Performance and Scalability of Datastore Technologies for Software Analysis Models

Kanishqk Singh and Robert J. Walker
Laboratory for Software Modification Research, Department of Computer Science, University of Calgary
Calgary, Canada
e-mail: kanishqk.singh@lsmr.org, walker@lsmr.org

***Abstract*—Software Development and Analysis Tools (SDATs) typically contain complex models (software analysis models) that are expensive to compute, and whose expense grows rapidly as the size of the software system under analysis increases. When these models are not stored in a manner that allows them to be restored after program restart, that expense is not amortized; re-computation results in undesirable downtime in the developer's daily workflow. We investigate options for storing and restoring software analysis models relative to a realistic set of use cases for SDATs. Existing work to study and identify optimal storage technology has been evaluated using datasets either that consist of random graphs—not simulating the nature of real world software—or that derive from excessively small software systems for which recomputing would be feasible. We perform an experimental study on the performance and scalability of datastore technologies exemplifying different approaches (flat files, relational databases, graph databases). We find that SDATs that are heavily focused on storing/retrieving models would find PostgreSQL (a relational database approach) to be the better fit. SDATs that are inclined towards analyzing a limited quantity of software at a given time but involving high maintenance of the models in the database would find Neo4j (a graph database approach) to be the most suitable option.**

Keywords—Software analysis models; persistence; datastore; performance; scalability.

I. Introduction

Real-world software systems tend to be large; they are developed over time with changing business and technical environments by changing groups of people [1]. While in principle a developer can make all needed changes with nothing more sophisticated than a text editor, this would place an excessive burden on them [2][3]. Instead, developers make use of *semi-automated software development and analysis tools* (SDATs) to analyze potential changes, to make changes, and to catch errors [4][5].

Most SDATs build atop one or more analysis-oriented models of the software (*Software Analysis Models*, SAMs). The space and time costs of building SAMs depend upon the depth of analysis, the nature of the analysis approach applied, and the software being analyzed. The larger and more complex the software, the more costly it is to build the SAM [6], and even optimized versions of SDATs can require long building times and large amounts of memory.

Often, the same piece of software needs to be analyzed anew by the same or different developers because its SAMs are no longer present in memory. While rebuilding may be straightforward and a minor inconvenience for small systems, that is not the case for large and critical ones. Traditional rebuild mechanisms simply recompute a SAM in its entirety, which is as costly as building it from scratch. The rebuild cost can be amortized if there is some means of storing all or part of the SAMs out of volatile core memory and restoring them to core memory when needed again, assuming that the cost of rebuilding would be greater than the cost of storing and retrieving the models. The validity of this assumption and the degree of the savings involved will depend on (a) the details of the analysis model, (b) the size of the analysis model, and (c) the mechanism of storing and retrieving the models to/from external memory.

Prior work on such storage mechanisms provides us little evidence to leverage in deciding the best approach to take. Typical datasets used to evaluate and compare relational databases and graph databases are random graphs which do not accurately represent the nature of real world software. The comparison of storage mechanism libraries has been performed on small software systems, where rebuilding SAMs for them is already feasible; the research results must scale up to industrial applications for them to be useful [7].

We investigate the performance and scalability of different technologies of potential use for storing and restoring software analysis models. Our initial investigations (not described for lack of space) into technology options pointed us to: (a) *flat files*, including simple text files, Comma-Separated Value (CSV) files [8], and JavaScript Object Notation (JSON) files [9]; (b) *relational database management systems*, such as MySQL, IBM DB2, and PostgreSQL all of which use Structured Query Language (SQL) for interactions; and (c) *non-relational databases* (NoSQL [10]) including *graph databases* which store data as graphs, such as Neo4j [11]; and (d) *cloud storage* techniques permitting remote storage of data, such as the Google Cloud Datastore: a cloud-based NoSQL database which allows the user to interact with the data on the cloud by SQL-like queries [12].

To reduce the scope of this issue to a more manageable level, we consider the problem of *change propagation*, a technique used to re-establish consistency to a system after a change has been made within the source code [13], as supported by the tool ModCP [14]. To investigate the growth behaviour as well as scalability of dependence graphs used for change propagation, we generate Barabási–Albert graphs, which are random, scale-free, and follow the preferential attachment model [15]. Dependency graphs, which lie at the core of the change propagation model, are a representation of data- or control-flow between the entities of a software program. We identify a set of use cases derived from the functionality of ModCP, a tool for change propagation, requiring the storage and/or manipulation of these simulated graphs, and perform them on a CSV approach implemented in Python ("Python-CSV"), two relational databases (MySQL and PostgreSQL), and a graph database (Neo4j).

The remainder of the paper is organized as follows. In Section II, we describe the design of our experimental study. In Section III, we provide and analyze the results. In Section IV, we discuss the remaining issues.

II. Experimental Study

We compare the database technologies we have identified as pertinent for their potential to reduce re-computation costs in SDATs. The purpose of this study is to address the following research question:

RQ1: *How do different database technologies perform on realistic operations over realistic software analysis models?*

We describe the measures of performance that we utilize in Section II-A. Section II-B describes the graphs that we generate as the experimental data for this study. The study setup is explained in Section II-C.

A. Performance measures

We utilize objective measures to evaluate in this study, including the complexity of the model graphs (number of nodes and of edges), the time taken to process a query, and the space required to store the model data on the candidate database technology. Subjective measures are also pertinent, but we do not include them in this paper for lack of space.

B. Experimental dataset

1) Mathematical preliminaries: A *graph* G is a pair (V, E), where V is a set of *vertices* and E is a set of *edges*, such that $E \subset V \times V$ and $\forall e \in E, e = (v_i, v_j) \Rightarrow v_i \neq v_j$; we concern ourselves with only undirected graphs in this study, without self-loops, and where any pair of nodes possess at most one edge between them. For convenience, we define $n = |V|$ and $m = |E|$ relative to the graph in context.

An edge $e = (v_i, v_j)$ is said to be *adjacent* to v_i and to v_j; the set of edges adjacent to vertex v is the set of all edges $e \in E, e = (v_i, v_j)$ such that $v = v_i$ or $v = v_j$; the *degree* of vertex v — represented as $\deg(v)$ — is the cardinality of the set of edges adjacent to it. The minimum degree of a graph G — represented as $\deg_{\text{min}}(G)$ — is the smallest degree of all vertices in V; the maximum degree of a graph G — represented as $\deg_{\text{max}}(G)$ — is the largest degree of all vertices in V. The average degree of a graph G is given by $\overline{\deg}(G) = \frac{1}{n} \sum_{i=1}^{n} \deg(v_i)$. The maximum number of edges in a graph G is given by $m_{\text{max}}(G) = \frac{1}{2} n(n-1)$. The density of a graph G is given by $d(G) = m/m_{\text{max}}$.

2) Generated graphs: We generated a total of nine graphs G_1–G_9, as detailed in Table I. These are scale-free graphs, generated via the Barabási–Albert model [16] following the linear preferential attachment rule — also known as "the rich become richer"; the implementation used to generate them (in Python) utilizes the NetworX library that supports such generation [17]. This model represents a random dynamic graph grown from a small "seed" graph by an indefinitely repeated addition of a new vertex with m edges. The free ends of the edges of each vertex are preferentially connected to vertices that are already rich in connections. The probability p_i of connecting an edge with the vertex v_i is proportional to the local degree of connectivity k_i of v_i [18][19]: $p_i = k_i / (\sum_{j=1}^{n} k_j)$.

As per Diestel [20], *sparse* graphs are those whose number of edges is about linear in their vertices. Similarly, *dense* graphs are those in which the number of edges is close to the maximal number of edges [21]. As shown in Table I, the column groups *2% Density*, *10% Density*, and *25% Density* represent the three categories we used based on the density of the graph. The columns G_1 through G_9 refer to the individually generated graphs, where G_1 to G_3 fall under 2% density, G_4 to G_6 fall under 10% density, and G_7 to G_9 fall under 25% density. The rows n and m provide the number of nodes and edges in the graphs, respectively; $m_{\text{max}}(G)$ is the maximum number of edges that could be present in this graph; $d(G)$ provides the actual density of the graph to three decimal points (this can vary from the exact target density of the category because of randomness in the generation algorithm); $\deg_{\text{min}}(G)$ is the minimum node degree in the graph; $\deg_{\text{max}}(G)$ is the maximum node degree in the graph; and $\overline{\deg}(G)$ provides the average of all the nodes' degree in the graph. The rows *nodes.csv* and *edges.csv* specify the size of the files recording the graphs, in bytes. One common characteristic among the nine graphs is that none of them is a complete graph, which closely represents real-world scale-free graphs exhibiting a "long tail" [22].

We store graphs as adjacency lists because of better space complexity, better growth characteristics with sparse graphs, faster access, and size limitations in various database technologies, as compared to adjacency matrices.

C. Setup for evaluation

In this section, we explain in detail the setup for our evaluation.

1) Graph creation: In our experiment, we simulate the creation of the models for an SDAT by preparing a scale-free graph using the Barabási–Albert model. The creation of the benchmark model datasets and the underlying graphs is done by using the NetworkX library in Python [23]. Once all the graphs were generated they were then stored in two files: nodes.csv and edges.csv (not shown).

2) Database technologies used in the experiment: We support semi-structured databases by using the NetworkX library in Python 3.6.2 and Comma-Separated Value (CSV) files. For relational database technology, we use MySQL Workbench (Version 8.0.22, Community Edition) and pgAdmin4 (v5.3) as a development platform for PostgreSQL which is a well known advanced object relational database technology. For non-relational database technology, we use Neo4j (v4.2) which is a graph database platform. The rest of this section describes the setup of the database technologies used.

TABLE I. UTILIZED SCALE-FREE GRAPHS GENERATED VIA THE BARABÁSI–ALBERT MODEL.

	2% Density			10% Density			25% Density		
	G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8	G_9
n	100	1,000	10,000	100	1,000	10,000	100	1,000	10,000
m	99	9,900	999,799	475	49,296	4,992,271	1,204	124,684	12,496,704
$m_{\max}(G)$	4,950	499,500	49,995,000	4,950	499,500	49,995,000	4,950	499,500	49,995,000
$d(G)$	0.020	0.020	0.020	0.096	0.099	0.099	0.243	0.250	0.250
$\deg_{\min}(G)$	1	10	101	3	52	527	14	146	1,464
$\deg_{\max}(G)$	14	156	1,651	42	391	3,706	64	621	6,120
$\overline{\deg}(G)$	1.98	19.80	199.96	9.50	98.59	998.45	24.08	249.36	2,499.34
nodes.csv	291	3,891	48,891	291	3,891	48,891	291	3,891	48,891
edges.csv	634	82,465	10,330,489	3,119	420,815	52,603,962	8,209	1,096,976	134,950,863

a) Python-CSV: We used the NetworkX library to create artificial datasets and the csv library to maintain the datasets with the changes implemented. We iterated a set of statements to implement each use case and restored the dataset again back to the initial state within the loop once the use case was complete. The execution time was captured for only those statements which were responsible to implement the use case.

b) MySQL Workbench: To import our database, we used the load data infile method which reads from text files at a very fast speed. For the dataset containing nodes, we set the node id as the primary key to avoid duplicate entries of the same node again. The datasets were stored locally on the system and "-secure-file-private" option was disabled during the experiment to provide access to documents from the whole file system. MySQL workbench uses SQL language to define and manipulate data.

c) Neo4j: For small datasets, the direct import function available on the user interface of Neo4j can be used to import data into the database. For medium sized datasets, up to import to the database. For huge datasets, batch import command is used. Since our datasets fall in the range of medium-sized datasets, we used the load csv method to import the artificial datasets into the system. We chose to commit periodically after every 10,000 entries in order to avoid memory overflow. For optimization, we increased the page cache size and kept the maximum memory heap size to fifty percent of the total RAM minus the heap size. We constructed the dependency graph in a similar manner as the social media friendship graph, since both graphs exhibit long tail behaviour. We utilized the node name as an index in the database to aid fast search. We also opted for "merge" rather than "create" to avoid redundancy in the node list. Neo4j uses the Cypher query language to define and manipulate graph and data.

d) pgAdmin 4: We set the node id as the primary key and the connection's source and target ids as foreign keys. We also made use of the update cascade and delete cascade methods in the edges table in order to automate a few processes, such as renaming a node in the node; subsequently, the database would alter the entries in the edges database without manual intervention. This has also helped us to prevent adding new connections for the nodes that did not exist. To import the dataset to the database, we used the copy method.

```
create table nodes(
  id INTEGER PRIMARY KEY
);

CREATE TABLE edges(
  a INTEGER NOT NULL references nodes(id)
    ON update cascade on delete cascade,
  b INTEGER NOT NULL references nodes(id)
    ON update cascade on delete cascade,
  PRIMARY KEY(a,b)
);

create index a_idx on edges(a);
create index b_idx on edges(b);
```

Figure 1. Defining schemas and creating tables in PostgreSQL and MySQL.

```
CREATE TABLE nodes (
  id INTEGER PRIMARY KEY,
  name VARCHAR(10) NOT NULL
);

CREATE TABLE edges (
  a INTEGER NOT NULL REFERENCES nodes(id)
    ON UPDATE CASCADE ON DELETE CASCADE,
  b INTEGER NOT NULL REFERENCES nodes(id)
    ON UPDATE CASCADE ON DELETE CASCADE,
  PRIMARY KEY (a, b)
);

CREATE INDEX a_idx ON edges (a);
CREATE INDEX b_idx ON edges (b);
```

Figure 2. Creating tables to store the graphs in PostgreSQL.

3) Defining schemas for MySQL and PostgreSQL: Figure 1 shows the setup required for the relational database technologies before we store the data in it. The command create table was used to create the table nodes, where id is the primary key. Similarly, we created the to and from columns for the table edges, referencing the primary key of the nodes table. Lastly, we used create index to add indices on the to and from columns of table edges.

4) Use cases realized via datastore technologies: We describe the use cases on which we evaluate the technologies.

UC1: *Create or store a graph.* We used SQL to store the

```
COPY edges (a, b)
FROM '/Applications/friendship.csv' DELIMITER ','
   CSV header;
```

Figure 3. UC1 via PostgreSQL.

```
load data infile "/Applications/edges_100.csv"
into table edges fields terminated by ',' lines
terminated by '\n' IGNORE 1 LINES;
```

Figure 4. UC1 via MySQL.

graphs in MySQL and PostgreSQL, and we used Cypher to store them in Neo4j (they were already in the appropriate format for Python-CSV, so no explicit store operation was needed). We implemented all the use cases using Python to represent semi structured database creation and manipulation. For relational database technology, we first defined the schema, e.g., nodes and edges. Figure 2 contains the SQL query to define the schema of the database in PostgreSQL. As shown in Figure 3, once the relations were defined, we loaded our graphs into them, via COPY in PostgreSQL. For MySQL, Figure 4 shows how we used load data infile to store the graphs. Similarly for Neo4j, Figure 5 shows the loading of graphs using CREATE to create the nodes. Unlike with relational databases, in graph databases like Neo4j, the schema of a graph (or relation) does not need to be explicitly defined prior to storing the data. The relationships (edges) are created along the way while loading data from .csv files.

UC2: *Read/access a graph*. Once the database was created, we retrieved the entries to simulate the process of a data request from an SDAT to perform a user-requested analysis. For Python-CSV, we loaded the dataset from a CSV file to a data structure in memory as shown in Figure 6. For MySQL and Neo4j, all the rows of the nodes and edges tables were loaded into memory. Figure 7 shows how the nodes and relationships were typically loaded in Neo4j by only using the Match clause (the data loaded in the memory was not returned/displayed on the Neo4j browser, i.e., the "return" clause was not used).

UC3–UC6: Use cases UC3–UC6 simulate the modification of the model graph: adding a new node (UC3), as shown in Figure 8 for MySQL/PostgreSQL and in Figure 9 for Neo4j; adding a new edge (UC4), as shown in Figure 10 for MySQL/ PostgreSQL and in Figure 11 for Neo4j; renaming a node (UC5), as shown in Figure 12 for MySQL/PostgreSQL and in Figure 13 for Neo4j; and modifying an edge (UC6), as shown in Figure 14 (MySQL/PostgreSQL) and in Figure 15 (Neo4j).

```
LOAD CSV WITH HEADERS FROM 'file:///people.csv' AS line
        WITH line
CREATE (:Person {id:line.id, name:line.name})
CREATE INDEX ON :Person(name);
USING PERIODIC COMMIT
```

Figure 5. UC1 via Neo4j.

```
with open('nodes.csv', 'r', newline='') as csvfile:
   node_data = csv.reader(csvfile, delimiter='\n', quotechar='|',
         quoting=csv.QUOTE_MINIMAL)
   nodes=list(node_data)
with open('edges.csv', 'r', newline='') as csvfile:
   edge_data = csv.reader(csvfile, delimiter=',', quotechar='|',
         quoting=csv.QUOTE_MINIMAL)
   edges=edge_data
```

Figure 6. UC2 via Python-CSV.

```
Match (n)–[r]–>(m)
```

Figure 7. UC2 via Neo4j.

```
INSERT INTO nodes (id, name)
VALUES (1, '1');
```

Figure 8. UC3 via MySQL and PostgreSQL.

```
MERGE (:GRAPH { id: '23',name: '23' })
```

Figure 9. UC3 via Neo4j.

```
INSERT INTO edges (a,b)
VALUES (2, 7);
```

Figure 10. UC4 via MySQL and PostgreSQL.

```
MATCH (to:GRAPH {name: '23'})
MATCH (from:GRAPH {name: '14'})
MERGE (to)–[:connects]–>(from)
```

Figure 11. UC4 via Neo4j.

```
UPDATE nodes SET id=6060  WHERE id=2;
UPDATE edges SET source=6060  WHERE source=2
UPDATE edges SET target=6060  WHERE target=2
```

Figure 12. UC5 via MySQL and PostgreSQL.

```
MATCH (n:GRAPH {id: "1"}) SET n.name="new_name"
```

Figure 13. UC5 via Neo4j.

```
DELETE FROM edges
WHERE source = 2 and target = 3;
INSERT INTO edges (source, target)
VALUES (2, 42);
```

Figure 14. UC6 via MySQL and PostgreSQL.

```
MATCH (s:GRAPH { name: '13' })–[r:connects]–>(t:GRAPH{
         name: '19'})
DELETE r
MATCH (source:GRAPH {name: '23'})
MATCH (target:GRAPH {name: '14'})
MERGE (source)–[:connects]–>(target)
```

Figure 15. UC6 via Neo4j.

```
MATCH (n:GRAPH) where n.name='3'
OPTIONAL MATCH (n)−[r]−()
DELETE n,r
```

Figure 16. UC7 via MySQL and PostgreSQL.

```
MATCH (n:GRAPH) where n.name='3'
OPTIONAL MATCH (n)−[r]−()
DELETE n,r
```

Figure 17. UC7 via Neo4j.

UC7–UC8: The remaining use cases simulate the deletion process for the model graph and its elements: deleting a node (UC7), as shown in Figures 16, for MySQL/PostgreSQL, and 17 for Neo4j; and deleting an edge (UC8), as shown in Figure 18 for MySQL/PostgeSQL and in Figure 19 for Neo4j.

5) Time measurement: The ModCP prototype tool is a Windows Forms application and thus, it uses a single-threaded apartment model [24]. We specified a single processor to run the threads of this process to improve the performance by reducing the number of times the processor cache is reloaded [25]. As shown in Figure 20, ProcessorAffinity was used to associate the threads of the process to a single processor. Then, we set the overall priority of the above associated process to high by using ProcessPriorityClass.High.

Each of the use cases were implemented on the candidate database technologies ten times, and the time taken to process the query were recorded from the user interface. The average of query processing time to implement a use-case taken by the each database was recorded to evaluate the database technologies. Similarly, the space measurement was noted from the user interface of the database technology.

6) System information: The system used for performing the experiment runs Microsoft Windows 10 Enterprise. It possesses an Intel Core i7-7700 CPU @ 3.60 GHz, 3600 MHz, 4 cores, 8 logical processors, and 8 GB of RAM. During the experiment, no user programs other than the test programs were running. A basic internet connection was on but not used during the experiment.

```
DELETE FROM edges
WHERE source = 2 and target = 3;
```

Figure 18. UC8 via MySQL and PostgreSQL.

```
MATCH (n:GRAPH {name: '13'})−[r:connects]−>(n:GRAPH{
        name: '19'})
DELETE r
```

Figure 19. UC8 via Neo4j.

```
var testProcess = Process.GetCurrentProcess();
testProcess.ProcessorAffinity = (System.IntPtr)1;
testProcess.PriorityClass = ProcessPriorityClass.High;
```

Figure 20. Setting the processor priority and affinity for running the experiment.

III. Results and analysis

We describe and analyze the results from our objective and subjective evaluation for the databases. Results are provided in terms of the measures used for each of the evaluations.

The complete results of the study are shown in Table II. Log–log plots of the results are available online [26]; we provide one sample in Figure 21, for UC2. We examine the results for individual use cases in subsequent subsections.

In all the plots we present for this comparative study:

1) we sort the data according to the number of edges in the graphs, resulting in the following sequence: G_1, G_4, G_7, G_2, G_5, G_8, G_3, G_6, and G_9;
2) we plot edge count on the x-axis and time taken in milliseconds on the y-axis;
3) we plot the data on log–log scales because the core results grow rapidly, otherwise obscuring their trends; and,
4) we prioritize the number of edges over the number of nodes to compare the database technologies, as number of edges tends towards being quadratic in the number of nodes, hence dominating.

Furthermore, we attempt to fit a linear model $\log y = p \log x + k$ (base e) to the log–log data, sorted by edge count, for each technology/use case combination. Because the lower ends of this data involve numbers of edges that are linear in the number of nodes, we consider only the uppermost six data points for each in fitting the linear model. We note that such a procedure has potential statistical imprecision, but suffices for the trend comparison between technologies in which we are interested. We report values for the *coefficient of determination* $R^2 \in [0, 1]$ (the proportion of the variance in the dependent variable that is predictable from the independent variable), but we acknowledge that this gives only some information about the goodness of fit of the model: low values can occur for well-fitting models (e.g., when the fitted line is nearly parallel with the x-axis) and high values can occur even when the fit is not obviously good. We also consider the visual fit in cases where our analyses depend on the evaluation of the model.

1) Examination of the use cases: We examine the results on the basis of the use cases.

UC1: *Create or store a graph.* UC1.png [26] shows the time taken by the individual database technologies to store the artificially generated graph; Python-CSV does not require any time to store the model as the dataset is already in CSV format, so its data does not appear on the plot (the logarithm of 0 is undefined). UC1regression.png [26] shows the linear regressions on the larger graphs.

We see that MySQL and PostgreSQL are roughly collinear (their linear regressions place their slopes at 0.95 and 0.93 with constant -3.66 and -3.50, respectively). Neo4j clearly has higher overhead (linear regression constant is 1.65) but its computation time appears to grow slightly more slowly than for the SQL variants (linear regression slope is only 0.72):

TABLE II. TIME TAKEN TO PROCESS THE USE CASES (IN MILLISECONDS).

Dens.	Gr.	Technology	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8
2%	G_1	Python-CSV	**0**	**1**	**1**	**1**	**2**	**2**	**3**	**2**
		MySQL	5	2	153	156	155	155	156	154
		Neo4j	665	18	3	3	6	7	11	**2**
		PostgreSQL	187	30	61	66	63	63	66	65
	G_2	Python-CSV	**0**	**11**	**3**	8	41	13	39	12
		MySQL	188	13	151	154	152	158	155	155
		Neo4j	6,852	223	**3**	**3**	**11**	**6**	**13**	**3**
		PostgreSQL	236	157	63	67	62	63	64	66
	G_3	Python-CSV	**0**	1,923	325	19	2,459	1,694	2,281	1,659
		MySQL	14,763	965	152	153	152	156	155	157
		Neo4j	47,970	1,753	**4**	**3**	**14**	**7**	**12**	**3**
		PostgreSQL	12,989	**1,586**	63	66	65	67	65	66
10%	G_4	Python-CSV	**0**	**2**	**2**	4	**3**	**3**	**4**	3
		MySQL	27	**2**	157	153	152	156	155	156
		Neo4j	2,172	157	3	**3**	8	7	13	**2**
		PostgreSQL	214	138	62	65	62	65	64	65
	G_5	Python-CSV	**0**	86	**3**	153	103	68	70	72
		MySQL	694	**57**	155	154	158	157	157	159
		Neo4j	8,936	480	4	**4**	**9**	**7**	**13**	**3**
		PostgreSQL	519	387	61	66	64	67	66	66
	G_6	Python-CSV	**0**	9,327	358	3,093	7,231	7,295	8,640	7,411
		MySQL	74,971	4,294	158	153	152	157	154	159
		Neo4j	357,118	2,935	**3**	**4**	**13**	**6**	**15**	**4**
		PostgreSQL	64,322	**1,822**	63	66	66	64	63	66
25%	G_7	Python-CSV	**0**	**4**	**2**	**3**	**3**	**4**	**3**	**3**
		MySQL	481	11	153	155	157	156	156	155
		Neo4j	2,328	396	3	4	8	7	12	**3**
		PostgreSQL	498	284	60	65	63	65	64	65
	G_8	Python-CSV	**0**	163	7	83	281	170	174	89
		MySQL	1,572	**52**	153	155	157	156	155	157
		Neo4j	25,739	1,067	**3**	**4**	**12**	**8**	**14**	**3**
		PostgreSQL	1,325	819	59	66	64	65	66	64
	G_9	Python-CSV	**0**	14,216	363	8,954	26,227	22,683	19,593	20,279
		MySQL	130,136	21,980	154	155	161	157	155	158
		Neo4j	1,262,200	8,139	**4**	**4**	**15**	**7**	**13**	**4**
		PostgreSQL	121,820	**5,128**	64	65	65	66	64	67

extrapolating the fitted models, we would expect the time for Neo4j and PostgreSQL to be equal when the edge count be in the vicinity of 4.5×10^{10}. Obviously, this assumes that the growth characteristics can be extrapolated in this manner; but, aside from the dubiousness of this extrapolation (the fitted model for Neo4j has an R^2 of only 0.73 and visually is not a great fit), this intersection point would only occur for truly enormous graphs, far beyond any we have encountered or that are likely in practice, even in extreme situations. Python-CSV outperforms all other competition, but this is a local anomaly as we will see examining the results for the other use cases.

UC2: *Read/access a graph.* UC2.png [26] (also shown in Figure 21) shows the time taken to retrieve the graphs from the database into memory. Both MySQL and Python-CSV were inexpensive to retrieve the model for graphs with relatively few edges when compared to PostgreSQL and Neo4j; however, the costs for all the technologies remained low ($\leq$1,067 ms for Neo4j) even in the worst case.

From UC2regression.png [26] we can see that the steeper slopes of MySQL and Python-CSV lead to a crossover point at around 10^6 edges when these two technologies become more expensive to use than Neo4j and PostgreSQL. We found that MySQL and Python-CSV have approximately the same cost (their slopes are 1.02 and 1.03 with constant -6.78 and -7.34, respectively). Similarly, Neo4j and PostgreSQL exhibit similar growth patterns (their slopes are 0.45 and 0.43 with constant 1.31 and 1.29, respectively). Based on these observations, the best database technology for smaller graphs is either MySQL or Python-CSV, with PostgreSQL or Neo4j being preferable for larger graphs (above 10^6 edges), for UC2.

UC3: *Update a graph; Create a node with no edges.* UC3.png [26] shows the performance comparison of the candidate database technologies to create a new node in an existing graph. UC3regression.png [26] shows the linear regressions on the larger graphs.

We see that Python-CSV and Neo4j have the lowest costs cost for G_2, but Python-CSV then scales poorly, leading to the highest cost for G_9 (Python-CSV shows a high linear regression slope of 1.38). We also found that MySQL, Neo4j, and PostgreSQL take essentially constant time to realize the use case (their linear regressions place their slopes at 0.01, 0.02, and 0.01, respectively).

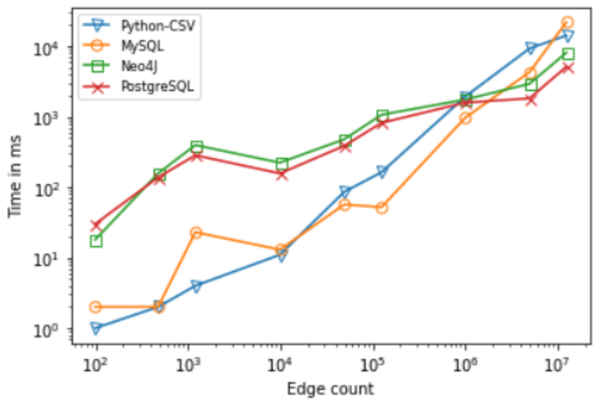

Figure 21. Edge count versus computation time for UC2.

Neo4j is the best option for UC3 as the base cost to create a node is the cheapest and remains constant for larger graphs. The slightly higher but constant cost for PostgreSQL makes a viable alternative as well, relative to UC3.

UC4: *Update a graph; Create an edge between existing nodes*. UC4.png [26] shows the time taken by the individual database technologies to create edges in the edge relations (for relational databases) as well as in the link database (for non-relational databases). UC4regression.png [26] shows the linear regressions on the larger graphs. We see that Python-CSV has near linear growth with respect to edge count (slope is 0.81) and so it is not competitive with the alternatives. MySQL and PostgreSQL take roughly the same time (slopes are 0.01 and 0.01, with constants 5.04 and 4.23, respectively). Neo4j also shows essentially constant performance (slope is 0.02) with lower fixed cost (constant is 1.02). Relative to UC4, MySQL, PostgreSQL, and Neo4j are all viable candidates.

UC5: *Update a graph; Rename a node*. UC5.png [26] shows the time taken by database technologies to rename the nodes in the node and edge relations (for relational databases) as well as in the link database (for non-relational databases). UC5regression.png [26] shows the linear regressions on the larger graphs.

We see that Python-CSV involves the highest query processing times (slope is 0.91 with constant -4.95). In contrast, MySQL, Neo4j and PostgreSQL show essentially constant performance (slopes are 0.01, 0.05, and 0.01, respectively) although the fixed cost is somewhat higher for MySQL (constants are 5.02, 1.82, and 4.1, respectively).

UC6: *Update a graph; Change source and target nodes of an edge*. UC6.png [26] shows the time taken by the individual database technologies to update a graph stored therein by modifying an existing edge. This involves changing the source and target node of an edge in the database. UC6regression.png [26] visualizes the linear regressions.

Python-CSV shows linear growth (slope is 1.04) making it unsuitable for larger graphs. We observe that both MySQL and PostgreSQL require constant processing time (slope is 0.01 for both). Neo4j has lower fixed cost than PostgreSQL (constants are 1.91 and 4.15, respectively).

TABLE III. SPACE REQUIREMENTS (IN BYTES) FOR DATABASE TECHNOLOGIES, RELATIVE TO G_9.

Technology	**Space**
MySQL	514,234,210
PostgreSQL	534,520,112
Neo4j	583,381,354

UC7: *Delete a graph; Delete a node and its corresponding edges*. UC7.png [26] shows the time taken by the individual database technologies to delete a node. We performed the operation by first deleting the node and then deleting the edges related to it. UC7regression.png [26] visualizes the linear regressions.

The performance of Python-CSV is competitive only for the smaller graphs; its linear growth (slope is 0.94) excludes it from further consideration with respect to UC7. MySQL, Neo4j and PostgreSQL provide essentially constant performance (slopes are 0.01, 0.02 and 0.01, respectively). However, the fixed cost for Neo4j is better than for PostgreSQL or MySQL (constants are 5.06, 2.29, and 4.21, respectively).

UC8: *Delete a graph; Delete a specific edge*. UC8.png [26] shows the time taken by the individual database technologies to delete a specific edge. This involved changing the source and target node of an edge in the database. UC8regression.png [26] visualizes the linear regressions.

Python-CSV displays near-linear growth in performance (slope is 1.05) and is competitive only for the smallest graphs. MySQL, Neo4j, and PostgreSQL display essentially constant performance (slopes are 0.01, 0.04, and 0.01, respectively), although the fixed cost of Neo4j is lower (constants are 5.03, 0.62 and 4.16, respectively).

IV. Discussion

We discuss remaining issues relative to this study.

A. Space and Energy

We did not include the space taken by the graph in the database technology as an evaluation measure because the difference between them was trivial, as illustrated in Table III when measured for G_9. For Python-CSV, the size of the dataset is identical to the size of the database. For PostgreSQL and MySQL, the database size was almost identical. For Neo4j, the size of the database was slightly higher than for MySQL and PostgreSQL, but it was not significant enough to be used as an indicator of definite better performance.

Furthermore, energy consumption of alternative approaches is another factor that we have not studied here, of growing concern in the field [e.g., 27].

```
TRUNCATE TABLE edges;
```

Figure 22. Deleting the graph via MySQL and PostgreSQL.

B. Deleting the graph

As shown in Figure 22, Truncate was used for clearing the graph data while maintaining the relation schema for future storage of the graphs. Figure 23 shows the process of deleting all the nodes and relationships from the database while maintaining the labels used, such as "GRAPH".

C. Other datastore technologies

When we were selecting the candidate datastore technologies for our study, we focused on picking one technology from relational databases and one from non-relational databases. As for relational databases, we picked MySQL as being the most popular free RDBMS. We also included PostgreSQL, an object-based RDBMS, in our study as it is widely known as "the world's most advanced RDBMS database," and on the fact that not many comparative studies have been conducted to compare it with MySQL. The existing studies focus on the difference between the features they offer instead of on their actual performance differences [28]. We picked up Neo4j as it is a graph database and made specifically to deal with graphs.

While we carefully designed the study plan, we recognize that other datastore options could have been valid. For non-relational databases, MongoDB, a popular document-based NoSQL database, could also have been a better alternative based on its ability to support huge volumes of both data and traffic. For relational databases, Oracle could have been a good alternative choice. In the end, it was not feasible to try all alternatives and so a selection was needed.

D. Threats to validity

In this section we discuss the threats to validity of our work.

1) Internal Validity: Changes in the variable under observation could very well be caused by additional variables or variations in such variables, which may be related to the manipulated variable but not explicitly modelled [29].

a) Selection Bias: We had to choose a graph generation model. We selected the Barabási–Albert model as a representation of random scale-free graphs. Other variations of growth network model can be found in Buckley and Osthus [30], which presents a directed preferential attachment model. Dorogovtsev et al. [31] and Drinea et al. [32] introduce a variation on the Barabási–Albert model. There also exist other models, such as the "copying model" [33]; Erdős and Rényi [34] present a random graph model which is much smaller than many real-world applications. We concluded that due to having been extensively studied from heuristic and experimental points of view made Barabási–Albert model a reliable choice to generate the datasets.

2) External Validity: Threats to external validity are conditions that limit the ability to generalize the study results and we explain these threats in this section.

```
MATCH (n)
DETACH DELETE n
```

Figure 23. Deleting the graph via Neo4j.

a) Selection of Subjects: There are four major kinds of non-relational database: key–value store [35], wide-column store [36], documents store [37], and graph store [38]. However, we did not evaluate all four kinds in our work. Our goal was to evaluate database technologies to find if any would be a good fit to implement offline storage mechanism in SDATs. We evaluated relational and non-relational databases and the study plan was inline with other existing work and therefore we can be confident in applying the results of our study to other SDATs without loss of generality [39]–[41].

b) Settings: Threats to external validity may also arise from the environment in which the experiments are conducted. The extent to which the results of an experiment can be generalized from the set of environmental conditions created by the researcher to other environmental conditions can greatly impact the generalizability of the results. The details of the windows machine used to conduct this study has been provided in Section II-C6. We evaluated this study, as well as the latter studies, on the same Windows machine.

3) Conclusion Validity: Every empirical study establishes relationships between the treatment, represented by the independent variables, and the outcomes, represented by the dependent variables [42]. Conclusion validity refers to the belief in the ability to derive conclusions from the relationships between the independent variables and the dependent variable.

a) Reliability of Measures: Reliability depends on a variety of factors, including, but not limited to, poor question wording, bad instrumentation, and subjective measures. Threats to conclusion validity may be caused by these factors. In our work, we used objective as well as subjective measures to evaluate the storage mechanisms. However, subjective measures may be considered to be unreliable as it requires the judgment of the researcher [43].

The objective measures consist of time and space taken to process a use case. Our results for this study uses average of all the runs; the actual processing time may be lower or higher than the average run time. The subjective measures used to evaluate the storage mechanisms may not accurately represent the measures being used universally in industrial practice [44]. However, factors such as software productivity, development technology, and interaction with the customer may vary substantially across different teams of developers even if they were working on building the same product. In fact, making software engineering processes, projects, and products reproducible is a challenge [45]. Therefore, when using a technology that may or may not be new to the people who have to work with it, it is important to consider the human factor and not just the objective data. Therefore, we believe that the subjective measures that we used enhance the reliability of our results more than any concerns about validity that may arise because of these measures.

V. Conclusion and Future Work

As our results indicate, we do not see any of the technologies coming out on top for all the use cases. Instead, we must focus on the results from individual use cases in combination with the results from our subjective evaluation to judge each candidate database technology.

Python-CSV is clearly the best technology for UC1 in all respects because it will necessarily always require zero time to perform. For UC2–UC8, we see that Python-CSV has poor scalability both in terms of nodes and edges, as compared to its best competitors. We therefore abandon Python-CSV from further consideration.

MySQL and PostgreSQL have comparable performance; however, PostgreSQL clearly outperforms MySQL in maintaining the graphs. MySQL is best in storing smaller graphs; PostgreSQL scales better in storing graphs and outperforms MySQL for larger datasets.

As for Neo4j, it performed poorly in retrieving the model and had the worst performance of all technologies in storing the model. Nonetheless, we did find that both Neo4j and PostgreSQL exhibit near constant performance and excellent scalability for UC3–UC8. The base processing time of Neo4j tends to be less than PostgreSQL, which is more desirable for any use case. Thus, Neo4j is the better database technology to maintain existing data in the database. It is also the newest technology among the considered candidate database technologies which offers greater flexibility but weaker security and less level of support.

Based on our study, SDATs which have a higher demand to store/retrieve new models would find PostgreSQL to be the better fit. Whereas SDATs which are inclined towards analyzing a limited quantity of software at a given time and involving high maintenance of the models in the database would find Neo4j as the most suitable option.

We thus conclude that PostgreSQL and Neo4j are the most likely candidates for an SDAT for which scalability and the full slate of our use cases are desirable. However, it remains to be seen whether this conclusion is maintained when the costs accruing from a database connector are factored in.

We note that, to utilize any of these database technologies, it is necessary to also use a database connector that provides programmatic access to the core-memory representations. While it is easy to assume that the cost of this access be negligible, the reality is likely different. It is also possible that a far simpler approach, such as object serialization, could suffice to provide reliable persistence for an SDAT. Both these points require further investigation to address.

Acknowledgment

This work was supported in part by a Collaborative Research and Development Grant from the Natural Sciences and Engineering Research Council of Canada in collaboration with Find it EZ Software Corporation.

References

[1] A. M. Davis, E. H. Bersoff, and E. R. Comer, "A strategy for comparing alternative software development life cycle models," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1453–1461, 1988.

[2] S. Horwitz, "Identifying the semantic and textual differences between two versions of a program," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1990, pp. 234–245.

[3] W. E. Riddle and R. E. Fairley, *Software Development Tools*. Berlin: Springer, 2012.

[4] N. E. Fenton and M. Neil, "Software metrics: A roadmap," in *Proceedings of the Future of Software Engineering*, 2000, pp. 357–370.

[5] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the International Conference on Software Engineering*, 2013, pp. 672–681.

[6] R. Dyer, "Bringing ultra-large-scale software repository mining to the masses with Boa," Ph.D. dissertation, Iowa State University, 2013.

[7] J. A. McDermid and K. H. Bennett, "Software engineering research: A critical appraisal," *IEE Proceedings—Software*, vol. 146, no. 4, pp. 179–186, 1999.

[8] Y. Shafranovich, *Common format and MIME type for Comma-Separated Values (CSV) files*, RFC 4180, 2005.

[9] D. Crockford, *The application/json media type for JavaScript Object Notation (JSON)*, RFC 4627, 2006.

[10] J. Han, E. Haihong, G. Le, and J. Du, "Survey on NoSQL database," in *Proceedings of the International Conference on Pervasive Computing and Applications*. IEEE, 2011, pp. 363–366.

[11] Neo4j, Inc., "Neo4j support." [Online]. Available from: https://neo4j.com/docs/. [retrieved: May 2023].

[12] Google, Inc., "Datastore." [Online]. Available from: https://cloud.google.com/datastore. [retrieved: May 2023].

[13] J. Han, "Supporting impact analysis and change propagation in software engineering environments," in *Proceedings of the IEEE International Workshop on Software Technology and Engineering Practice Incorporating Computer Aided Software Engineering*, 1997, pp. 172–182.

[14] H. Men, "Fast and scalable change propagation through context-insensitive slicing," Ph.D. dissertation, University of Calgary, 2018.

[15] I. J. Farkas, I. Derényi, A.-L. Barabási, and T. Vicsek, "Spectra of "real-world" graphs: Beyond the semicircle law," *Physical Review E*, vol. 64, no. 2, pp. 026 704–1–026 704–12, 2001.

[16] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[17] P. Wan, T. Wang, R. A. Davis, and S. I. Resnick, "Fitting the linear preferential attachment model," *Electronic Journal of Statistics*, vol. 11, no. 2, pp. 3738–3780, 2017.

[18] A. Hagberg, P. Swart, and D. Schult, "Exploring network structure, dynamics, and function using NetworkX," Los Alamos National Laboratory, Tech. Rep., 2008.

[19] D. A. Schult and P. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the Python in Science Conference*, vol. 2008, 2008, pp. 11–15. [Online]. Available from: http://conference.scipy.org/proceedings/scipy2008/paper_2/.

[20] R. Diestel, *Graph Theory*. Berlin: Springer, 2017.

[21] P. E. Black, *Dictionary of algorithms and data structures*, National Institute of Standards and Technology, 1998. [Online]. Available from: https://xlinux.nist.gov/dads/.

[22] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 1, pp. 2:1–2:26, Oct. 2008.

[23] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring

network structure, dynamics, and function using NetworkX," in *Proceedings of the Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., 2008, pp. 11–15, https://networkx.org/ [retrieved: May 2023].

[24] Microsoft, Inc., "COM threading model." [Online]. Available from: https://docs.microsoft.com/en-us/previous-versions/dotnet/netframework-3.0/ms182351(v=vs.80)?redirectedfrom=MSDN [retrieved: April 2023].

[25] ——, "Processor affinity." [Online]. Available from: https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process.processoraffinity?view=net-5.0 [retrieved: May 2023].

[26] K. Singh and R. J. Walker, "Log–log plots pf the results." [Online]. Available from: https://doi.org/10.6084/m9.figshare.22561564 [retrieved: May 2023].

[27] E. Jagroep, J. Broekman, J. M. E. M. van der Werf, S. Brinkkemper, P. Lago, L. Blom, and R. van Vliet, "Awakening awareness on energy consumption in software engineering," in *Proceedings of the International Conference on Software Engineering*, 2017, pp. 76–85. [Online]. Available from: https://doi.org/10.1109/ICSE-SEIS.2017.10.

[28] R. Poljak, P. Poščić, and D. Jakšić, "Comparative analysis of the selected relational database management systems," in *Proceedings of the International Convention on Information and Communication Technology, Electronics and Microelectronics*. IEEE, 2017, pp. 1496–1500.

[29] H. K. Wright, M. Kim, and D. E. Perry, "Validity concerns in software engineering research," in *Proceedings of the FSE/SDP Workshop on the Future of Software Engineering Research*, 2010, pp. 411–414.

[30] P. G. Buckley and D. Osthus, "Popularity based random graph models leading to a scale-free degree sequence," *Discrete Mathematics*, vol. 282, no. 1-3, pp. 53–68, 2004.

[31] S. N. Dorogovtsev, J. F. F. Mendes, and A. N. Samukhin, "Structure of growing networks with preferential linking," *Physical Review Letters*, vol. 85, no. 21, pp. 4633–4636, 2000.

[32] E. Drinea, M. Enachescu, and M. Mitzenmacher, "Variations on random graph models for the web," Computer Science Group, Harvard University, Tech. Rep. TR-06-01, 2001.

[33] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal, "Stochastic models for the web graph," in *Proceedings of the Symposium on Foundations of Computer Science*, 2000, pp. 57–65.

[34] P. Erdős and A. Rényi, "On the evolution of random graphs," *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, vol. 5, no. 1, pp. 17–60, 1960.

[35] J. Kepner, V. Gadepally, D. Hutchison, H. Jananthan, T. Mattson, S. Samsi, and A. Reuther, "Associative array model of SQL, NoSQL, and NewSQL databases," in *Proceedings of the IEEE High Performance Extreme Computing Conference*, 2016, pp. 1–9.

[36] V. Sharma and M. Dave, "SQL and NoSQL databases," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 2, no. 8, pp. 20–27, 2012. [Online]. Available from: https://www.researchgate.net/publication/303856633_SQL_and_NoSQL_Databases/link/5758557f08ae9a9c954a7573/download.

[37] V. Guimaraes, F. Hondo, R. Almeida, H. Vera, M. Holanda, A. Araujo, M. E. Walter, and S. Lifschitz, "A study of genomic data provenance in NoSQL document-oriented database systems," in *Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine*, 2015, pp. 1525–1531.

[38] J. J. Miller, "Graph database applications and concepts with Neo4j," in *Proceedings of the Southern Association for Information Systems Conference*, 2013, pp. 141–147. [Online]. Available from: https://aisel.aisnet.org/sais2013/24.

[39] G. Petri, "A comparison of Oracle and MySQL," *SELECT Journal*, vol. 12, no. 1, article 6, Independent Oracle Users Group, 2005.

[40] S. Batra and C. Tyagi, "Comparative analysis of relational and graph databases," *International Journal of Soft Computing and Engineering*, vol. 2, no. 2, pp. 509–512, 2012.

[41] S. Rautmare and D. M. Bhalerao, "MySQL and NoSQL database comparison for IoT application," in *Proceedings of the IEEE International Conference on Advances in Computer Applications*, 2016, pp. 235–238.

[42] X. Zhou, Y. Jin, H. Zhang, S. Li, and X. Huang, "A map of threats to validity of systematic literature reviews in software engineering," in *Proceedings of the Asia–Pacific Software Engineering Conference*. IEEE, 2016, pp. 153–160.

[43] A. Ampatzoglou, S. Bibi, P. Avgeriou, M. Verbeek, and A. Chatzigeorgiou, "Identifying, categorizing and mitigating threats to validity in software engineering secondary studies," *Information and Software Technology*, vol. 106, pp. 201–230, 2019.

[44] J. D. Herbsleb and A. Mockus, "An empirical study of speed and communication in globally distributed software development," *IEEE Transactions on Software Engineering*, vol. 29, no. 6, pp. 481–494, 2003.

[45] B. C. D. Anda, D. I. K. Sjøberg, and A. Mockus, "Variability and reproducibility in software engineering: A study of four companies that developed the same system," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 407–429, 2008.