

Loading Europeana Metadata into Semantic Repository

by Borys Omelayenko, Ph.D.
europeana.eu

Databases are efficient in storing and querying data records of rigid structure, with database schema fixed at design time. Rich metadata records, used in Europeana, come with a variety of structures. To store them in a database we need to exploit a metadata repository, with Sesame being a leading one. In this paper we discuss how Sesame stores metadata in a database and discuss our experiences in bulk loading of Europeana metadata into Sesame. Open-source Sesame does not provide a bulk loader, and a number of issues need to be solved to build an open-source repository for Europeana metadata based on Sesame with PostgreSQL. We also evaluate another option, a Sesame connector to the commercial Oracle database (TBD).

Introduction.....	3
Metadata.....	3
Usage.....	3
Sesame	4
Native Store.....	4
Sesame on RDBMS.....	5
Populating Sesame on RDBMS.....	6
Statements.....	6
Sequenced mode.....	7
Property tables.....	7
Temporary tables and batches.....	8
Problems.....	8
Problem 1: Statement uniqueness.....	8
Problem 2: URI/Literal uniqueness.....	8
Problem 3: Hash values.....	9
Problem 4: Autocommit.....	9
Problem 5: Batch size.....	10
Problem 6: Memory allocation.....	10
Problem 7: Indexes and optimizations.....	11
Problem 8: Property Tables.....	11
Attempts to modify Sesame.....	11
Managing uniqueness of statements.....	11
Managing uniqueness of values.....	12
Minimal batch size.....	13
Extension problem: Repository Versioning.....	13
Europeana Metadata Repository.....	13
RepositoryDao.....	13
Import module.....	13
Conclusions.....	13
Appendix: Comparisons.....	14
Experimental setup.....	14
Sesame on MySql vs PostgreSQL.....	14
Sesame/PostgreSQL vs Virtuoso.....	14
Links.....	15

Changes

- | | |
|----------|--|
| 14/01/10 | Initial revision, a write-up of the original wiki page on EuropeanaLabs. Makes the wiki page obsolete. Final experiments and description of the RepositoryDao missing. |
| 23/02/10 | Finished discussion on updating Sesame. Starting EMR import module. Final experiments are still missing. |
| 23/04/10 | Introductory edits, title changes, and reference to Sesame Oracle connection. |



Introduction

Metadata

Metadata about cultural objects forms the core asset of Europeana. This metadata is collected from numerous providers, aggregated, and presented to end users in different ways on the Europeana.eu portal.

In the scope of the project we realized that the metadata has the following properties:

- it adheres to various schemas and data models,
- it is owned, and this ownership should be traceable,
- it is interlinked to form a graph.

At the moment of writing, we collected more than 100 mln statements about more than 1 mln objects belonging to Europeana cultural heritage, all provided by dozens different institutions, and stored in 164 XML files, one file per collection, of the total size of nearly 7 Gb. These numbers should increase tenfold in the next two years.

The data is originally structured along the Dublin Core, with each object having about 10 literal properties. However, in the near future we expect data in other formats, specific to cultural subdomains, to appear, together with vocabularies in SKOS or SKOS extensions. Accordingly, we would need to handle the variety of data models, that is associated with RDF.

Finally, metadata bits are coming from different sources. For each statement the original source should be kept, in the spirit of named graphs known in SPARQL.

Usage

In Europeana Version 1 we envisage various services utilizing the data:

- Solr search needs most of the fields to be dumped into Solr records for plain-text search;
- Geo browsing needs several fields, including the locations, plus the corresponding Geonames fields with geo location to present metadata on a map;
- Timeline browser needs several fields, including the dates;
- Data enrichment tools from partner projects, e.g. name recognition tools that need names to map them to directory of people;
- and more to come.

In addition, some services are also contributing to the repository:

- data ingestion tools that add or update collections;
- enrichment services that align the data to external vocabularies, such as Geonames, ULAN, AAT;
- user input that allows (selected) users to add statements directly (in the future).

Despite their variety, these services require just two operations:

- importing metadata, and
- querying metadata in a SPARQL manner.

Basically, the architecture should look like this:

- central metadata repository,
- model-agnostic,
- SPARQL queryable,



- based on a database.

The repository operates in batch. That means that each upload or query request may be fulfilled in minutes, dozens of minutes for large datasets. It is possible to queue the requests and have a singleton repository instance fulfilling them.

To summarize, we are looking for an RDF (<http://www.w3.org/RDF/>) repository capable of running on top of a relational database and supporting SPARQL (<http://www.w3.org/TR/rdf-sparql-query/>) querying. Due to project constraints, we are interested in an open-source solution with good Java integration.

Sesame

There are not that many options that fulfill our requirements, leaving Sesame (<http://openrdf.org>) as the only choice. Sesame is a leading RDF metadata store able to run on top of a relational database. It is distributed under an open source license and is written in Java, that both fit our requirements.

In Europeana we have earlier made a choice for PostgreSQL, a leading open-source database.

Accordingly, in the rest of this paper we explore using Sesame 2.3.0 on Postgres 8.4. Experiments are performed on a iMac Core Duo, 2.93 GHz, 4Gb RAM.

Native Store

Let us use Sesame in the fastest mode, using Sesame native store instead of a database. First, we create a Repository:

```
NativeStore sail = new NativeStore(new File("data"));
Repository repository = new SailRepository(sail);
```

Here `data` is the path where Sesame would persist the data.

Beware! If this directory already exists then Sesame would load all the data from this directory. That means, at the first run a repository is created in `data`. At the second run Sesame would not re-create a repository, but import all the data persisted in `data` during the first run. Thus, starting with a populated repository. To ensure that the repository is clean before each run we need to clean the `data` directory prior creating a repository there.

Then we add data to the repository with the following code:

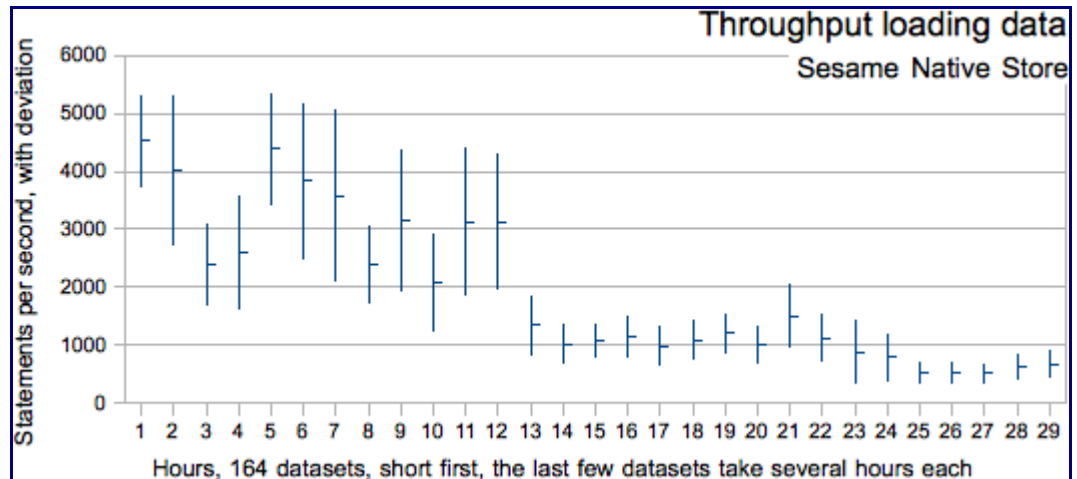
```
ValueFactory factory = repository.getValueFactory();
RepositoryConnection connection = repository.getConnection();

// the parameters below are all of type String
connection.add(
    factory.createStatement(
        factory.createURI(recordUri),
        factory.createURI(propertyUri),
        factory.createLiteral(literalString),
        factory.createURI(collectionUri)));
```

It is important to use factory methods, such as `createURI` instead of instantiating implementation classes, such as `RdbmsURI`. Among other benefits, factory methods cache the URI objects, that is quite efficient in practice.

Now we are ready to try loading our 100 mln triples on Sesame Native Store. We can leave it running overnight, and we can take a long night. A short answer is: we loaded it all in 29 hours. What was it doing that long?

The following diagram illustrates throughput on loading data:



Half way it becomes quite slow: 20% of the data are loaded in 50% of time. A possible reason for that is that it may hit memory limits. Hopefully, giving the Java VM more memory would help here.

Qua timing, this result is acceptable: middle-size collections are loaded in minutes.

Accordingly, Sesame Native Store is performing, but suffers from everything it should suffer by not being a database:

- maintainability: backup;
- scalability: replication, clustering;
- transparency: in-house solution low on docs;
- accessibility: no low-level access.

This is a promising start, however, not yet a solution for Europeana, a maintainable production service.

Sesame on RDBMS

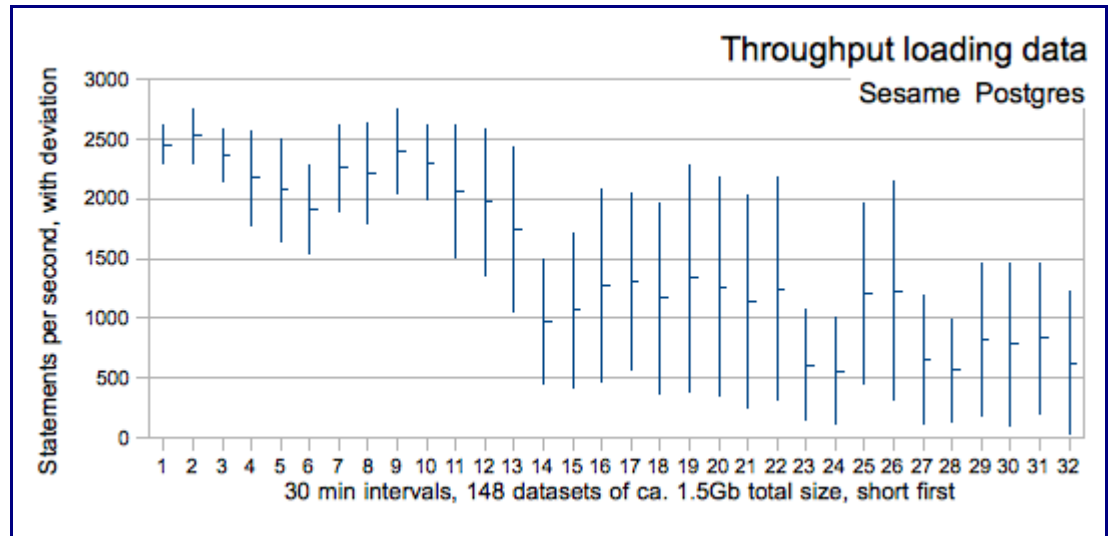
Let us try Sesame on a PostgreSQL database. Again, we create a Repository with the following code:

```
PgSqlStore sail = new PgSqlStore();
sail.setUser(USER);
sail.setPassword(PASSWD);
sail.setDatabaseName(DATABASE);

Repository repository = new SailRepository(sail);
```

Again, we have the same behavior as with Native Store: Sesame does not clean the database for us. To clean it, we need to execute SQL statements `DROP DATABASE` followed by a `CREATE DATABASE`.

Now we are ready to try loading our 100 mln triples on Sesame on PostgreSQL. 32 hours later it is not even half way!



It seems that it does not perform out-of-the box. Let us continue with an in-depth discussion on populating Sesame on RDBMS.

Populating Sesame on RDBMS

Let us come back to basics and explore what Sesame does to store a metadata statement. For historical reasons statements are often referred to as triples, and we may use both as synonyms.

Statements

Sesame stores metadata statements represented as quadruples

<context, subject, property, object>

where:

- context is the source of the statement, an URI of a graph where this statement belongs to;
- subject is an URI of a metadata object;
- property is an URI of a property of the object;
- object is the value of the property, either a literal (a string) or a link to another object (URI).

To store it in a database we should execute something similar to the following SQL statement

```
INSERT INTO triples VALUES (context, subject, property, object)
```

This would do the job in a very dirty way. It would create a database with excessive duplication. The same URIs would be copied again and again, for each statement, degrading performance and eating up computer memory. Standard database solution would be to store values (URIs or literals) in separate tables, assign them unique IDs, and use these IDs everywhere. In the database world this is called database normalization.

Sesame stores statements in a normalized way. It creates several tables:

- uri_values to store URI values and their IDs,
- label_values to store literal values and their IDs,
- long_uri_values to store long URI values and their IDs,
- long_label_values to store long literal values and their IDs,
- and other xxx_values tables for BNodes, datatypes, etc.

Then, table triples stores only the IDs of the contexts, subjects, properties, and objects. It also

disallows duplicates, in accordance with the RDF specification that assumes that all triples are unique.

Sequenced mode

Here we come to the way the IDs are generated. Theoretically speaking, each URI or literal should have a unique ID that is stored in the `triples` table. However, it is very expensive to guarantee this uniqueness because before inserting an URI we need to iterate through all other URIs and compare.

Sesame comes with a workaround: it uses a hash function over the URI or literal value as its ID. Hash codes for different values are almost always different. There is a small chance that two values would end up having the same ID. With a perfect hash function this chance should be $1/\text{max_int}$ that is lower than the chance of hardware failure and many other events.

Accordingly, Sesame uses hash functions as IDs of URI's and literals. To control this, Sesame can create the database in two modes: `sequenced` or `not`.

In the sequenced mode it keeps the `triples` table neat:

- the IDs are numbered from 1, 2, etc
- the first row in the `triples` table is likely to look like 1, 2, 3, 805306369 (as Sesame assigns longer IDs to literals)
- there is a separate table `hash_values` with IDs and their hash values

In the non-sequenced mode it puts hash codes directly into the `triples` table. In both modes Sesame still uses hash codes as IDs.

Property tables

Imagine, our triple store uses 100 properties, and we want to query just two of them, such as:

```
SELECT ?id
FROM {?id <department> <IT> . ?id <position> <senior> }
```

For execution, this query should be translated to something similar to the following SQL:

```
SELECT id
FROM triples as t1, triples as t2
WHERE t1.property = 'department' and t1.object = 'IT' and
t1.subject = id
and t2.property = 'position' and t2.object = 'senior' and
t2.subject = id
```

This means a join on the whole table `triples` that can be quite large. And most of this join is useless as we are only interested in the values of just two properties.

To improve this, Sesame can create separate tables for each property. This is controlled by property `setMaxNumberOfTripleTables`, where values below 2 allow Sesame to create as many of them as needed.

These tables are named according to the format `name_ID`, for example, property `http://europeana.eu/type` may be named `eutype_3`, where `eu` is extracted from the namespace, `type` is the name, and 3 is the ID of this property.

In the sequenced mode these IDs start with 1 and the tables have 1- or 2-digit IDs. In the non-sequenced mode table names include large hash codes, such as `eutype_1069724910754998485`.

Temporary tables and batches

Sesame has a thought-through mechanism of inserting data. Inserts into each of the `xxx_value` tables is done in a separate thread, where individual inserts are aggregated into batches of default size of 8K.

Sesame does not write to the `triples` and `values` table directly: it first writes to a temporal table and then copies it with the `INSERT ... SELECT ...` statement. It may be caused by the fact that Sesame RDBMS stack was originally developed for MySQL, that does not support transactions. And Sesame had to have its own transaction support, in this case by creating a temporal table and allowing to rollback.

Problems

Let us look at the technical problems that affect Sesame performance.

Problem 1: Statement uniqueness

Here we hit the first performance bottleneck. Before adding a new triple, Sesame needs to check if this triple is already in the repository. It does something like the following:

```
INSERT INTO triples
  SELECT DISTINCT ctx, subj, obj, expl FROM transaction_statements tr
  WHERE NOT EXISTS (SELECT * FROM triples st
  WHERE st.ctx = tr.ctx AND st.subj = tr.subj AND st.obj = tr.obj AND
  st.expl = tr.expl)
```

Here `transaction_statements` is a temporal table containing triples added during the current transaction. The problem comes with the nested query `WHERE NOT EXISTS (SELECT * FROM triples ...)` performed on the large table `triples`.

Basically, Sesame has no other way of ensuring uniqueness but to check for it at the database level. If an application can guarantee triple uniqueness, then these expensive checks can be moved outside of Sesame.

Problem 2: URI/Literal uniqueness

Not only triples should be unique, but each URI or literal in tables `uri_values`, `label_values`, `long_uri_values`, `long_label_values`.

Sesame handles it in the same way, e.g. for literal values:

```
INSERT INTO label_values (id, value)
  SELECT DISTINCT id, value FROM insert_label_values tmp
  WHERE NOT EXISTS (SELECT id FROM label_values val WHERE val.id =
  tmp.id)
```

And we have the same nested `SELECT` here. Not much can be done to remove it. If a value occurs for the first time in an upload, it may have already been used in other records.

The negative effect of this nested `SELECT` is reduced by using Sesame factory caches:

- `context` URI is created once per collection upload and is efficiently cached;
- `subject` URI is created once per metadata record, and its cached copy is used for all properties of the record;
- `property` URI is efficiently cached as well, as properties are standardized, and the whole metadata collection uses a few dozens of them in total;
- `object` URI or literal values form a potential bottleneck, especially if they do not repeat

much.

These caches do not guarantee that a new value is there, and Sesame needs to check for value uniqueness at the database level. However, if an application can guarantee that a value is added for the first time and is always found in cache each time after, then these expensive checks can be moved outside Sesame.

Problem 3: Hash values

In the sequenced mode, Sesame maintains a table with hash values for URIs/literals. Before a record is committed, Sesame gets all URI/literal values mentioned there and generates their hash values. Before assigning a value, it checks if that hash value is already used. It results massive execution of the queries, similar to the following:

```
SELECT id, value
FROM hash_values
WHERE value IN ('3517122232376268977', '4245190426859369984',
$3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13, $14, $15)
```

If a hash value is not found in the table, a following `INSERT INTO hash_values (id, value) VALUES ($1, $2)` would be called.

Solution: the maintenance of hash values can be switched off by setting parameter `sequenced` of a `sail` to `false`.

```
PgSqlStore sail = new PgSqlStore();
sail.setSequenced(false);
```

Problem 4: Autocommit

Database commits are typically expensive. As a rule of thumb, infrequent large commits should be preferred over frequent short ones on populating a database. By default a Sesame `RepositoryConnection` is in autocommit mode. That means that each triple is committed individually.

It makes sense to switch it to manual commits:

```
connection.setAutoCommit(false);
```

And to commit manually, after adding a few hundreds or thousands metadata statements:

```
connection.commit();
```

In the Sesame RDBMS module, method `RdbmsTripleRepository.commit()` has an interesting feature: after each commit it switches it back to the autocommit mode. A solution can be in switching it back to the manual commit mode:

```
connection.commit();  
connection.setAutoCommit(false);
```

Problem 5: Batch size

With autocommit under control, we can create reasonably large transactions of, say, 5.000-10.000 statements.

Between transaction commits, Sesame creates database/jdbc batches with SQL INSERT statements. Batch size is limited with `TransTableManager.BATCH_SIZE` for triples and `ValueTable.BATCH_SIZE` for value tables (it is also used as the value cache size), with the default value of 8K. If a batch exceeds this limit, a new batch is created.

Database operations are time-consuming, and they can be done without delaying the main application. Accordingly, Sesame creates a queue with batches to be inserted, and a separate thread, both in class `ManagerBase`. This thread calls method `insertThread()` that continuously examines the batch queue and flushes each flushable batch it finds in the queue.

Maximal batch size is limited, but its minimal size is 1. The insert thread flushes everything it can flush, and a smaller batch is perfectly flushable. Practically, it means that, if the insert thread gets enough CPU time, it can massively flush batches of size 1. This makes the whole idea of batch data insertion as good as useless.

Solution: modification of Sesame.

Problem 6: Memory allocation

Databases such as PostgreSQL and MySQL are coming with very conservative presets. Specifically, memory settings should be increased.

For PostgreSQL the following parameters are essential:

- `shared_buffers` - RAM made available for Postgres;
- `effective_cache_size` - RAM for cache;

For MySQL the following parameters are essential:

- `key_buffer_size` - RAM made available for MySQL using MyISAM tables;
- `innodb_buffer_pool_size` - RAM made available for MySQL using InnoDB tables.

For more info on the databases you may search Internet for `performance postgres` or `performance mysql` where several good pages are easy to find.

On Linux systems you may have system-wide constraint on the amount of memory to be allocated by its application launcher, historically stemming from BSD code. These are parameters `kern.sysv.shmmax` and `kern.sysv.shmall` of Linux kernel. They are set in file `/etc/sysctl.conf`. Here is mine, allowing 2Gb of RAM:

```
kern.sysv.shmmax=2147483648  
kern.sysv.shmmin=1  
kern.sysv.shmmni=256  
kern.sysv.shmseg=64  
kern.sysv.shmall=2147483648
```

Arbitrary values, such as 2000000000 are not accepted (and reported to the syslog). Apparently it should be a multiplier of some block size. To give it 4Gb you need to multiply 2147483648 by 2.



Problem 7: Indexes and optimizations

The rule of thumb in populating a database is: It is always cheaper to rebuild an index over the whole table, when it is all populated, then to update the indexes at each insert. In fact, these index updates form one of the worst performance bottlenecks, as the system starts an expensive index update after each new insert. We need to drop all indexes before starting a massive update, and recreate them after.

The same holds for optimizations, such as Postgres `VACUUM`.

A technical solution here lays in adding drop/create index, and enable/disable optimizations functionalities to Sesame.

Problem 8: Property Tables

If all statements would be stored in a single database table (Sesame names it `triples`) then each query would involve an expensive join on the same large table. This would seriously deteriorate query performance. A solution would be in separating the statements into multiple tables, each table storing statements with a specific property.

Sesame Sail has a parameter, called `setMaxNumberOfTripleTables`. Values 2 and above would set the maximal number of tables, one per property, that Sesame would create before it would start dumping them to `triples`. Values 0 and negative are interpreted as 'unlimited number of triples can be created'.

A few considerations on creating more tables:

- it improves query performance, hence it is recommended;
- it decreases data load throughput, as each INSERT statement is split into many INSERT statements, one per property. Ten properties per record are common in our data. We did brief testing that shows marginal performance gain on reducing the number of tables. Accordingly, having a dozen tables should not be a bottleneck.
- dynamically-created tables are difficult to replicate, e.g. PostgreSQL Slony requires an explicit list of tables to be replicated;
- it only makes sense to create separate tables for properties that occur reasonably often in both data and queries.

A solution to this issue may lay in creating tables for a controlled set of properties. Sesame has `TripleTableManager.getPredicateTable()` method responsible for creating new tables. It may be changed to allow a list of tables through.

Attempts to modify Sesame

Managing uniqueness of statements

Sesame manages statement uniqueness at the database level that causes Problem 1: Statement uniqueness. And this is the only way to do it in a general-purpose metadata store. However, uniqueness can be better managed at the application level, boosting Sesame data upload performance.

Sesame generates SQL queries for inserting triples in `TransactionTable.buildInsertSelect()`. This method can be modified as following, commenting out the `WHERE NOT EXISTS` check:

```
protected String buildInsertSelect() throws SQLException
{
    String tableName = triples.getName();
    StringBuilder sb = new StringBuilder();
    sb.append("INSERT INTO ").append(tableName).append("\n");
    sb.append("SELECT DISTINCT ctx, subj, ");
    if (triples.isPredColumnPresent()) {
        sb.append("pred, ");
    }
    sb.append("obj, expl FROM ");
    sb.append(temporary.getName()).append(" tr\n");
    /*
    sb.append("WHERE NOT EXISTS (");
    sb.append("SELECT * FROM ");
    sb.append(tableName).append(" st\n");
    sb.append("WHERE st.ctx = tr.ctx");
    sb.append(" AND st.subj = tr.subj");
    if (triples.isPredColumnPresent()) {
        sb.append(" AND st.pred = tr.pred");
    }
    sb.append(" AND st.obj = tr.obj");
    sb.append(" AND st.expl = tr.expl");
    sb.append(")");
    */
    return sb.toString();
}
```

An attempt to insert a duplicating statement would cause an index violation exception (as the table has a primary index defined over all fields: context, subject, property, and value). However, adding a new statement would go without this, pretty expensive, check.

Before adding a collection, EMR removes all old statements that belong to the collection, and thus reduces uniqueness management from the whole database to this specific collection. Then, for each subject, it ensures that there are no duplicating property-object pairs.

Managing uniqueness of values

Sesame manages value uniqueness at the database level, same as it does for statements, that causes Problem 2: URI/Literal uniqueness. To solve it, we need to ensure that only unique values are inserted into Sesame and that we never try to insert a value that is already present in the database.

When populating Sesame with a collection, if a value was recently added, then it will be in the corresponding factory cache. By default, as many as 8,192 (default cache size) most recent values would never be inserted again, but found in cache instead.

In EMR we extend cache size to accommodate all values that were ever inserted. This guarantees that if a value is inserted once then its next occurrences would be coming from the cache, and would not be inserted to the database again.

This solves the problem in the scope of a collection. However, if a value occurs in a collection for the first time, but if it has been previously inserted in the database, it would not be found in the cache, and would be inserted again with a key violation. To solve this, for each collection EMR guarantees that each value is in cache if it is in the database. To ensure this, it first scans the collection file and extracts all values mentioned there. Then it selects those that are already present in the database and caches them (method `cacheExistingResources`).

As a side effect, the cache can grow large, as large as a collection. We allow this with the assumption that each collection fits in memory, that is typically the case.



Minimal batch size

To solve Problem 5: Batch size we had to update Sesame to ensure that small batches are not flushed to the database.

Extension problem: Repository Versioning

For each statement, URI, or a literal, Sesame creates Java objects that it manipulates before they are serialized into SQL INSERT commands. And Sesame caches these objects to prevent unnecessary (and extremely expensive) SQL queries to boost performance. However, there is a problem Sesame has to address: what to do if a statement has been removed from a repository, but the corresponding Java objects are still hold in caches, as if the statement is still in the database.

Sesame maintains repository version number: an integer that is incremented after each removal operation, and not touched after operations that cannot remove data. Thus, each increment of repository version means that something may have been removed.

When Sesame creates a Java object for an URI or a literal, it assigns it a version, equal to the current repository version. Before reusing such an object, Sesame compares its version with the current repository version at the moment of reuse. A difference means that something was deleted from the repository, and there is a chance that the Java object refers to something that is absent in the database. In this case Sesame does not reuse the object and re-creates it.

These checks on the versions is rooted deeply in Sesame code and alternating this behaviour requires substantial updates to Sesame code base.

Europeana Metadata Repository

This describes the Europeana Metadata Repository (EMR) developed for Europeana Version 1.0. In the repository we address the problems identified above to make performance of Sesame on Postgres matching Europeana requirements.

Technically, EMR consists of a Spring DAO object and a few classes.

We had to develop custom module to perform massive import to Sesame database.

RepositoryDao

The main EMR class is `RepositoryDao` that provides the contract for metadata repository access.

Methods `createIndexes()`, `dropIndexes()` and `setOptimized(boolean optimized)` resolve Problem 7: Indexes and optimizations.

Method `importEseCollection()` should be used to import a new collection. Let us explore how does it solves the problems. Mostly, we will be talking about class `RepositoryDaoImpl` and classes around it.

Import module

Here we describe the import module (under construction).

Conclusions

Sesame on Postgres comes out of the box with poor performance. Mostly, it is caused by the fact that, as a general-purpose repository, Sesame performs a number of checks and operations at the database level. By moving these to the application level we achieved substantial performance improvement on populating the repository.

At the moment of writing we achieved good preliminary results. To finalize it we need to fix some issues with the code (somehow Postgres halts on DROP INDEX) and then we can do a nice and neat

experiment with final statistics.

Appendix: Comparisons

Experimental setup

To compare the the tools we run them on our all 164 collections, starting with the smallest one of just 4 objects up to the largest one of 300.000 objects.

Measuring performance of a repository is tricky:

- number of statements (triples) does not reflect on their size, as some are longer than the other
- size of triples does not reflect on their diversity, as they may be repeating, that often happens with cultural metadata
- diversity of triples is difficult to assess as there are no duplicating triples, they all differ a bit
- file size does not tell anything about its complexity.

However, with a large number of different collections, we can expect that file size is a reasonable candidate to compare different systems.

Sesame on MySql vs PostgreSQL

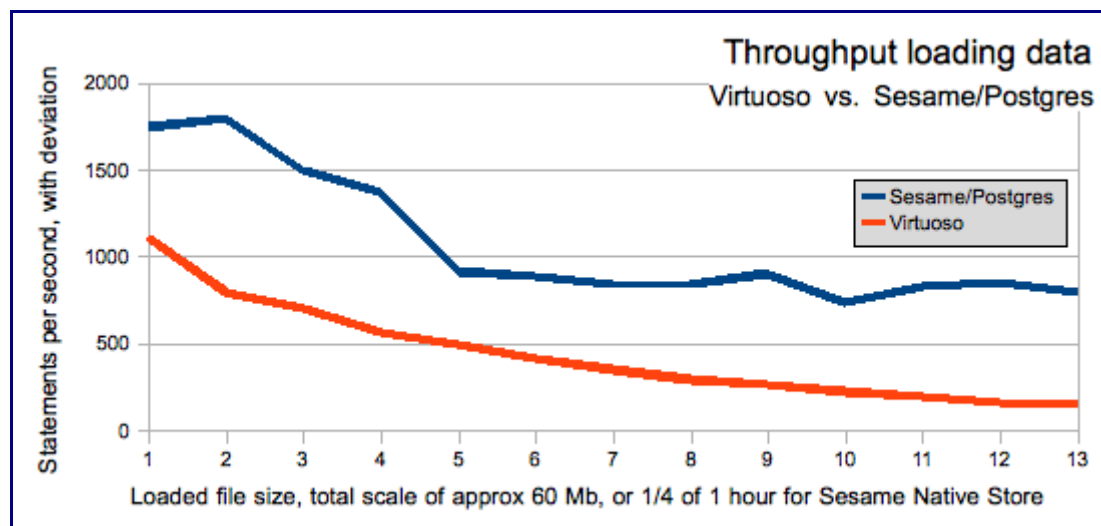
We did a test on loading the first 1.5Gb of our data to both, with hash, similar db setup:

- MySql done in 11:45 min
- PostgreSQL done in 14:25 min

Postgres is known for being slower than MySQL, e.g. <http://www.sqlite.org/speed.html>

Sesame/PostgreSQL vs Virtuoso

We also examined Virtuoso (<http://virtuoso.openlinksw.com/>), while it does not make such a good match with Europeana requirements as Sesame. Example of loading two files, 28 Mb each on a clean Virtuoso triple store (numbers are seconds needed to upload 5.000 statements on iMac DualCore/4Mb: Sesame on (a very unoptimized configuration of) Postgres looks a more promising alternative to Virtuoso:



In this run we used Sesame out of the box, prior to addressing any of the problems described in this paper. Both Sesame and Virtuoso are seriously outperformed by Sesame Native Store.

Sesame on Oracle

Oracle has Semantic features built in to their database, and there is a non-open-source Sesame connector to Oracle with bulk loading feature. We managed to make it running (in fact, it was easy) but did not manage to get acceptable results yet. TBD.

Links

<http://developer.postgresql.org/pgdocs/postgres/populate.html>

<http://www.postgresql.org/docs/current/static/runtime-config-resource.html>

