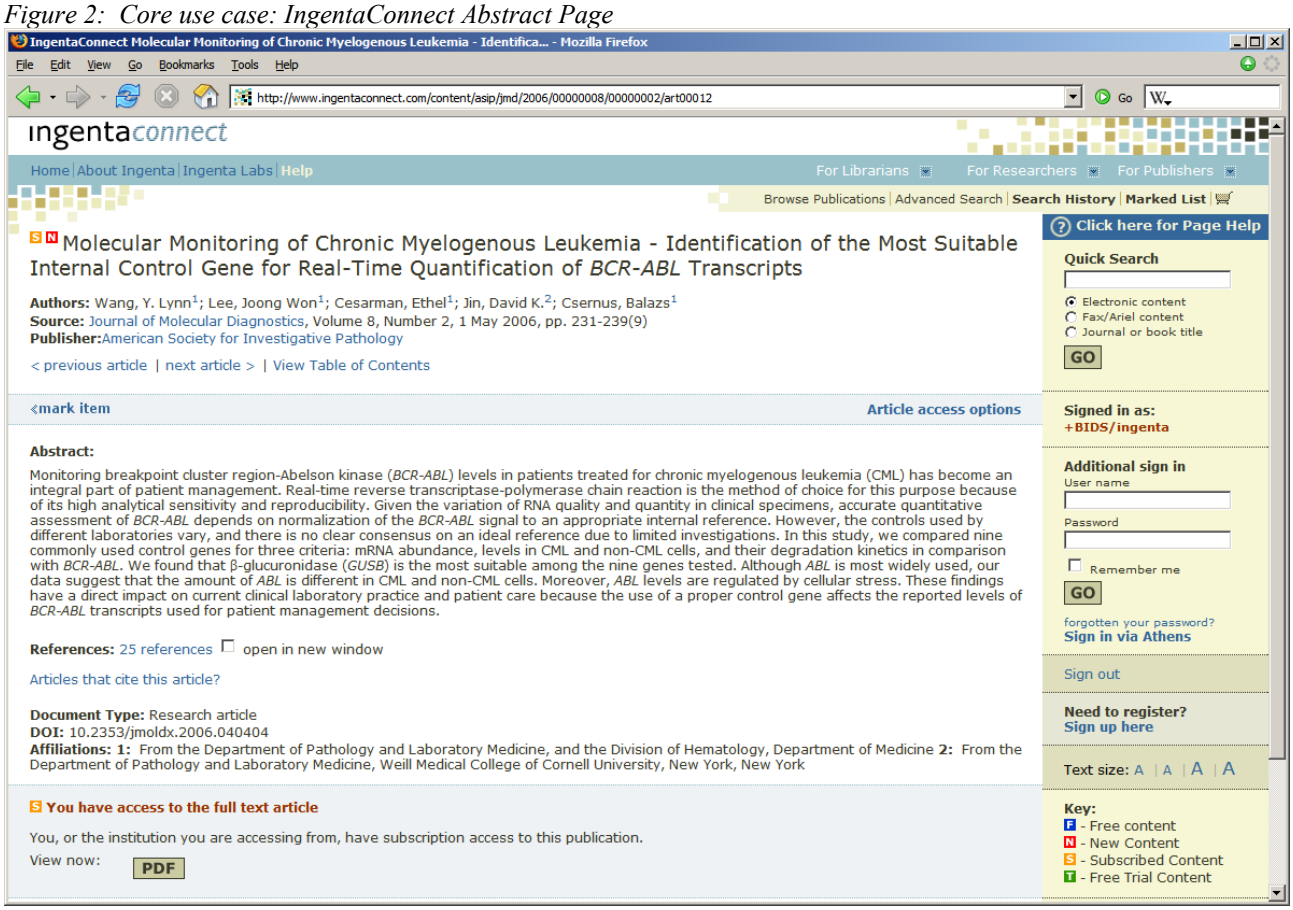
1.1 Aim

The aim of the MetaStore Project is to merge Ingenta's huge, heterogeneous set of metadata into a single centralised repository, with a highly flexible data model. The repository will be the backbone of the IngentaConnect website. It will interface with existing applications and be a

1 http://www.ingentaconnect.com
2 http://jena.sourceforge.net/
3 http://www.w3.org/TR/rdf-sparql-query/

platform for future development. An example core use case of the new repository is to produce metadata suitable for transformation into an "abstract page" on the IngentaConnect website, such as that shown in Figure 2.

Figure 2: Core use case: IngentaConnect Abstract Page



1.2 Background

The IngentaConnect website provides online access to scientific research publications including journal articles, book, and statistics databases. It contains electronic metadata for 4.3 million online articles, and supports about 2 million sessions per month.

1.3 Problem

Currently, IngentaConnect metadata is distributed across a variety of data stores, including various relational database platforms and an SGML file-based store. Synchronising and linking across these data stores is a problem. Furthermore, our publisher clients are increasingly producing a varied array of electronic publications: for example, books, supplementary material, "virtual journals" and multi-lingual publications. Extending existing systems to support these new types of data tends to involve significant development effort and modelling compromises.

1.4 Requirements

1. **Centralised store.** Metadata from a variety of existing data stores will be integrated and stored in unified way. A variety of existing applications will need to interact with it.
2. **Flexible data model.** The repository should have the ability to store all existing content items (including articles, journals, authors, etc), and also be extensible in the future as new data requirements arise.
3. **Scalable.** The repository must support our very large dataset.
4. **Query Performance.** The repository will be the backbone of the IngentaConnect website. When a user is viewing an abstract page[4] on IngentaConnect, she expects to see the abstract of the article, along with other metadata such as authors, keywords, references, and expects to receive the page quickly. At peak usage (weekday mornings GMT), the website services four requests for abstract pages per second. Some caching will take place, but as an aim/a benchmark, the repository should be able to service about four article queries per second.
5. **Distributable.** Ingenta has distribution agreements with partner organisations. Ingenta and the partner need to agree a suitable XML format for transmission, which they can both understand easily. Therefore the new data model should conform to industry standards wherever possible.
6. **Integratable.** Existing applications will need to be changed to use the repository as their data source, and future applications will be developed on the platform. Other application programmers must be able to interact with the repository in a simple way, and retrieve data suitable for their needs easily.

Based on these requirements, an RDF triplestore was chosen for this project. The RDF model is naturally flexible, and standard vocabularies such as Dublin Core[5] can be used to meet the distributability requirement. Centralisation was achieved by careful modelling, followed by bulk loading. Scalability and query performance were key goals in the selection of the RDF engine for the project – as discussed later. Integration, scalability, and query performance were addressed in the system architecture - below.

1.5 Architecture

The master triplestore is at the core of the MetaStore framework (Figure 1). As a centralised store replacing multiple legacy databases, it implements the shared database[6] pattern. The project developers undertook a round of extensive, careful modelling in consultation with other application developers and database owners across the organisation, in order to design a model which would support all existing functionality and provide entry points for integration.

As shown in Figure 1, the database server architecture is master-slave. This architecture provides reliability through redundancy. It also ensures consistent query performance, and cuts down on concurrency problems, as data is loaded on the master only, while clients query the slaves only. The system scales easily as more slaves can be added. By using an RDF API which uses a standard RDBMS backend, the system leverages the mature slaving functionality of the database (PostgreSQL[7]). With a native store, it would have been necessary to develop a custom replication strategy.

4 http://www.ingentaconnect.com/content/apl/ebt/2005/00000005/00000011/art00001
5 http://dublincore.org/
6 http://www.awprofessional.com/articles/article.asp?p=169483&seqNum=3&rl=1
7 http://www.postgresql.org/

A loader program subscribes to a JMS[8] queue[9] of newly arrived data. Each day, about 500 articles are loaded. The loader validates, transforms and inserts the new resource using the Jena API. At this point the resource is also assigned a unique identifier. Only after it has successfully been loaded into the repository are notifications sent to other systems (for example Search Indexers, Full Text Delivery Servers). In addition to this, other applications (for example, External Reference Resolvers, Spot Updaters, Sequence Generators) also regularly make additions, updates and deletes to the repository.

Clients query the store using a REST[10]-ful XML API. The advantage of providing this interface is that all the RDF Engine (Jena/Java) code is encapsulated. Application programmers across the organisation can produce language-agnostic, engine-agnostic clients, with little consultation with the repository developers.

Figure 3: Sample RDF/XML for an Article

1.6 Modelling with RDFS[11]

Existing vocabularies were used where possible: Dublin Core, PRISM[12] and FOAF[13] . Custom

8 http://java.sun.com/products/jms/
9 http://www.awprofessional.com/articles/article.asp?p=169483&seqNum=5&rl=1
10 http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
11 http://www.w3.org/TR/rdf-schema/
12 http://www.prismstandard.org/
13 http://www.foaf-project.org/

vocabularies were developed where necessary. The new vocabularies in the Ingenta namespace, including "branding", "identifiers", "structure", extend the standards where possible. For example, `struct:Author` extends `foaf:Person`, and `ident:infobike` extends `dc:identifier`. This range is demonstrated in Figure 3 - sample RDF/XML for an Article (the core use case).

The data model is strongly hierarchical. Publishers have journals, journals have issues, issues have articles. All these resource types extend `struct:Part` and are linked with the `prism:isPartOf` property. Query performance testing results presented later should be interpreted in the context[14] of this hierarchical model.

In its current state, the RDFS model includes 28 Classes and 72 Properties, 4 /18 of which respectively are from the standard vocabularies. This model continues to evolve.

1.7 Scale

Headers and references for ~4.3 million articles have been loaded into the repository. This has resulted in *~200 million rows* in `jena_g1t1_stmt`, and the resulting database size on disk is 65Gb. Thus on average, each article is responsible for ~ 47 triples.

The `jena_long_lit` table contains approximately 4.5 million records – these mainly represent article abstracts and some long article titles. The `jena_long_uri` table has approximately 0.14 million records.

2. Experiences with Jena

2.1 Why Jena?

The first stage of the project was to evaluate and choose an RDF engine. Developers investigated:

- Jena with a PostgreSQL backend
- Sesame with a PostgreSQL backend
- Kowari with a native Kowari backend

With the limited timeframe and the versions of APIs available at that time, (early 2005,) Jena was chosen for the following reasons:

- **Java** – Ingenta has in-house expertise in Java. In a commercial environment, code maintainability years into the future is a paramount concern.
- **RDBMS backend** – A relational database was preferred over a native store because of the built-in support for replication, and trusted backup mechanism. Jena has support for a wide range of database platforms; including PostgreSQL which is Ingenta's standard database.
- **Usability** - The Jena API is easy to use and stable. The API is well-documented and the mailing lists are active.
- **Performance** – Preliminary load-testing indicated that Jena is scalable with regard to query performance. In contrast, the cost of opening the model in Sesame became prohibitive with large numbers of triples (20 minutes with 100 million triples).
- **Scalability** – Preliminary load-testing also demonstrated that Jena is scalable with regard to memory usage. With Kowari, memory problems were encountered at 25 million triples.
- **Debuggable** - The raw data can be viewed through the PostgreSQL client. This was useful

14 http://jeenbroekstra.blogspot.com/2006/02/pitfalls-in-benchmarking-triple-stores.html

while developing and debugging.

2.2 Challenges

2.2.1 Insert Performance - Batching

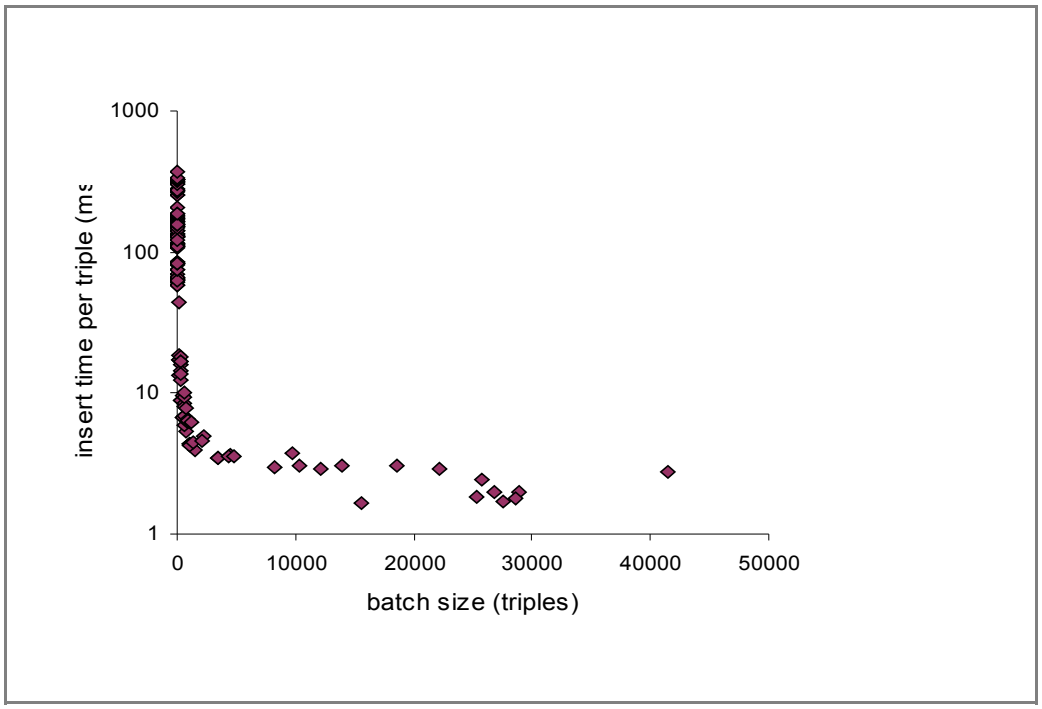
Early versions of the loading program inserted a single article at a time. In one, the developer made API calls for each triple; in another, the developer created a chunk of RDF/XML for a single article, and inserted that.

However, as the size of the repository increased beyond 0.5 million articles, these approaches proved to be too slow. This may have been due to database cleanup or index rebuilding after each insert. Turning off the index-rebuild, however, was not a suitable solution, since it caused subsequent queries to be inefficient. Any reduction in query performance impacts on loading performance too – in order to establish cross-links, querying the existing data is an essential part of the loading mechanism.

Others have suggested turning off duplicate checking in Jena[15] and instead, forcing the loading code to check for duplicates. This could have been viable for a single one-off load. However, with frequent ongoing loading, comparing the existing 65 Gb store with the new batch would be problematic.

Finally, a batching approach to loading the data was developed. RDF/XML was created for a batch of several thousand triples at a time, and then added to the repository in one insert command.

Figure 4: Effect of batching on Insert Performance



15 http://nuin.blogspot.com/2006/02/jena-tip-optimising-database-load.html

Batching substantially improved the loading performance, (as shown in Figure 4). Batch sizes of around 5000 triples are optimal – approximately 5ms per triple. Beyond this point there are diminishing returns to larger batches, and memory errors become a risk.

The cost of batching is extra complexity in the loading code. For example, while creating data for each new article, various searches are performed on existing data. With the batching approach, these queries had to be extended to check within the current batch as well as in the repository.

2.2.2 Ontologies - Memory Problems

Initially, the project design included OWL inferencing in the main model:

- To reduce size of store by inferring some properties.
- To avoid hardcoding relations - these would be inferred instead, thereby making it easier to update data in the store.
- To facilitate validation of key properties in the data.

However, the Jena implementation of OWL did not scale for a dataset of this size[16]. In fact, the JVM ran out of memory by 11 million triples.

It was therefore not possible to use inferencing to reduce the size of the store – for example, it could not be used to infer inverse relations such as `prism:references` and `prism:isReferencedBy`. One option to achieve the same effect would be at the program level – for instance query rewriting. Another is to explicitly load both sides of the relation. For the core use case, (delivering everything needed for an abstract page) the latter approach was taken. In general, and particularly in areas where performance was critical, data was pre-calculated wherever possible (see section 3.2 for further discussion).

It may still be possible to make use of the OWL ontology for data validation purposes – that is, on small batches of data during the loading process; this is a future goal.

2.2.3 The Object Model – encapsulating Jena code / limiting flexibility

The Jena API allows the programmers to talk to the store in terms of triples. However, to promote re-use and encapsulation of the Jena code, application programmers need to talk in terms of objects like Book and Article. An object model was developed which closely mirrored the schema, using interfaces to capture multiple inheritance. A FinderDAO and a Factory[17] class were developed for each data object type, to encapsulate all the Jena/RDF code needed to find a resource, and construct an instance of the appropriate class. The instance would have not only literal values but also references to other instances; for example, a retrieved `Article` instance needs a reference to its parent `Issue` instance, and grandparent `Journal` instance, because many useful properties are in fact held higher up the hierarchical model.

The intended goal of encapsulating all Jena code in Factory classes was to isolate interaction with the triplestore from other application logic in the loading and querying programs. The disadvantage of course is inflexibility: developers working in terms of `Article`, `Journal`, `Book` – instead of in terms of Triple – may not see the benefits of the underlying RDF model.

16 http://groups.yahoo.com/group/jena-dev/message/20582
17 http://en.wikipedia.org/wiki/Factory_method_pattern

Other problems encountered included:

- Developing a clean implementation hierarchy to match the multiple inheritance of the `Interface` model.
- Limiting recursion depth and avoiding infinite loops while populating instances (For example, article A `prism:references` article B , article B `prism:references` article A etc.)

2.2.4 Prefixes – Suggested addition to Schemagen

Jena's Schemagen[18] utility makes resource and property names from an RDFS schema available to Java application programmers as static variables. This promotes maintainable code – essential in our environment. However, the current schemagen does not include preferred namespace prefix. For this project, schemagen was extended to interpret the `preferredNamespacePrefix` property from the VANN[19] vocabulary.

For example in the STRUCTURE schema:

```
<vann:preferredNamespacePrefix>struct</vann:preferredNamespacePrefix>
```

Thus, while constructing an RDF/XML snippet for a Publisher resource, the prefix could be used as follows:

```
publisherDoc.createElement(
            STRUCTURE.getPrefix() + ":" + STRUCTURE.Publisher.getLocalName()
);
```

2.3 Performance Testing SPARQL

SPARQL queries may be executed within a Jena program using the ARQ API[20]. Performance tests were carried out on an Intel(R) Xeon(TM) server (CPU 3.20GHz 6 SCSI Drives) with 4G RAM, running PostgreSQL 7 and Jena 2.3, on Debian.

The basic SPARQL query used for testing was of the form:

Example 1: Standard SPARQL query (Title-type only)

```
PREFIX  dcterms: <http://purl.org/dc/terms/>
PREFIX  struct: <http://metastore.ingenta.com/ns/structure/>
PREFIX  dc:    <http://purl.org/dc/elements/1.1/>
PREFIX  linking: <http://metastore.ingenta.com/ns/linking/>
PREFIX  rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  prism: <http://prismstandard.org/namespaces/1.2/basic/>
PREFIX  ident: <http://metastore.ingenta.com/ns/identifiers/>
SELECT ?title ?issue ?article
WHERE {
?title    rdf:type        struct:Journal .
?title    dc:identifier   <http://metastore.ingenta.com/content/issn/02670836> .
?issue    prism:isPartOf  ?title .
?issue    prism:volume    ?volumeLiteral .
?issue    prism:number    ?issueLiteral .
```

18 http://jena.sourceforge.net/how-to/schemagen.html
19 http://vocab.org/vann/
20 http://jena.sourceforge.net/ARQ/app_api.html

```
?article  prism:isPartOf  ?issue .
?article  prism:startingPage  ?firstPageLiteral .
FILTER ( ?volumeLiteral = "20" )
FILTER ( ?issueLiteral = "4" )
FILTER ( ?firstPageLiteral = "539" )
}
```

The standard query is a good demonstration example because it ranges over several resources in the model, and filters based on literal values. Furthermore, it is a realistic query which might be performed on the repository; a client application looking for an Article in the store would probably have access to bibliographic metadata such as start page, ISSN, volume, etc. (In fact, this query was used in an early version of the loading program.) The literal values were chosen such that a result set of exactly one resource would be returned in each instance.

Note the inclusion of the "`?title rdf:type..`" triple. In fact, early attempts at composing the basic article query did not include this statement, and the developers immediately noticed a very serious problems with performance. Developers then experimented with several other versions of the query, including and excluding rdf:type statements, and experimenting with filters. Formal testing was then conducted.

Two of other logical versions are presented here, since they are illustrative: The first is a version with `rdf:type` restrictions for every resource:

Example 2: All-Types SPARQL query

```
SELECT ?title ?issue ?article
WHERE {
?title    rdf:type        struct:Journal .
?title    dc:identifier   <http://metastore.ingenta.com/content/issn/02670836> .
?issue    prism:isPartOf  ?title .
?issue    rdf:type        struct:Issue .
?issue    prism:volume    ?volumeLiteral .
?issue    prism:number    ?issueLiteral .
?article  prism:isPartOf  ?issue .
?article  rdf:type        struct:Article .
?article  prism:startingPage  ?firstPageLiteral .
FILTER ( ?volumeLiteral = "20" )
FILTER ( ?issueLiteral = "4" )
FILTER ( ?firstPageLiteral = "539" )
}
```

The other is a version without any of the `rdf:type` restrictions:

Example 3: No-types SPARQL query

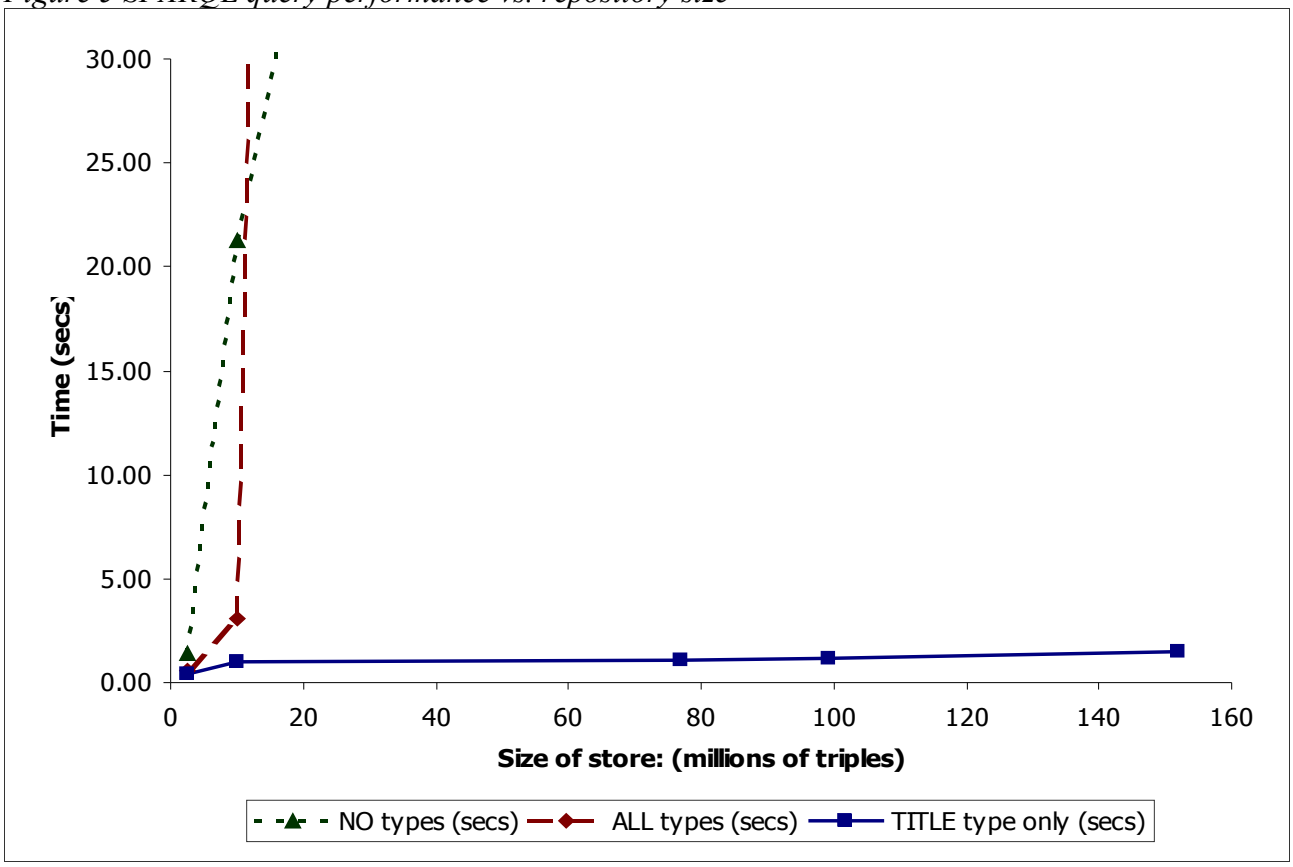
```
SELECT  ?title ?issue ?article
WHERE {
?title    dc:identifier   <http://metastore.ingenta.com/content/issn/02670836> .
?issue    prism:isPartOf  ?title .
?issue    prism:volume    ?volumeLiteral .
?issue    prism:number    ?issueLiteral .
?article  prism:isPartOf  ?issue .
?article  prism:startingPage  ?firstPageLiteral .
FILTER ( ?volumeLiteral = "20" )
FILTER ( ?issueLiteral = "4" )
FILTER ( ?firstPageLiteral = "539" )
}
```

Results:

Size of store (millions of triples)	2	9	77	99	152
NO types (ms)	1404	21237	123547	*	*
ALL types (ms)	598	3114	*	*	*
TITLE type only (ms)	385	987	1048	1128	1465

(* = never returns)

Figure 5 SPARQL query performance vs. repository size



Performance of the standard query is represented by the blue line on Figure 5. It can be seen that performance deteriorates slowly as store size increases. Notwithstanding this point, SPARQL performance is still reasonable, even with a very large store: at **150 million triples**, it returns in **less than 1.5 seconds**.

However, the all-types and the no-types versions performed very poorly with the larger store sizes. All three queries are logically equivalent, but the addition or removal of conditions has an immense adverse effect on performance. This means that there is no simple conclusion: it is not the case that more statements improve performance, nor that fewer statements improve performance.

In fact, the explanation lies in the way the PostgreSQL query planner chooses to execute the final SQL queries. For example, in the no-types query, a sequential scan is performed. Further information can be seen in Appendix 1.

In this case, a suitable optimisation was found by trial-and-error. The development of a strategy

for optimising SPARQL queries (with PostgreSQL) in the general case, is a goal for future research.

However, the following recommendations can be made at this stage:

A. It is worth experimenting with different versions of a SPARQL query.

B. Developers may need to investigate at the RDBMS level, so it is worth choosing an database with which the organisation has expertise.

3. Current Status

3.1 Recent Developments

The MetaStore has been loaded with 4.3 million article headers, through an initial bulk loading process. Processes have also been put in place to ensure that the repository stays in synch with the ongoing daily changes to data.

Books – a deviation from the standard journal-issue-article data model – have been independently modelled[21] and loaded into the store.

A RESTful query API has been developed in order to allow client applications to start using the MetaStore. Given a resource identifier, the API returns all the data associated with that resource in XML format (See Figure 3). Every resource in the MetaStore has been assigned a primary identifier. This identifier has been designed to be a stable and unique URI in the following style:

```
http://metastore.ingenta.com/content/ [type] / [auto-incremented
number]
```

Example 4: Stable primary identifiers

```
http://metastore.ingenta.com/content/articles/1
http://metastore.ingenta.com/content/titles/42
```

The MetaStore developers have attempted to establish effective query strategies for various types of applications depending on their requirements.

For cases where query performance is critical, it has been decided that identifier-based queries are used as far as possible. However, client systems are unlikely to have easy access to the primary identifier and usually have their own identifiers in their databases. Porting all such existing identifiers into the repository provides a simple, powerful integration hook.

Example 5: Predictable secondary identifiers

```
<struct:Article rdf:about="http://metastore.ingenta.com/content/articles/42">
      <linking:genlinkerRefId rdf:resource="genlinker://refid/5518325"/>
      <ident:infobike
rdf:resource="infobike://maney/mint/2003/00000112/00000003/art00001"/>
      <dc:identifier
rdf:resource="http://metastore.ingenta.com/content/maney/03717844/v112n3/s1"/>
      <ident:doi>10.1179/037178403225003582</ident:doi>
      <ident:sici>0371-7844(20031201)112:3L.141;1-</ident:sici>
</struct:Article>
```

21 http://allmyeye.blogspot.com/2006/03/does-your-boy-scout-handbook-look-as.html

Some of these identifiers shown in Example 5, have the potential to be used by multiple clients and they have been modelled using the `dc:identifier` property.

There are other identifiers that are relevant only for particular clients. For example `linking:genlinkerRefId` is an article identifer in the legacy reference linking database. Application-specific namespaces have been developed for such properties.

In addition, there are industry standard identifiers for example, DOI and SICI, that enable integration with external partners.

The primary use case and the most important client to use the MetaStore is the IngentaConnect application. In most cases, IngentaConnect will query the API using predictable identifiers. This will ensure that the query performance scales to requirements.

For cases where query performance is not such a critical concern for example batch processing applications or cases where identifiers are not sufficient for example reference matching applications – SPARQL queries can be used where necessary. The developers have tried to identify SPARQL queries that are useable - for example a query to get a list of all the publishers loaded into the store- and queries that are not useable because of their slow performance – for example a query to search on authors using a literal value.

Example 6: Useable SPARQL query

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX struct: <http://metastore.ingenta.com/ns/structure/>
SELECT ?pubid ?pubname
WHERE {
 ?pubid rdf:type struct:Publisher .
 ?pubid dc:title ?pubname .
}
```

Example 7: Unuseable SPARQL query

```
PREFIX struct: <http://metastore.ingenta.com/ns/structure/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX branding: <http://metastore.ingenta.com/ns/branding/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?au
WHERE {
 ?au rdf:type struct:Author .
 ?au foaf:family_name ?name
 FILTER ( ?name = "Zhu" )
}
```

3.2 Current developments

The developers are currently investigating the use of SPARQL for specialised or experimental cases – for example, merging[22] of MetaStore data with externally held data to produce a richer data set.

The developers are also still working on the integration of the MetaStore with other Ingenta systems. The IngentaConnect developer team need to produce a “Table of Contents” for every issue - this is a set of articles and is ordered. Therefore the articles belonging to an issue will need to be

22 http://allmyeye.blogspot.com/2006/04/amazon-ingenta-sparql-nifty.html

represented by a sequence (`rdf:Seq`). These sequences will be pre-calculated and stored for performance and simplicity.

The initial bulk loading of reference linking data into the store is also in progress. This data has been modelled as a combination of standard (for example `prism:references`) and custom (for example `linking:HostingDescription`) properties and classes.

In addition, the developers are working on the replication of the database – an important concern for scalability (see section 1.5).

4. Conclusions

The Ingenta MetaStore case study demonstrates that Jena scales to 200 million triples. This paper has explained why Jena is a good choice of RDF engine for a commercial triplestore implementation. Some problems with Jena 2.3 in the context of a very large store were presented. First, loading performance is a challenge, although one proven solution is a batching approach. Second, inferencing using OWL is not currently scalable to a store of this size. Third, in a commercial environment, it is useful to separate RDF/Jena code from other application logic, but this approach brings its own costs. Finally, quantitative testing shows that it is possible to achieve good SPARQL performance even with a very large store, but that queries should be carefully optimised.

Acknowledgments

Leigh.Dodds@ingenta.com
Charlie.Rapple@ingenta.com

APPENDIX I SQL / Query Plan for the SPARQL performance testing /optimisation queries

Excerpts from the postgres logs showing the SQL generated by Jena, and query plans generated by PostgreSQL.

1. Title Type Only

```
Select A0.Subj, A2.Subj, A3.Obj, A4.Obj, A5.Subj, A6.Obj From jena_g1t1_stmt A0, jena_g1t1_stmt A1,
jena_g1t1_stmt A2, jena_g1t1_stmt A3, jena_g1t1_stmt A4, jena_g1t1_stmt A5, jena_g1t1_stmt A6 Where
A0.Prop='Uv::http://www.w3.org/1999/02/22-rdf-syntax-ns#type' AND
A0.Obj='Uv::http://metastore.ingenta.com/ns/structure/Title' AND A0.GraphID=1 AND A0.Subj=A1.Subj
AND A1.Prop='Uv::http://purl.org/dc/elements/1.1/identifier' AND
A1.Obj='Uv::http://metastore.ingenta.com/content/issn/1478422x' AND A1.GraphID=1 AND
A2.Prop='Uv::http://prismstandard.org/namespaces/1.2/basic/isPartOf' AND A0.Subj=A2.Obj AND
A2.GraphID=1 AND A2.Subj=A3.Subj AND
A3.Prop='Uv::http://prismstandard.org/namespaces/1.2/basic/number' AND A3.GraphID=1 AND
A2.Subj=A4.Subj AND A4.Prop='Uv::http://prismstandard.org/namespaces/1.2/basic/volume' AND
A4.GraphID=1 AND A5.Prop='Uv::http://prismstandard.org/namespaces/1.2/basic/isPartOf' AND
A2.Subj=A5.Obj AND A5.GraphID=1 AND A5.Subj=A6.Subj AND
A6.Prop='Uv::http://prismstandard.org/namespaces/1.2/basic/startingPage' AND A6.GraphID=1
```

```
 Nested Loop  (cost=0.00..207796.39 rows=1 width=336)
   ->  Nested Loop  (cost=0.00..207790.37 rows=1 width=284)
         ->  Nested Loop  (cost=0.00..207784.34 rows=1 width=344)
               ->  Nested Loop  (cost=0.00..207778.32 rows=1 width=232)
                     ->  Nested Loop  (cost=0.00..157951.71 rows=1 width=120)
                           ->  Nested Loop  (cost=0.00..58298.49 rows=2 width=120)
                                 ->  Index Scan using jena_g1t1_stmt_ixo on jena_g1t1_stmt a0  (cost=0.00..36761.97 rows=847
width=60)
                                       Index Cond: ((obj)::text = 'Uv::http://metastore.ingenta.com/ns/structure/Title'::text)
                                       Filter: (((prop)::text = 'Uv::http://www.w3.org/1999/02/22-rdf-syntax-ns#type'::text) AND
(graphid = 1))
                                 ->  Index Scan using jena_g1t1_stmt_ixsp on jena_g1t1_stmt a1  (cost=0.00..25.41 rows=1 width=60)
                                       Index Cond: ((("outer".subj)::text = (a1.subj)::text) AND ((a1.prop)::text =
'Uv::http://purl.org/dc/elements/1.1/identifier'::text))
                                       Filter: (((obj)::text = 'Uv::http://metastore.ingenta.com/content/issn/1478422x'::text) AND
(graphid = 1))
                           ->  Index Scan using jena_g1t1_stmt_ixo on jena_g1t1_stmt a2  (cost=0.00..49824.98 rows=130 width=112)
                                 Index Cond: ((a2.obj)::text = ("outer".subj)::text)
                                 Filter: (((prop)::text = 'Uv::http://prismstandard.org/namespaces/1.2/basic/isPartOf'::text) AND
(graphid = 1))
                     ->  Index Scan using jena_g1t1_stmt_ixo on jena_g1t1_stmt a5  (cost=0.00..49824.98 rows=130 width=112)
                           Index Cond: (("outer".subj)::text = (a5.obj)::text)
                           Filter: (((prop)::text = 'Uv::http://prismstandard.org/namespaces/1.2/basic/isPartOf'::text) AND (graphid
= 1))
               ->  Index Scan using jena_g1t1_stmt_ixsp on jena_g1t1_stmt a3  (cost=0.00..6.01 rows=1 width=112)
                     Index Cond: ((("outer".obj)::text = (a3.subj)::text) AND ((a3.prop)::text =
'Uv::http://prismstandard.org/namespaces/1.2/basic/number'::text))
                     Filter: (graphid = 1)
         ->  Index Scan using jena_g1t1_stmt_ixsp on jena_g1t1_stmt a4  (cost=0.00..6.01 rows=1 width=112)
               Index Cond: (((a4.subj)::text = ("outer".subj)::text) AND ((a4.prop)::text =
'Uv::http://prismstandard.org/namespaces/1.2/basic/volume'::text))
               Filter: (graphid = 1)
   ->  Index Scan using jena_g1t1_stmt_ixsp on jena_g1t1_stmt a6  (cost=0.00..6.01 rows=1 width=112)
         Index Cond: ((("outer".subj)::text = (a6.subj)::text) AND ((a6.prop)::text =
'Uv::http://prismstandard.org/namespaces/1.2/basic/startingPage'::text))
         Filter: (graphid = 1)
(27 rows)
```

2. All Types

```
Select A0.Subj, A2.Subj, A3.Obj, A4.Subj, A5.Obj, A8.Obj From jena_g1t1_stmt A0, jena_g1t1_stmt A1,
jena_g1t1_stmt A2, jena_g1t1_stmt A3, jena_g1t1_stmt A4, jena_g1t1_stmt A5, jena_g1t1_stmt A6,
jena_g1t1_stmt A7, jena_g1t1_stmt A8 Where A0.Prop='Uv::http://www.w3.org/1999/02/22-rdf-syntax-
ns#type' AND A0.Obj='Uv::http://metastore.ingenta.com/ns/structure/Title' AND A0.GraphID=1 AND
A0.Subj=A1.Subj AND A1.Prop='Uv::http://purl.org/dc/elements/1.1/identifier' AND
A1.Obj='Uv::http://metastore.ingenta.com/content/issn/13621718' AND A1.GraphID=1 AND
A2.Prop='Uv::http://prismstandard.org/namespaces/1.2/basic/isPartOf' AND A0.Subj=A2.Obj AND
A2.GraphID=1 AND A2.Subj=A3.Subj AND
A3.Prop='Uv::http://prismstandard.org/namespaces/1.2/basic/volume' AND A3.GraphID=1 AND
A4.Prop='Uv::http://prismstandard.org/namespaces/1.2/basic/isPartOf' AND A2.Subj=A4.Obj AND
A4.GraphID=1 AND A4.Subj=A5.Subj AND
A5.Prop='Uv::http://prismstandard.org/namespaces/1.2/basic/startingPage' AND A5.GraphID=1 AND
A2.Subj=A6.Subj AND A6.Prop='Uv::http://www.w3.org/1999/02/22-rdf-syntax-ns#type' AND
A6.Obj='Uv::http://metastore.ingenta.com/ns/structure/Part' AND A6.GraphID=1 AND A4.Subj=A7.Subj
AND A7.Prop='Uv::http://www.w3.org/1999/02/22-rdf-syntax-ns#type' AND
A7.Obj='Uv::http://metastore.ingenta.com/ns/structure/Article' AND A7.GraphID=1 AND A2.Subj=A8.Subj
AND A8.Prop='Uv::http://prismstandard.org/namespaces/1.2/basic/number' AND A8.GraphID=1
```

```
 Nested Loop  (cost=0.00..207840.72 rows=1 width=336)
   ->  Nested Loop  (cost=0.00..207834.70 rows=1 width=456)
```

```
        ->  Nested Loop  (cost=0.00..207812.53 rows=1 width=396)
              ->  Nested Loop  (cost=0.00..207806.51 rows=1 width=284)
                    ->  Nested Loop  (cost=0.00..207784.34 rows=1 width=344)
                          ->  Nested Loop  (cost=0.00..207778.32 rows=1 width=232)
                                ->  Nested Loop  (cost=0.00..157951.71 rows=1 width=120)
                                      ->  Nested Loop  (cost=0.00..58298.49 rows=2 width=120)
                                            ->  Index Scan using jena_g1t1_stmt_ixo on jena_g1t1_stmt a0  (cost=0.00..36761.97
rows=847 width=60)
                                                  Index Cond: ((obj)::text =
'Uv::http://metastore.ingenta.com/ns/structure/Title'::text)
                                                  Filter: (((prop)::text = 'Uv::http://www.w3.org/1999/02/22-rdf-syntax-
ns#type'::text) AND (graphid = 1))
                                            ->  Index Scan using jena_g1t1_stmt_ixsp on jena_g1t1_stmt a1  (cost=0.00..25.41 rows=1
width=60)
                                                  Index Cond: ((("outer".subj)::text = (a1.subj)::text) AND ((a1.prop)::text =
'Uv::http://purl.org/dc/elements/1.1/identifier'::text))
                                                  Filter: (((obj)::text =
'Uv::http://metastore.ingenta.com/content/issn/13621718'::text) AND (graphid = 1))
                                      ->  Index Scan using jena_g1t1_stmt_ixo on jena_g1t1_stmt a2  (cost=0.00..49824.98 rows=130
width=112)
                                            Index Cond: ((a2.obj)::text = ("outer".subj)::text)
                                            Filter: (((prop)::text =
'Uv::http://prismstandard.org/namespaces/1.2/basic/isPartOf'::text) AND (graphid = 1))
                                ->  Index Scan using jena_g1t1_stmt_ixo on jena_g1t1_stmt a4  (cost=0.00..49824.98 rows=130
width=112)
                                      Index Cond: (("outer".subj)::text = (a4.obj)::text)
                                      Filter: (((prop)::text = 'Uv::http://prismstandard.org/namespaces/1.2/basic/isPartOf'::text)
AND (graphid = 1))
                          ->  Index Scan using jena_g1t1_stmt_ixsp on jena_g1t1_stmt a5  (cost=0.00..6.01 rows=1 width=112)
                                Index Cond: ((("outer".subj)::text = (a5.subj)::text) AND ((a5.prop)::text =
'Uv::http://prismstandard.org/namespaces/1.2/basic/startingPage'::text))
                                Filter: (graphid = 1)
                    ->  Index Scan using jena_g1t1_stmt_ixsp on jena_g1t1_stmt a7  (cost=0.00..22.15 rows=1 width=60)
                          Index Cond: (((a7.subj)::text = ("outer".subj)::text) AND ((a7.prop)::text =
'Uv::http://www.w3.org/1999/02/22-rdf-syntax-ns#type'::text))
                          Filter: (((obj)::text = 'Uv::http://metastore.ingenta.com/ns/structure/Article'::text) AND (graphid = 1))
              ->  Index Scan using jena_g1t1_stmt_ixsp on jena_g1t1_stmt a8  (cost=0.00..6.01 rows=1 width=112)
                    Index Cond: (((a8.subj)::text = ("outer".obj)::text) AND ((a8.prop)::text =
'Uv::http://prismstandard.org/namespaces/1.2/basic/number'::text))
                    Filter: (graphid = 1)
        ->  Index Scan using jena_g1t1_stmt_ixsp on jena_g1t1_stmt a6  (cost=0.00..22.15 rows=1 width=60)
              Index Cond: (((a6.subj)::text = ("outer".obj)::text) AND ((a6.prop)::text = 'Uv::http://www.w3.org/1999/02/22-rdf-
syntax-ns#type'::text))
              Filter: (((obj)::text = 'Uv::http://metastore.ingenta.com/ns/structure/Part'::text) AND (graphid = 1))
   ->  Index Scan using jena_g1t1_stmt_ixsp on jena_g1t1_stmt a3  (cost=0.00..6.01 rows=1 width=112)
         Index Cond: ((("outer".obj)::text = (a3.subj)::text) AND ((a3.prop)::text =
'Uv::http://prismstandard.org/namespaces/1.2/basic/volume'::text))
         Filter: (graphid = 1)
(35 rows)
```

3. No Types

```
Select A0.Subj, A1.Subj, A2.Obj, A3.Obj, A4.Subj, A5.Obj From jena_g1t1_stmt A0, jena_g1t1_stmt A1,
jena_g1t1_stmt A2, jena_g1t1_stmt A3, jena_g1t1_stmt A4, jena_g1t1_stmt A5 Where
A0.Prop='Uv::http://purl.org/dc/elements/1.1/identifier' AND
A0.Obj='Uv::http://metastore.ingenta.com/content/issn/09680519' AND A0.GraphID=1 AND
A1.Prop='Uv::http://prismstandard.org/namespaces/1.2/basic/isPartOf' AND A0.Subj=A1.Obj AND
A1.GraphID=1 AND A1.Subj=A2.Subj AND
A2.Prop='Uv::http://prismstandard.org/namespaces/1.2/basic/number' AND A2.GraphID=1 AND
A1.Subj=A3.Subj AND A3.Prop='Uv::http://prismstandard.org/namespaces/1.2/basic/volume' AND
A3.GraphID=1 AND A4.Prop='Uv::http://prismstandard.org/namespaces/1.2/basic/isPartOf' AND
A1.Subj=A4.Obj AND A4.GraphID=1 AND A4.Subj=A5.Subj AND
A5.Prop='Uv::http://prismstandard.org/namespaces/1.2/basic/startingPage' AND A5.GraphID=1
```

```
 Nested Loop  (cost=3411130.77..6719962.25 rows=1538 width=336)
   ->  Nested Loop  (cost=3411130.77..6714473.80 rows=911 width=396)
         ->  Nested Loop  (cost=3411130.77..6711220.49 rows=540 width=344)
               ->  Hash Join  (cost=3411130.77..6709292.61 rows=320 width=232)
                     Hash Cond: (("outer".obj)::text = ("inner".subj)::text)
                     ->  Seq Scan on jena_g1t1_stmt a4  (cost=0.00..3220501.56 rows=839536 width=112)
                           Filter: (((prop)::text = 'Uv::http://prismstandard.org/namespaces/1.2/basic/isPartOf'::text) AND (graphid
= 1))
                     ->  Hash  (cost=3411129.37..3411129.37 rows=561 width=120)
                           ->  Merge Join  (cost=3407609.57..3411129.37 rows=561 width=120)
                                 Merge Cond: ("outer"."?column2?" = "inner"."?column3?")
                                 ->  Sort  (cost=36810.89..36813.35 rows=984 width=60)
                                       Sort Key: (a0.subj)::text
                                       ->  Index Scan using jena_g1t1_stmt_ixo on jena_g1t1_stmt a0  (cost=0.00..36761.97 rows=984
width=60)
                                             Index Cond: ((obj)::text =
'Uv::http://metastore.ingenta.com/content/issn/09680519'::text)
                                             Filter: (((prop)::text = 'Uv::http://purl.org/dc/elements/1.1/identifier'::text) AND
(graphid = 1))
                                 ->  Sort  (cost=3370798.68..3372897.52 rows=839536 width=112)
                                       Sort Key: (a1.obj)::text
                                       ->  Seq Scan on jena_g1t1_stmt a1  (cost=0.00..3220501.56 rows=839536 width=112)
                                             Filter: (((prop)::text =
'Uv::http://prismstandard.org/namespaces/1.2/basic/isPartOf'::text) AND (graphid = 1))
               ->  Index Scan using jena_g1t1_stmt_ixsp on jena_g1t1_stmt a2  (cost=0.00..6.01 rows=1 width=112)
                     Index Cond: ((("outer".obj)::text = (a2.subj)::text) AND ((a2.prop)::text =
'Uv::http://prismstandard.org/namespaces/1.2/basic/number'::text))
                     Filter: (graphid = 1)
         ->  Index Scan using jena_g1t1_stmt_ixsp on jena_g1t1_stmt a5  (cost=0.00..6.01 rows=1 width=112)
               Index Cond: ((("outer".subj)::text = (a5.subj)::text) AND ((a5.prop)::text =
```

```
'Uv::http://prismstandard.org/namespaces/1.2/basic/startingPage'::text))
               Filter: (graphid = 1)
   ->  Index Scan using jena_g1t1_stmt_ixsp on jena_g1t1_stmt a3  (cost=0.00..6.01 rows=1 width=112)
         Index Cond: (((a3.subj)::text = ("outer".subj)::text) AND ((a3.prop)::text =
'Uv::http://prismstandard.org/namespaces/1.2/basic/volume'::text))
         Filter: (graphid = 1)
(28 rows)
```