Comparing Relational and Graph Databases for Pedigree Datasets

Graham Kirby, Conrad de Kerckhove, Ilia Shumailov,
Jamie Carson, Alan Dearle
University of St Andrews

Chris Dibben, Lee Williamson
University of Edinburgh

Background

- **Digitising Scotland** project (ESRC ES/K00574X/1)
  - Transcribe vital events records 1855-1973
  - Code *death cause*, *occupation*, *place* to standard classifications
  - Link records to form pedigree

Digitising Scotland Aims

- Produce resource for researchers in geography, health, history etc
- Preserve data, metadata, provenance of decisions
- Model, query, visualise uncertainties
- Tailor coding and linking algorithms to historical periods
- Automate as much as possible
  - processing, documentation, planning of human input
- Experiment with
  - storage and processing technologies
  - adaptive linkage algorithms

Scale of Source Data

	Records	Estimated Size
Births	14 million	1.9 GB
Deaths	11 million	2.3 GB
Marriages	4.2 million	0.8 GB

Overview

- Compared relational and graph database for storing and traversing linked structure
  - query performance
  - ease of expression
- MariaDB
  - open source fork of MySQL
  - used with Hibernate
- Neo4j
  - Java-based graph database

Synthetic Data

- Generate synthetic family tree with parameters:
  - population size
  - date range
  - birth rate over time
  - age at marriage, length and number of marriages
  - age at child birth, inter-child gap, number of children
  - age at death
  - immigration rate
- Memory bottleneck with current approach
- Export to database, GEDCOM or graph file

b: 12/10/1805
d: 10/07/1870

b: 13/06/1830
d: 24/01/1910

b: 26/11/1849
d: 12/11/1934

b: 22/10/1895
d: 17/01/1977

b: 12/02/1913
d: 06/02/1967

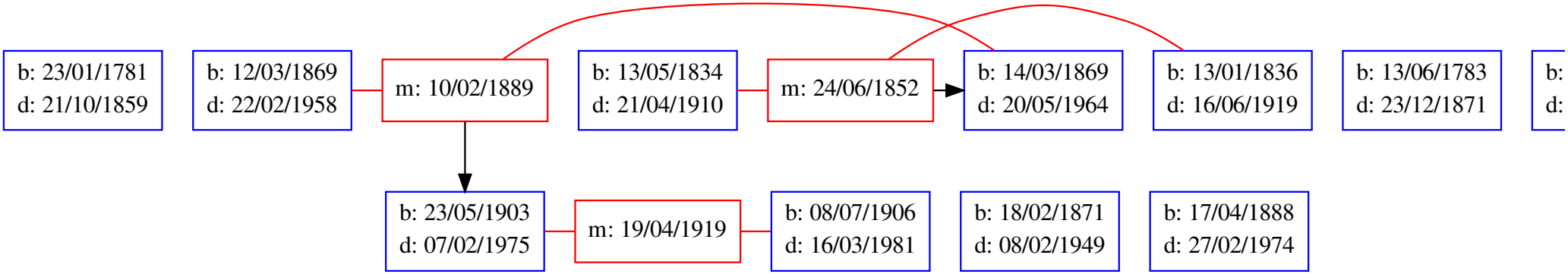
b: 13/04/1930
d: 07/04/1996

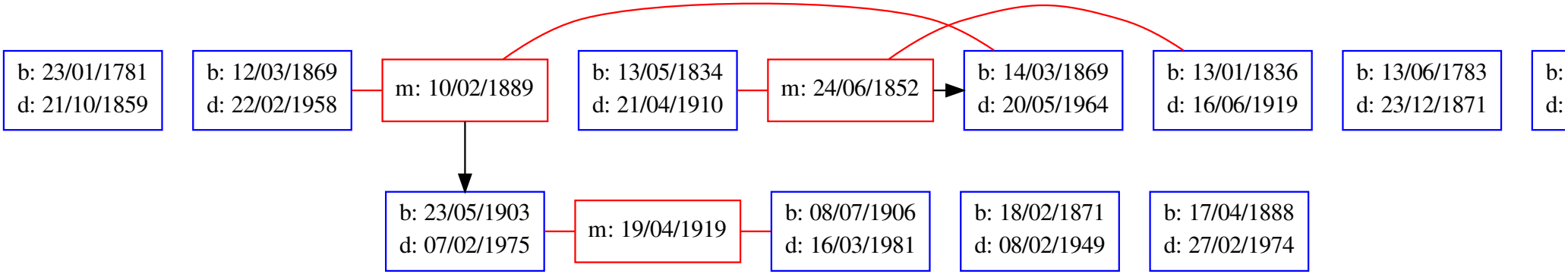
b: 27/12/1937
d: 04/01/2003

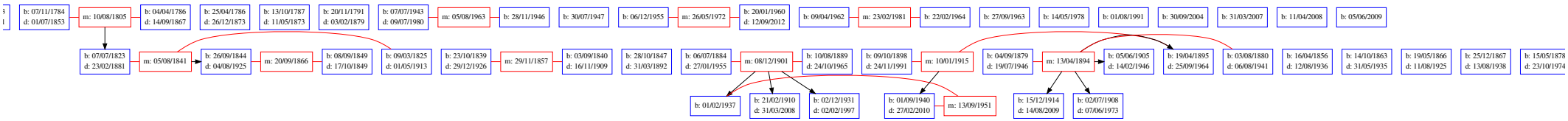
b: 26/11/1952

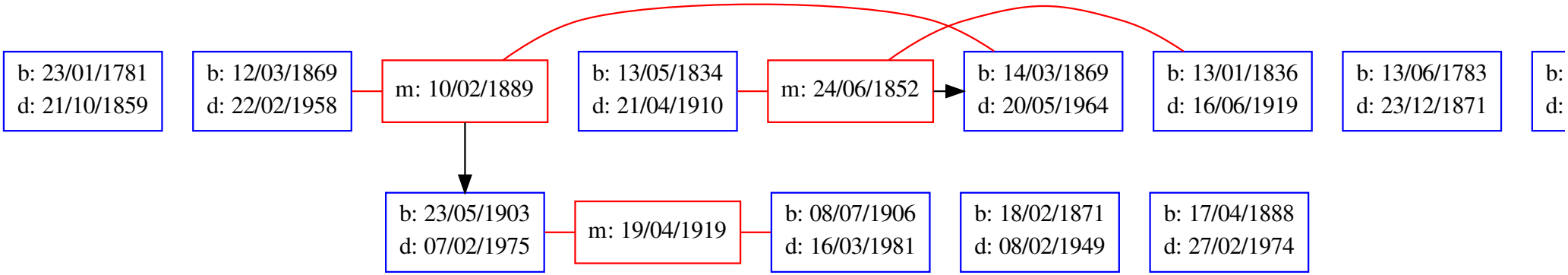
b: 06/08/1955

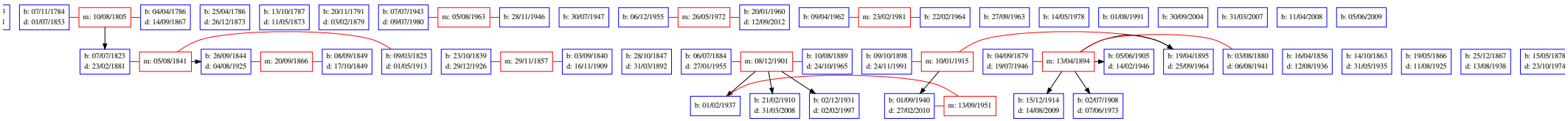
b: 06/02/1978

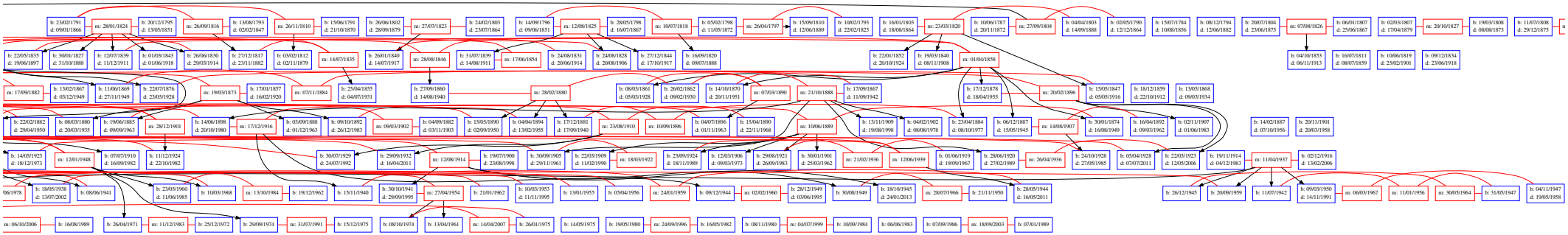




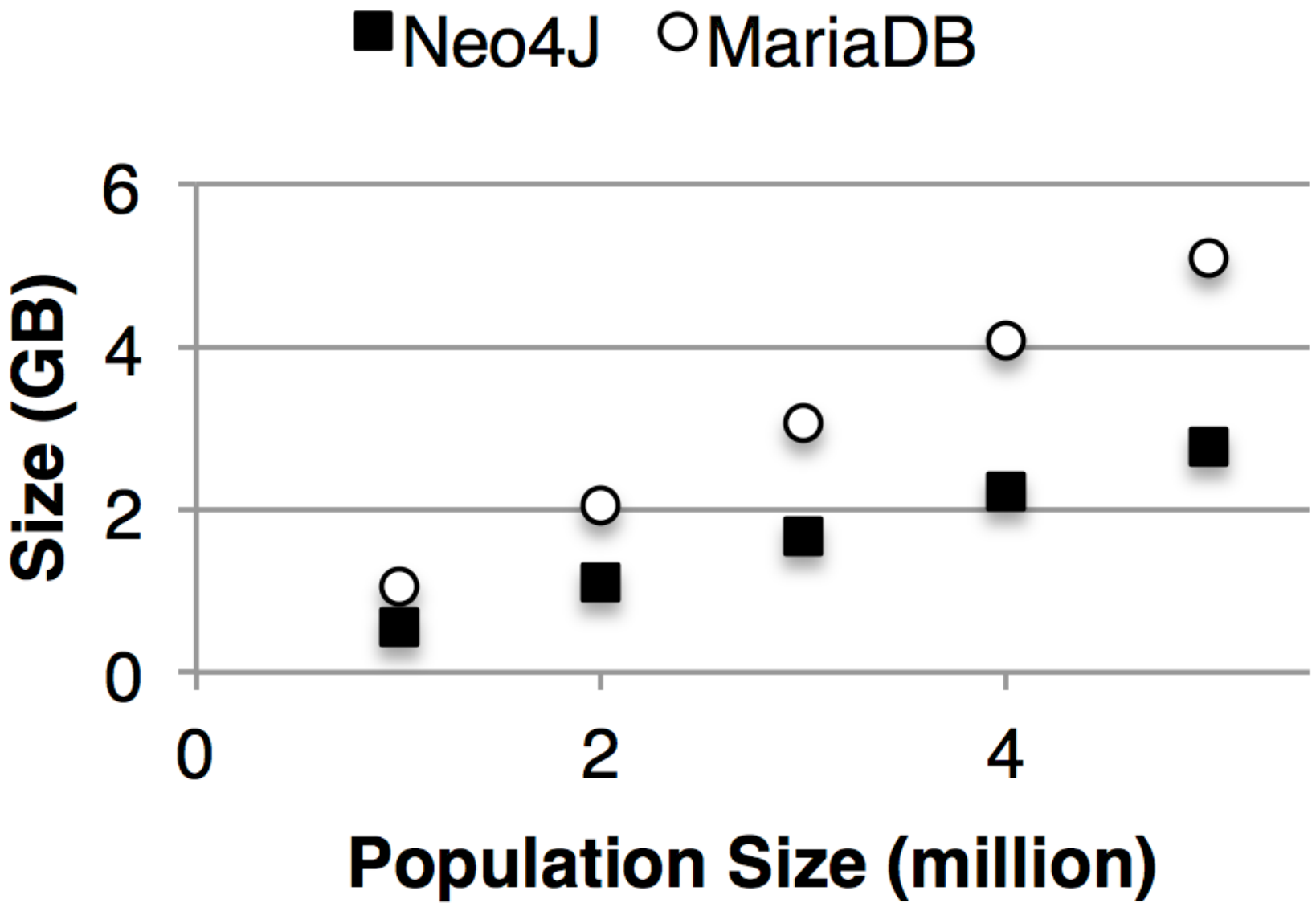








Database size on disk



Queries

- Find the children of a given person
- Find all ancestors of a given person (bounded)
  - going back given number of generations
- Find all descendants of a given person (bounded)
  - going forward given number of generations
- Find all relatives of a given person (bounded)
  - to given distance in graph: ancestors, descendants, siblings, cousins etc

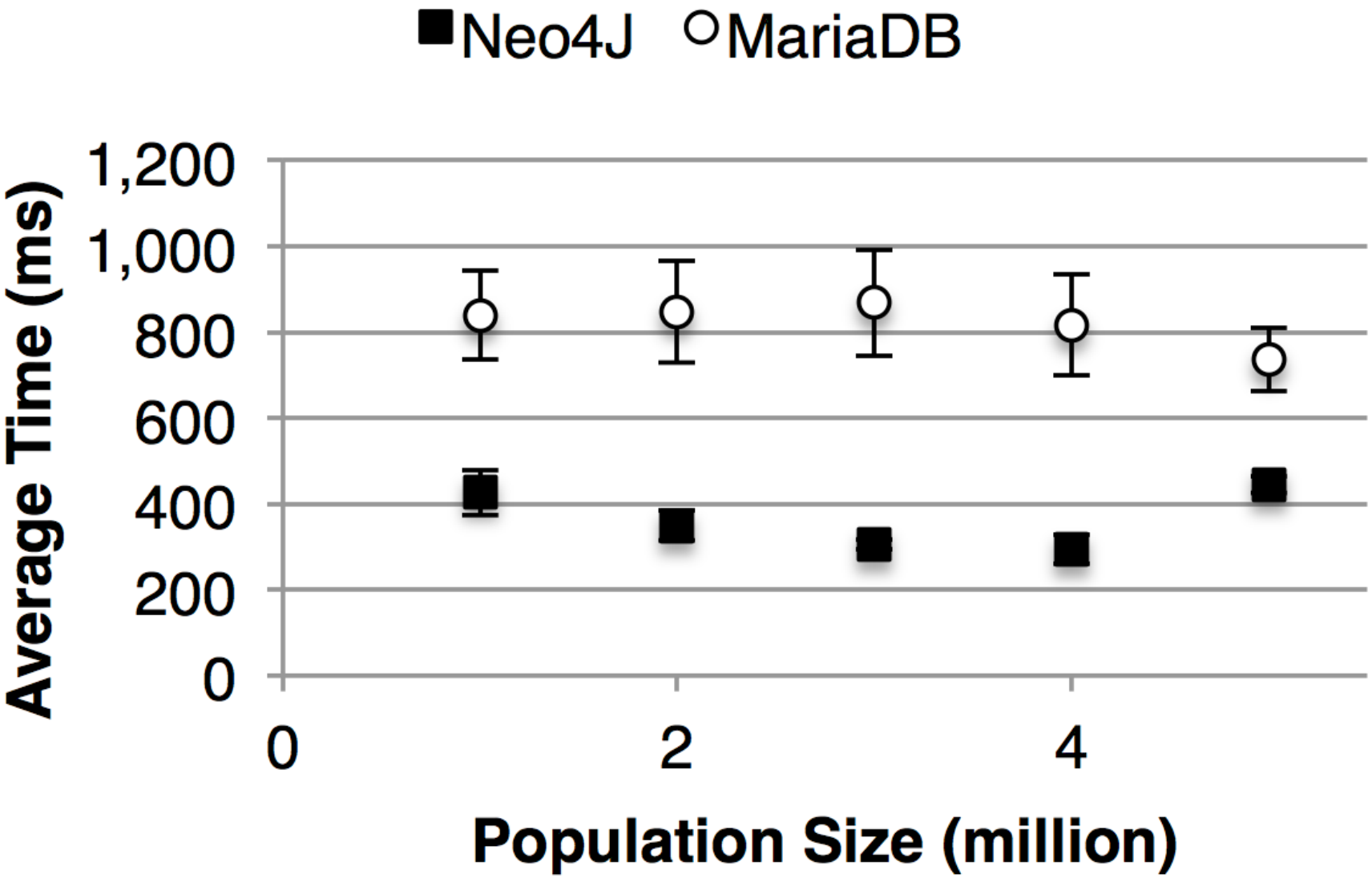
Expressing Relatives Query

Expressing Relatives Query

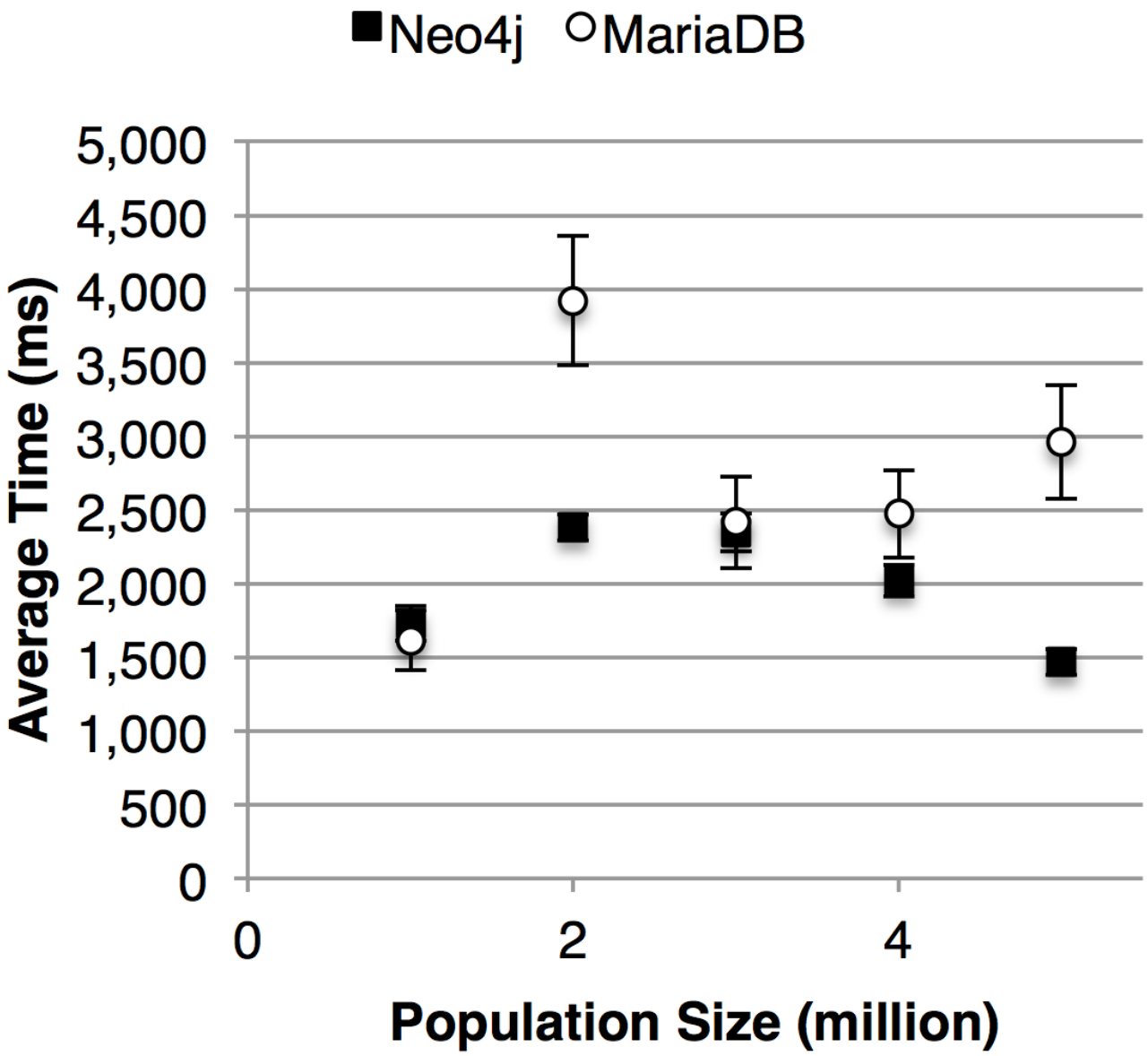
- Neo4j
  - ```
    start person=node(ID)
    match person-[:PARTNER|CHILD*1..10]-relatives
    where relatives.__type__! = 'Neo4jPerson'
    return distinct relatives
    ```
    ```

Expressing Relatives Query

- Neo4j
  - ```
    start person=node(ID)
    match person-[:PARTNER|CHILD*1..10]-relatives
    where relatives.__type__! = 'Neo4jPerson'
    return distinct relatives
    ```
- MariaDB/Hibernate
  - need a new slide...
    ```

```
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 = PopulationUtilsID.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 = PopulationUtilsCommons.getPersonById(id);
               finalSet.add(currPerson);
        }
        final Set<Person> siblings = finalSet;
        final Set<Long> parentsIDs = PopulationUtilsID.getParentsIDs(p.getId());
        final Set<Person> parents = new HashSet<Person>();

        for (final Long id1 : parentsIDs) {
               final Person currPerson = PopulationUtilsCommons.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;
```

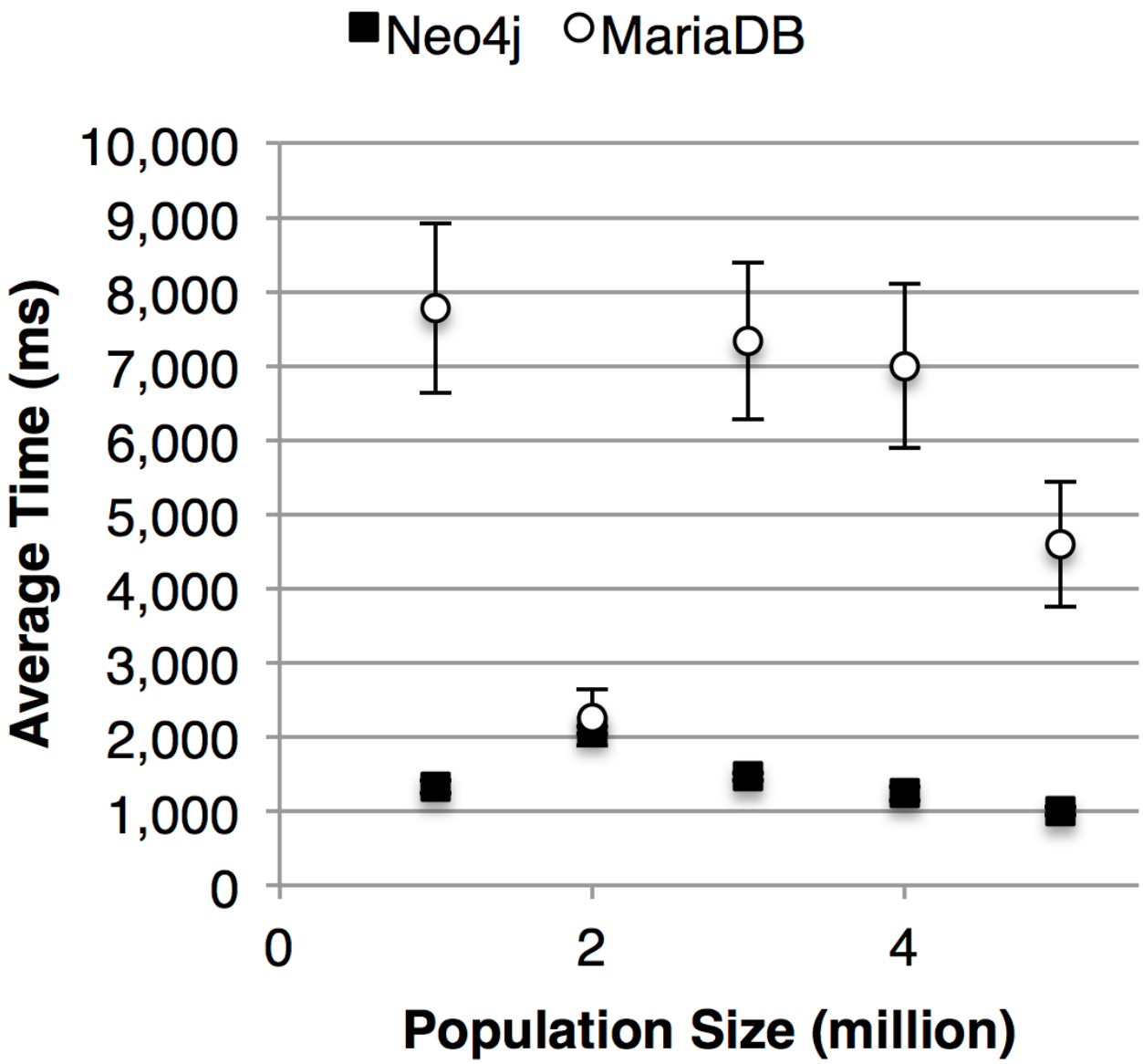
Time to retrieve 20 people and their children, 100 times, 20 warm trials



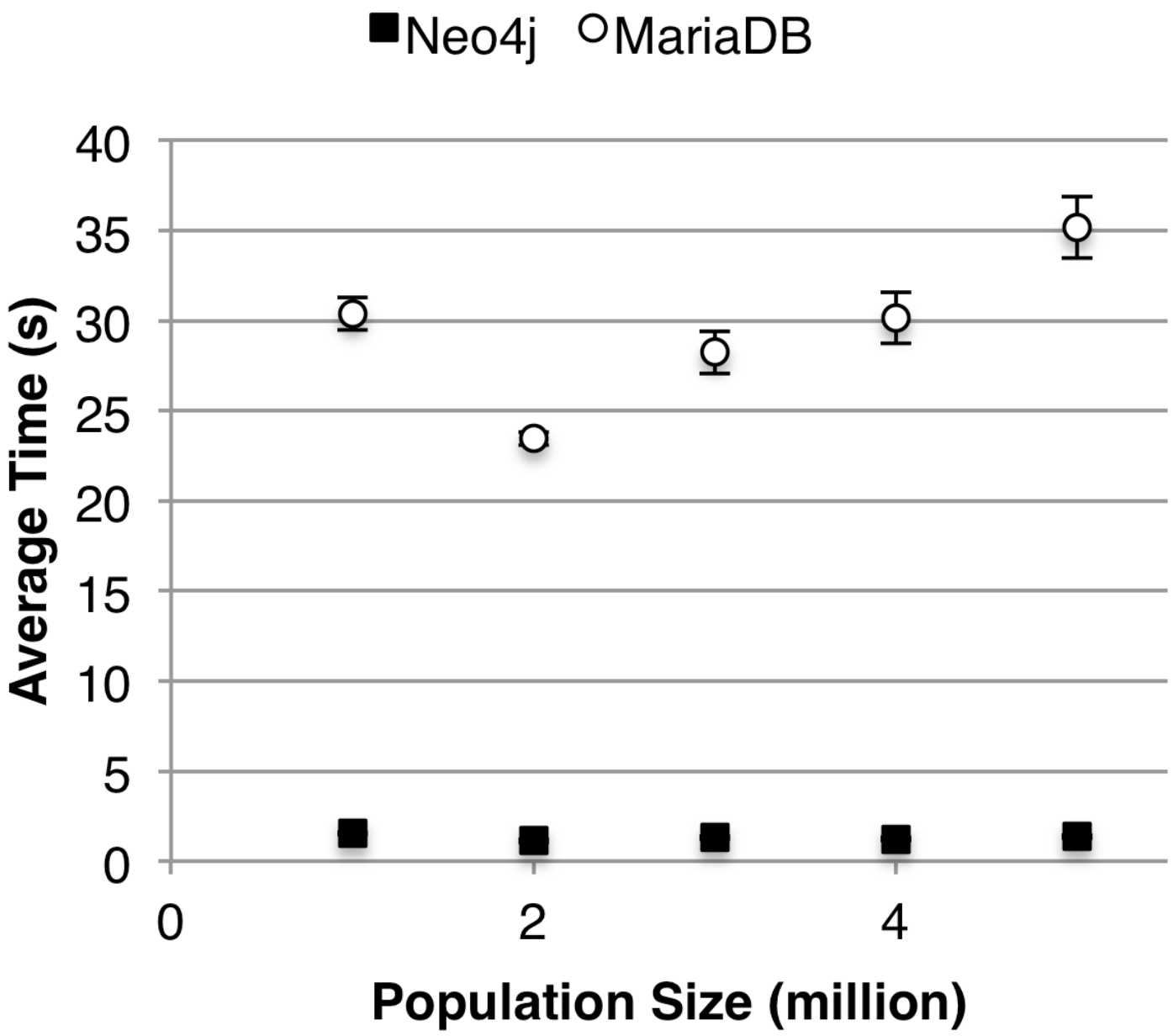
Time to retrieve ancestors to height 10 for 20 people, 100 times, 20 warm trials



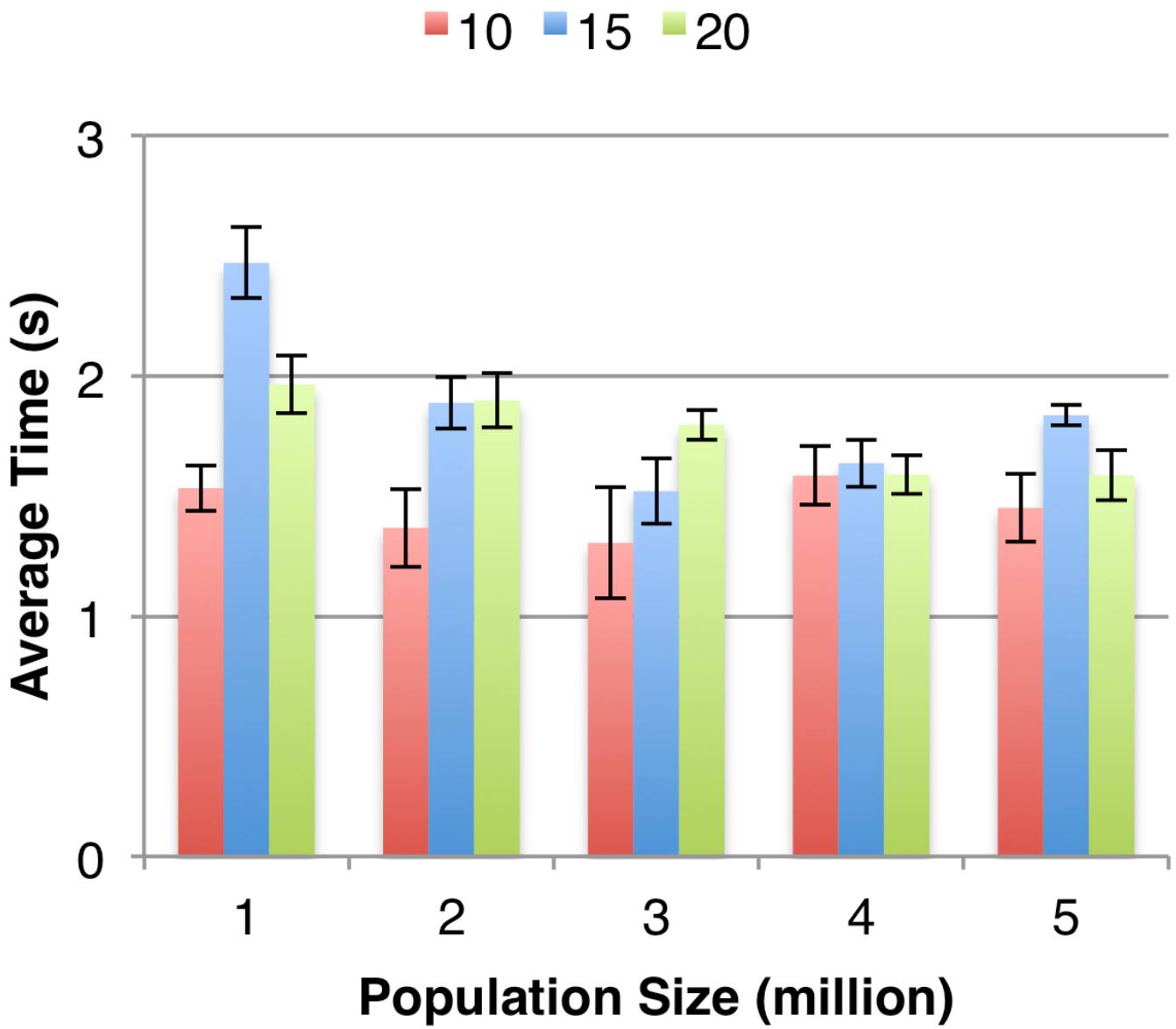
Time to retrieve descendants to depth 10 for 20 people, 100 times, 20 warm trials



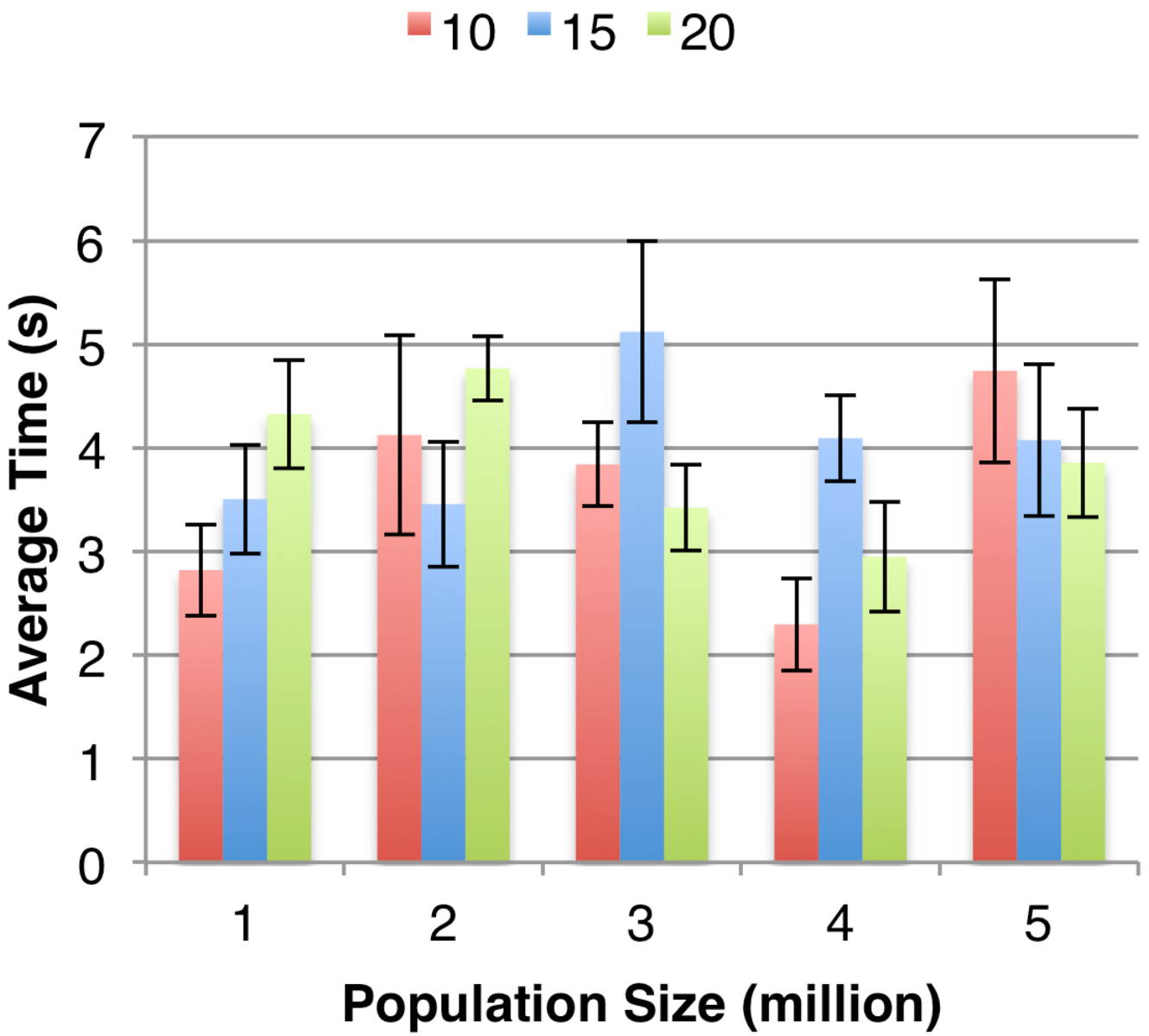
Time to retrieve relatives to distance 5 for 10 people, 20 warm trials



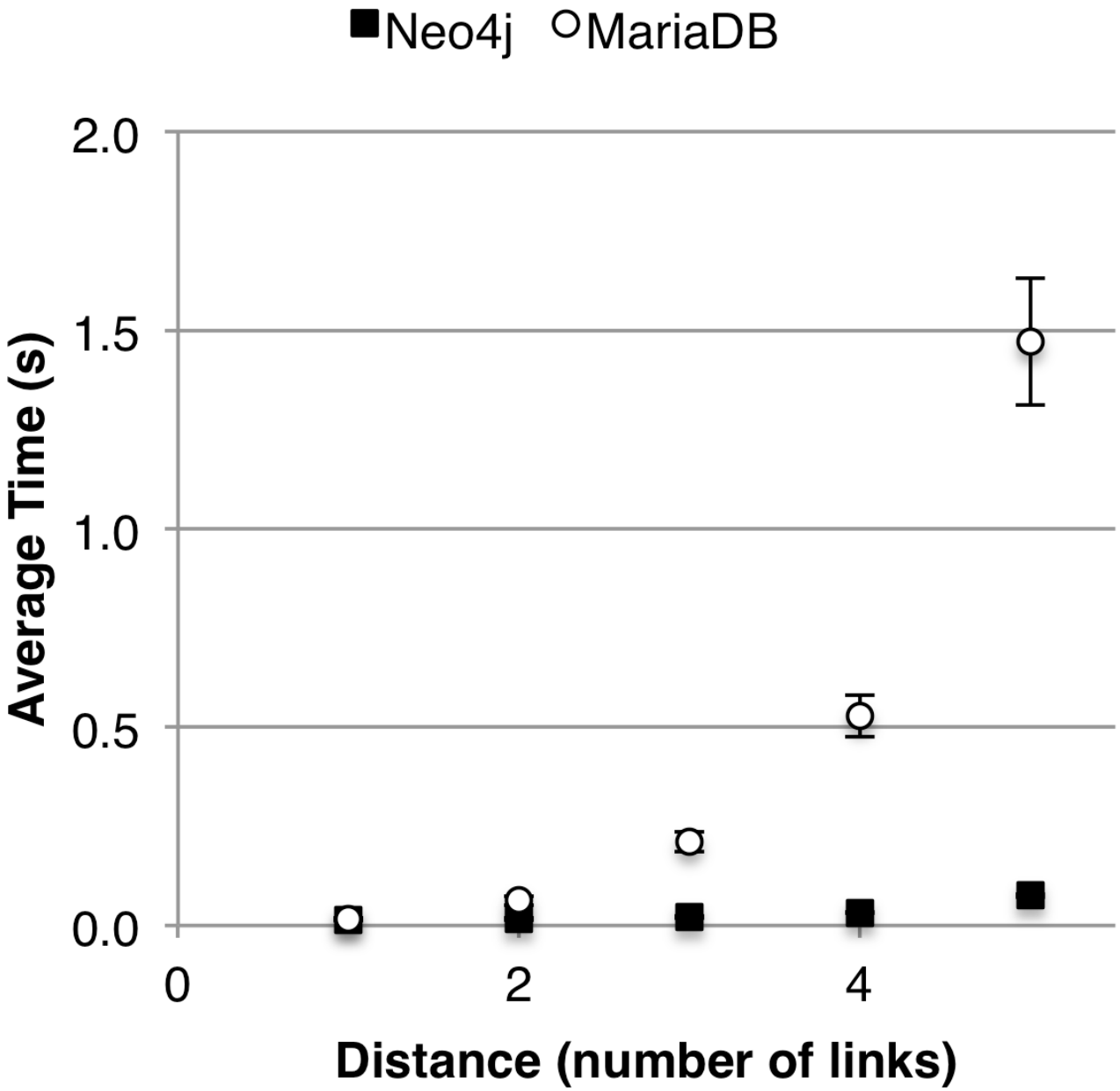
Time to retrieve ancestors to various heights for Neo4j, 100 times, 10 warm trials



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



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



Further Work

- More queries
  - nearest common ancestor
  - coefficient of relatedness
  - mean coefficient of relatedness
  - taking account of uncertainties
- More databases
  - stored procedures
  - other database implementations
  - other types: object, key-value, RDF triple

Further Work

- Enhance synthetic population generation as useful tool in developing linkage process
  - synthetic vital event records
  - realistic names, occupations, locations
  - incremental generation to mitigate memory constraint

Conclusions

- Need to consider both linkage process and subsequent queries
- Neo4j out-performed MariaDB in these tests
  - execution time
  - storage efficiency
  - easier to express most queries
- Planning to investigate RDF triple stores

```
CREATE PROCEDURE getRelatives
(IN init_person_id BIGINT(20), IN distance INT)
BEGIN

	SET @initper = init_person_id;
	SET @currdistance = distance;

	DROP TABLE IF EXISTS result_table;
	DROP TABLE IF EXISTS curriter_persons_to_check;
	DROP TABLE IF EXISTS persons_to_check;

	CREATE TABLE IF NOT EXISTS result_table  (
					person_id BIGINT(20)
					);

	CREATE TABLE IF NOT EXISTS persons_to_check (
					person_id BIGINT(20)
					);

	INSERT INTO persons_to_check SELECT child_id FROM PARTNERSHIP_CHILDREN WHERE child_id!=@initper AND partnership_id=(SELECT partnership_id FROM PARTNERSHIP_CHILDREN WHERE child_id=@initper LIMIT 1);
	INSERT INTO persons_to_check SELECT person_id FROM PERSON_PARTNERSHIP WHERE partnership_id=(SELECT partnership_id FROM PARTNERSHIP_CHILDREN WHERE child_id=@initper LIMIT 1);
	INSERT INTO persons_to_check SELECT child_id FROM PARTNERSHIP_CHILDREN WHERE partnership_id IN(SELECT partnership_id FROM PERSON_PARTNERSHIP WHERE person_id=@initper);
	INSERT INTO persons_to_check SELECT t1.person_id FROM PERSON_PARTNERSHIP t1 INNER JOIN PERSON_PARTNERSHIP t2 ON t1.partnership_id=t2.partnership_id AND t2.person_id = @initper AND t1.person_id<>@initper;

	whileloop: WHILE (@currdistance > 0)
	DO
		BEGIN
			DECLARE done INT DEFAULT FALSE;
			DECLARE i BIGINT(20);

			DECLARE curs1 CURSOR FOR SELECT person_id FROM persons_to_check;
			DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

			IF NOT EXISTS (SELECT 1 FROM persons_to_check LIMIT 1 )
			THEN
				LEAVE whileloop;
			END IF;

			INSERT INTO result_table SELECT * FROM persons_to_check;

			CREATE TABLE curriter_persons_to_check (
						person_id BIGINT(20)
						);

			OPEN curs1;

			read_loop: LOOP
				FETCH curs1 INTO i;
				IF done THEN
					LEAVE read_loop;
				END IF;

				INSERT INTO curriter_persons_to_check SELECT child_id FROM PARTNERSHIP_CHILDREN WHERE child_id!=i AND partnership_id=(SELECT partnership_id FROM PARTNERSHIP_CHILDREN WHERE
child_id=i LIMIT 1);
				INSERT INTO curriter_persons_to_check SELECT person_id FROM PERSON_PARTNERSHIP WHERE partnership_id=(SELECT partnership_id FROM PARTNERSHIP_CHILDREN WHERE child_id=i LIMIT
1);
				INSERT INTO curriter_persons_to_check SELECT child_id FROM PARTNERSHIP_CHILDREN WHERE partnership_id IN(SELECT partnership_id FROM PERSON_PARTNERSHIP WHERE person_id=i);
				INSERT INTO curriter_persons_to_check SELECT t1.person_id FROM PERSON_PARTNERSHIP t1 INNER JOIN PERSON_PARTNERSHIP t2 ON t1.partnership_id=t2.partnership_id AND
t2.person_id = i AND t1.person_id<>i;

			END LOOP;
			CLOSE curs1;

			TRUNCATE persons_to_check;
			INSERT INTO persons_to_check SELECT * FROM curriter_persons_to_check;

			DROP TABLE curriter_persons_to_check;

			SET @currdistance = @currdistance - 1;
		END;
	END WHILE;

	IF EXISTS(SELECT 1 FROM result_table WHERE person_id=@initper LIMIT 1)
	THEN
		DELETE FROM result_table WHERE person_id=@initper;
	END IF;


	SELECT DISTINCT * FROM result_table ORDER BY person_id;
	DROP TABLE result_table;
	DROP TABLE persons_to_check;

END//
```