Comparing Relational and Graph Databases for Pedigree Data Sets

Graham Kirby
School of Computer Science
University of St Andrews
Fife KY16 9SX, Scotland
gnck@st-andrews.ac.uk

Conrad de Kerckhove
School of Computer Science
University of St Andrews
Fife KY16 9SX, Scotland
cfedk@st-andrews.ac.uk

Ilia Shumailov
School of Computer Science
University of St Andrews
Fife KY16 9SX, Scotland
is33@st-andrews.ac.uk

Jamie Carson
School of Computer Science
University of St Andrews
Fife KY16 9SX, Scotland
jkc25@st-andrews.ac.uk

Alan Dearle
School of Computer Science
University of St Andrews
Fife KY16 9SX, Scotland
alan.dearle@
st-andrews.ac.uk

Chris Dibben
Longitudinal Studies Centre Scotland
Universities of St Andrews &
Edinburgh
cjld@st-andrews.ac.uk

Lee Williamson
Longitudinal Studies Centre Scotland
Universities of St Andrews &
Edinburgh
lepw@st-andrews.ac.uk

Abstract

Increasingly large family pedigree data sets are being constructed from routine civil and religious registration data in various parts of the world. These are then being used in health, social and genetic research in a variety of different ways. Often the type of questions that are being asked involve complex queries, such as the degree of relatedness between multiple sets of individuals, and involve traversing through the data typically multiple times. There is therefore an important issue of efficiency of querying. In this paper we evaluate the suitability of two classes of database, relational (MariaDB) and graph (Neo4j), for storing and querying pedigree datasets representing millions of individuals. We report results of measurements of scalability, query performance, and the ease of query expression, using synthetic datasets.

1 Introduction

The ESRC-funded project *Digitising Scotland* is in the process of transcribing the text of the 29 million vital events record images (births, marriages and deaths) for all people in Scotland between 1855 and 1973. This will greatly facilitate research access to individual level information on some 18 million individuals, a large proportion of those who have ever lived in Scotland between 1855 and the present day.

The data set will be linked to existing external longitudinal data sets. However in this paper we are primarily concerned with the internal linkage process required to create a pedigree data set representing genealogical relationships across the entire population.

The primary aim of this linkage process is to produce a resource of interest to researchers in geography, health, history and related disciplines. Subsidiary technical aims include:

- support for retention of the provenance of linkage decisions;
- support for modelling, querying and visualising uncertainty in the data and inferred links;
- support for tailoring coding and linking algorithms to particular historical periods;
- maximising automation of the overall process.

This paper reports the results of some exploratory experiments to start to understand the relative capabilities of relational and graph databases, both during the linkage process, and in supporting queries on the resulting linked data.

Popular implementations of both database types—MariaDB and Neo4j—were used to store synthetic population data sets of various sizes and execute simple genealogical queries, and execution times recorded. To give context, Table 1 shows the expected scale of the full data set.

	Records	Estimated Size
births	14 million	1.9 GB
deaths	11 million	2.3 GB
marriages	4.2 million	0.8 GB

Table 1. Scale of source data.

2 Related Work

A comprehensive qualitative analysis of the relative suitability of a number of graph databases for storing genealogical structures is given in (Perea, 2013). That work does not address the qualitative evaluation of storage costs and query execution performance.

General systems for generating synthetic records for benchmarking and testing record linkage approaches are described in (Christen and Vatsalan, 2013; Ioannou, Rassadko, and Velegrakis, 2013; Talburt, Zhou, and Shivaiah, 2009). The data generation tool used in this work is specialised for genealogical population structures, and the global configuration parameters that are supported reflect this.

3 Experiments

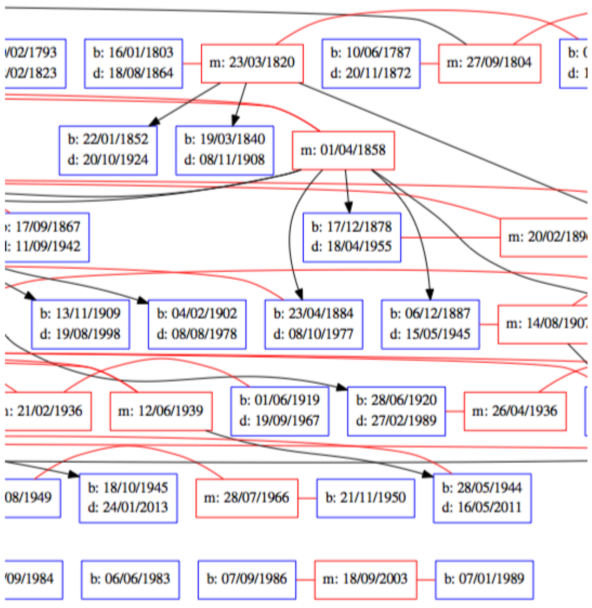
3.1 Synthetic Data

In order to obtain family tree data sets of various sizes and with controlled characteristics, a synthetic population generator was developed. The generator accepts the following parameters:

- size of population
- date range within which all events occur
- distribution of birth rates over time
- distributions of age at marriage, length of marriage, number of marriages[1]
- distributions of age at childbirth, inter-child gap, number of children
- distribution of age at death
- distribution of immigration rate over time[2]

When executed, the generator builds an in-memory data structure that can be exported to a database, to a GEDCOM file (GEDCOM, 2013), or to a graph file in DOT format (for the Graphviz graph visualisation software (Graphviz, 2013)).

The population generator is implemented in Java and is freely available (Digitising Scotland 2013). **Figure 1** illustrates a fragment of a small generated population rendered by Graphviz. Boxes represent individuals and marriages.



b: 16/01/1803 d: 18/08/1864
m: 23/03/1820
b: 10/06/1787 d: 20/11/1872
m: 27/09/1804
b: 22/01/1852 d: 20/10/1924
b: 19/03/1840 d: 08/11/1908
m: 01/04/1858
b: 17/12/1878 d: 18/04/1955
b: 13/11/1909 d: 19/08/1998
b: 04/02/1902 d: 08/08/1978
b: 23/04/1884 d: 08/10/1977
b: 06/12/1887 d: 15/05/1945
m: 12/06/1939
b: 01/06/1919 d: 19/09/1967
b: 28/06/1920 d: 27/02/1989
m: 26/04/1936
b: 18/10/1945 d: 24/01/2013
m: 28/07/1966
b: 21/11/1950
b: 28/05/1944 d: 16/05/2011
b: 06/06/1983
b: 07/09/1986
m: 18/09/2003
b: 07/01/1989

Figure 1. Sample graph rendering of synthetic population.

In the experiments reported here, the population generator was used to create synthetic populations of between 1 and 5 million individuals.

[1] For simplicity all partnerships that produce children are considered as marriages.

[2] This controls the number of people in the population that do not have parents in the population.

The generator allows distributions to be specified in a fairly flexible way. For these experiments, relatively simple distributions were used. For example, dates of birth were uniformly randomly distributed within the overall date range, while ages at death were taken from modern tables published by the UK Office for National Statistics {ONS:wv}.

3.2 Databases

The relational database chosen was MariaDB (2013), a popular open source implementation. Hibernate ORM (Hibernate, 2013) was used to simplify the task of mapping between in-memory objects and relational tables.

The graph database was Neo4j (2013), an open source Java-based graph database. In common with other graph databases, Neo4j models data in the form of graphs comprising nodes and edges. Both nodes and edges can be annotated with labels. Neo4j provides a succinct query language, Cypher, that is designed to make it easy to express graph traversal queries, and efficient to execute them. For example, it is very simple to express a query to find all of the nodes within a given number of edges of a given node—a task that is cumbersome in SQL. An example is given in the next section.

Figure 2 shows how the disk storage used by the two databases varies with the size of the population stored.

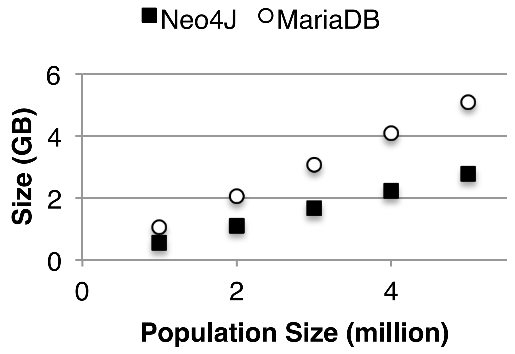


Figure 2. Database size on disk.

The disk usage was measured by examining the databases' implementation files directly in the underlying file system.

It can be seen that in both cases the storage occupied varies roughly linearly with the population size, with the relational database growing about twice as fast.

3.3 Queries

The following simple queries were selected, with the intention of being representative of the type of expected queries on the linked genealogical graph:

- find the children of a given person
- find all the ancestors of a given person to a given height in the tree, i.e. going back a given number of generations
- find all the descendants of a given person to a given depth in the tree, i.e. going forward a given number of generations
- find all the relatives of a given person to a given distance in the graph: this includes ancestors, descendants, siblings, cousins etc

Figure 3 shows the Cypher query, for the chosen schema, to find all relatives to distance 10 of a person with a given identifier.

```
start person = node(ID)
match person-[:PARTNER|CHILD*1..10]-
      relatives
where relatives.__type__! =
      'Neo4jPerson'
return distinct relatives
```

Figure 3. Example query in Cypher.

In contrast, the equivalent query for the MariaDB schema has a significant procedural component, and is therefore much more complex. A slightly simplified version is shown in **Figure 4**. This is not intended to be legible, but simply to give an indication of the complexity.

```
final Set<Person> finalList = new HashSet<Person>();
final Set<Person> personsFromLastIter = new HashSet<Person>();
personsFromLastIter.add(person);
for (int i = 0; i < radius; i++) {
 final Set<Person> currentIteration = new HashSet<Person>();
 for (final Person p : personsFromLastIter) {
  final long p1 = p.getId();
  final EntityManager eManager = ConnectionManager.getEManager();
  final Long parentPartnershipID = getPartnershipIDByChildID(p1);
  final Query currentQuery = eManager.createQuery(
   "SELECT child.id FROM Partnership AS mp JOIN
    mp.children AS child WHERE mp.id=:partnership_id AND
    child.id!=:child_id");
  currentQuery.setParameter("child_id", p1);
  currentQuery.setParameter("partnership_id", parentPartnershipID);
  final List<Long> tempIds = currentQuery.getResultList();
  final Set<Long> siblingIDs = new HashSet<Long>(tempIds);
  final Set<Person> finalSet = new HashSet<Person>();
  for (final Long id : siblingIDs) {
   final Person currPerson = getPersonById(id);
   finalSet.add(currPerson);
  }
  final Set<Person> siblings = finalSet;
  final Set<Long> parIDs = getParentsIDs(p.getId());
  final Set<Person> parents = new HashSet<Person>();
  for (final Long id1 : parIDs) {
   final Person currPerson = getPersonById(id1);
   parents.add(currPerson);
  }
  final Set<Person> children = getChildren(p);
  final Set<Person> partners = getPartners(p);
  currentIteration.addAll(siblings);
  currentIteration.addAll(parents);
  currentIteration.addAll(children);
  currentIteration.addAll(partners);
 }
 finalList.addAll(currentIteration);
 personsFromLastIter.clear();
 personsFromLastIter.addAll(currentIteration);
}
finalList.remove(PopulationUtilsCommons.getPersonById(person.getId()));
return finalList;
```

Figure 4. Example query in Java and embedded SQL.

Figure 5 shows the times taken to retrieve the details of 20 randomly selected people and their children, all performed 100 times so that the elapsed times measured were reasonably long relative to the granularity of the timer. The error bars show the confidence intervals for 20 trials, after a warm-up period of 5 initial trials.

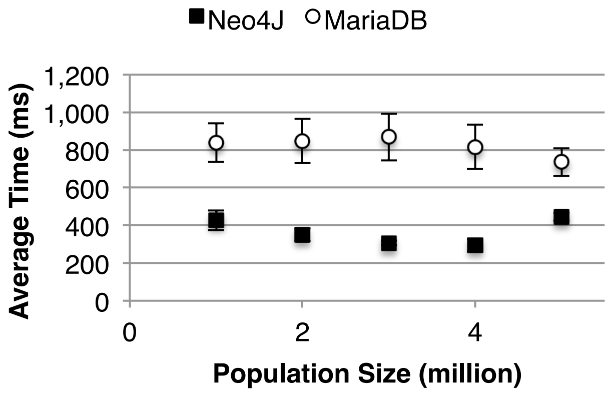


Figure 5. Time to retrieve 20 people and their children, 100 times, 20 warm trials.

It can be seen from the relative sizes of the confidence intervals that for Neo4j the times across repeated trials are more closely clustered than for MariaDB, and that there are no clear trends relating execution time to size. It is hypothesised that the variations are in part due to random differences in the population topologies. So for example, the number of children retrieved is likely to vary for different database sizes.

Figure 6 shows the times taken to retrieve the details of all ancestors to a maximum height of 10 generations for 20 people. Again the timings were for sequences of 100 repetitions, and the entire process was repeated 20 times.

As with the previous experiment, there is no obvious correlation between database size and elapsed time, and it seems likely that features of the randomly generated population topologies exert the strongest influence.

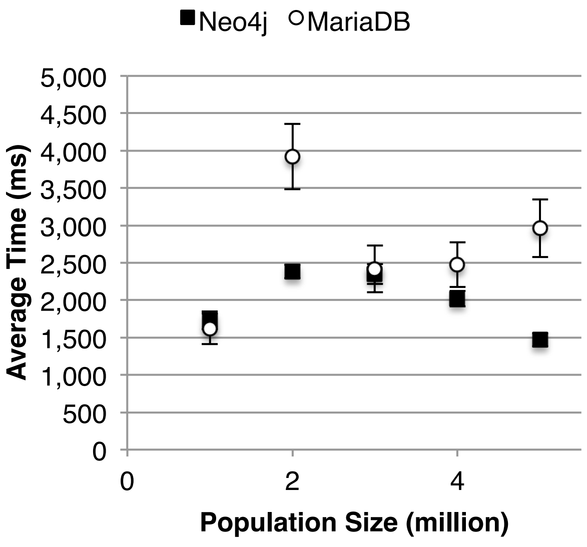


Figure 6. Time to retrieve ancestors to height 10 for 20 people, 100 times, 20 warm trials.

Figure 7 shows the results of a similar experiment to the previous one, in which descendants are retrieved rather than ancestors. Here in 4 out of the 5 population sizes Neo4j performs significantly faster than MariaDB.

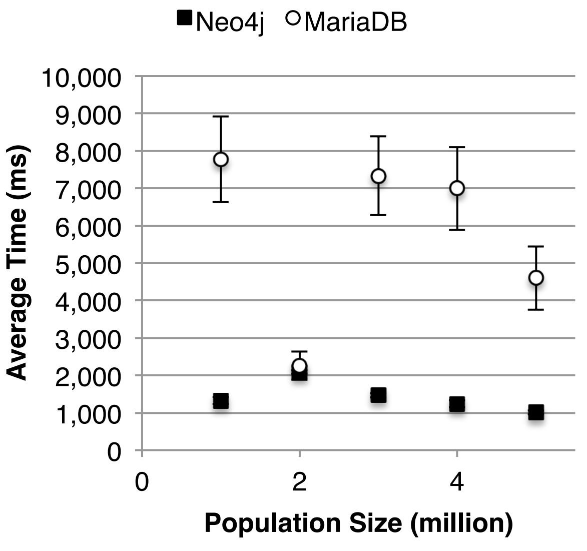


Figure 7. Time to retrieve descendants to depth 10 for 20 people, 100 times, 20 warm trials.

Figure 8 shows the times to retrieve all blood relatives to distance 5, where each step involves navigating from a parent to child or vice-versa. Thus siblings are separated by a distance of 2.

Execution times are significantly longer than in previous experiments, so the process is not repeated within each measurement, and only 10 people are included. Here Neo4j shows a highly significant speed advantage over MariaDB.

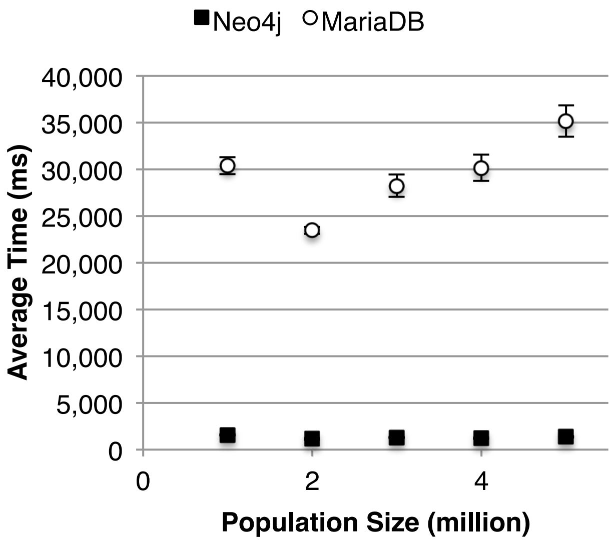


Figure 8. Time to retrieve relatives to distance 5 for 10 people, 20 warm trials.

Figure 9 shows how the time taken to execute the ancestors query varies with the height limit, for Neo4j. There is no particularly clear trend. For completeness, measurements were taken up to 20 generations. However, given the parameters of the population it is unlikely that there are many occurrences of 20 generations. Again it seems likely that population topology variation accounts for most of the effects.

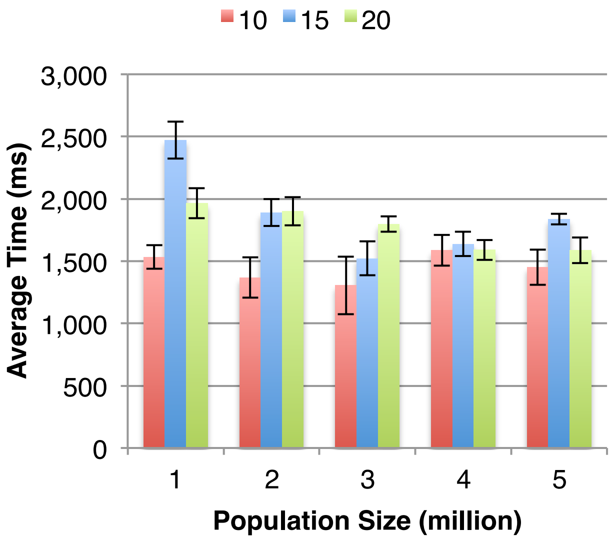


Figure 9. Time to retrieve ancestors to various heights for Neo4j, 100 times, 10 warm trials.

Figure 10 shows the equivalent times for MariaDB. Note the different vertical scale from **Figure 9**. Here the lack of a clear trend is even more apparent. As in previous experiments, the variation between trials is significantly greater than for Neo4j.

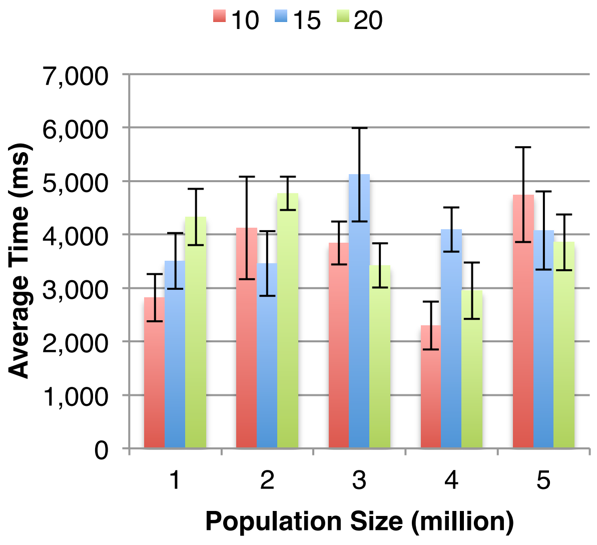


Figure 10. Time to retrieve ancestors to various heights for MariaDB, 100 times, 10 warm trials.

Finally, **Figure 11** shows how the time taken to execute the relatives query varies with the distance limit, for both databases. There are hints at an exponential increase with MariaDB. The growth curve is not discernible with Neo4j, but certainly it performs much more quickly and consistently than MariaDB at the distances measured.

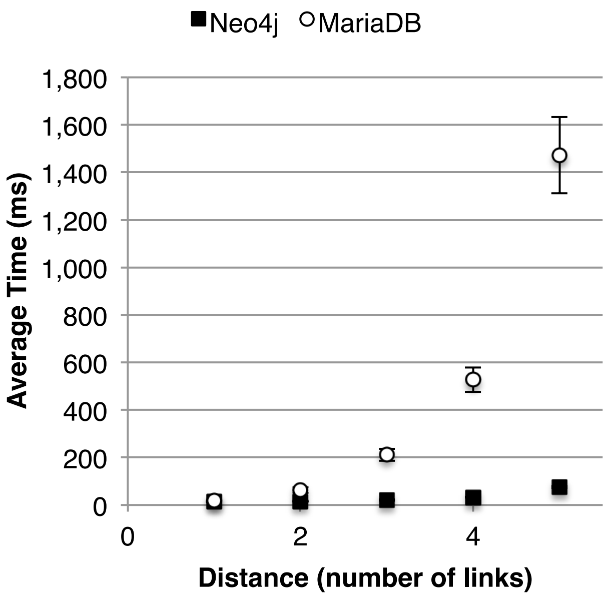


Figure 11. Time to retrieve relatives to various distances, for 10 people, 9 warm trials.

4 Conclusions

The focus in these exploratory experiments has been on support for storage and querying of linked pedigree graphs. In this respect, Neo4j out-performed MariaDB in terms of storage efficiency, execution time and ease of expressing graph-traversal queries.

Further consideration will have to be given to whether these or other attributes are the most significant during the initial and ongoing linkage process. One possibility is to use both types of database in parallel.

Plans for further work include:

- Generation of synthetic vital event records from the synthetic population.
- Addition of realistic names and other attributes to the synthetic population using tools such as FEBRL (Christen and Pudjijono, 2009)
- Incremental population generation; the current approach builds the entire topology in memory before exporting, which represents a bottleneck (which can be addressed to some extent by generating multiple disjoint populations and cross-linking).
- Measurement of more complex queries, such as nearest common ancestor, coefficient of relatedness, and mean coefficient of relatedness for a set of people.
- Measurement of a relational database using stored procedures.
- Measurement of other database types, implementations and query languages.
- Investigation of the effects on performance of the more complex graph structures that will be required in order to represent multiple possible family links, given the uncertainties inherent in the linkage process.

Acknowledgements

This work was supported by ESRC grant ES/K00574X/1 Digitising Scotland. We thank the anonymous referee for their very helpful comments.

References

Peter Christen and Agus Pudjijono. 2009. Accurate Synthetic Generation of Realistic Personal Information. Springer-Verlag.

Peter Christen and Dinusha Vatsalan. 2013. Flexible and Extensible Generation and Corruption of Personal Data. In ACM International Conference on Information and Knowledge Management 1165–1168. New York, USA: ACM Press.

Digitising Scotland. 2013. Digitising Scotland. *http://digitisingscotland.host.cs.st-andrews.ac.uk/.*

GEDCOM. 2013. The GEDCOM Standard: Release 5.5. *https://devnet.familysearch.org/docs/gedcom/gedcom55.pdf.*

Graphviz. 2013. The DOT Language. *http://www.graphviz.org/doc/info/lang.html.*

Hibernate. 2013. Hibernate - JBoss Community. *http://www.hibernate.org/.*

Ekaterini Ioannou, Nataliya Rassadko, and Yannis Velegrakis. 2013. On Generating Benchmark Data for Entity Matching. Journal on Data Semantics, 2(1), 37–56: 37–56.

MariaDB. 2013. Welcome to MariaDB! *https://mariadb.org/.*

Neo4j. 2013. Neo4j - the World's Leading Graph Database. *http://www.neo4j.org/.*

Lucía Pasarin Perea. 2013. Analysis of Alternatives to Store Genealogical Trees Using Graph Databases. Masters Thesis, Universita Politecnica de Catalunya.

John R Talburt, Yinie Zhou, and Savitha Yalanadu Shivaiah. 2009. SOG: a Synthetic Occupancy Generator to Support Entity Resolution Instruction and Research. In 14th International Conference on Information Quality.