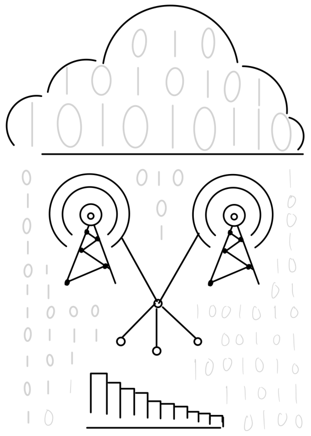
ICR: Database Storage Strategies

TEAM aMALGamation

Manisha Jaiswal
Asher Bursnall
Lauren Champlin
Graham Stookey

Revised Jun 15, 2022



CSCI 370 Advanced Software Engineering

Advisor: Dr. Christopher Painter-Wakefield

Client: ICR

Summer 2022

Table 1: Revision history

Revision	Date	Comments
New	May 19 2022	Completed Sections: I. Introduction II. Functional Requirements III. Non-functional Requirements IV. Risks V. Definition of Done XIII. Team Profile Bibliography Appendix A – Key Terms
Rev – 2	May 26 2022	Updated Sections: II. Functional Requirements III. Non-functional Requirements Bibliography Completed Sections: VI. System Architecture
Rev – 3	June 2 2022	Completed Sections: VII. Software Test and Quality VIII. Ethical Considerations
Rev – 4	June 9 2022	Updated Sections: VII. Software Test and Quality Completed Sections: Appendix B – Queries
Rev – 5	June 13 2022	Updated Sections: VII. Software Test and Quality IX. Results Completed Sections: VIII. Project Ethical Considerations X. Project Completion Status XI. Future Work XII. Lessons Learned
Rev – 6	June 15 2022	Adjusted All Sections Based on Feedback

Table of Contents

I. Introduction

We are working with a defense company that is based in Aurora, Colorado called ICR. They are having issues with efficiency and scalability in their current database (DB) setup due to the highly relational and highly dimensional nature of their data. Their current system is based on a postgres DB in Amazon RDS that is inefficient at producing results from queries and is not scalable to the number of nodes and data that they need to include in the system.

ICR is looking to find a DB system that fits their needs - able to store highly dimensional and relational data, well established with the ability to find troubleshooting help, and retrieves that data in a memory-efficient and quick fashion. Their final goal is to be given a suitable candidate for a DB system with the ability to traverse the DB based on any of the many relationships between the data.

ICR is a defense company, so their data is classified and cannot be given to us with the purpose of testing the system. As such, we make the data ourselves, using code provided by the client as a model to base the data off of. The source of data is a component of our final deliverable product and must be an apt model of the relational and dimensional nature of the data ICR uses.

This project is highly exploratory and comprised mainly of research and data creation with code mainly to test the DB efficiency and demo its ability to perform the tests that ICR requires it pass for their needs. Thus, on the software side of things, our only stakeholders are ICR and their employees, who will be the ones to use, maintain, and depend upon the DB system that we suggest to them.

II. Functional Requirements

We provide and justify a database/data storage recommendation. To do that, we need a way to generate test data that represents the kind of data ICR would actually use, and test it on the following set of requirements to ensure that they hold:

- Scalability - need to be able to handle between ten thousand and ten million different records without a significant decrease in efficiency
- Query Response Time - the response time on queries should be quicker than the current system, Amazon RDS by a factor of 2
- Concurrency - the data is dynamic and so needs to be efficient with concurrent reads and writes to the data
- Native or well documented Python and Java Drivers
- Must be compliant and runnable on both Kubernetes and Docker

The one optional requirement that ICR put forward was the ability of the program to hold and display geospatial data. This is mostly based on the DB having the native data types necessary to hold longitudinal and latitudinal data and show it accurately on a map.

III. Non-Functional Requirements

The DB must follow several general requirements that pertain to the ease of use of the scheme and aid in the ability of ICR to run the database on their system independently.

- Open Source - the DB must be able to be run natively on a closed system without using outside servers to store the data due to ICR requiring the use of the database securely offline
- Documentation and Support - the DB must be well documented and have an active community for support and troubleshooting issues via community forums and company support
- Ease of Use - the DB must be relatively straightforward to set up and navigate

IV. Risks

One of the most concerning risks to us is researching and spending time learning a DB that is not suitable for the requirements that ICR has put out to us. The impact, however, is minor, as we can simply pivot to a new DB without a high consequence for doing so.

Another risk is the possibility of not passing tests required by ICR with the working code or ending up choosing the wrong database for particular requirements. This could mean delving into a database that has unsatisfactory documentation or poor ease of use. This, similar to our first risk, requires us to be able to pivot quickly to a new database or system, though it has a more moderate effect, as we will have put substantial time into that DB.

Aside from research risks, when creating software for someone, even if we do not believe that they will use it, there is the risk that we will hurt them or their software in some way. This can be through a test script only meant to test dummy data making its way into actual software, which can insert dummy data or delete necessary data from the DB. Though the risk of this is severe, it is unlikely as the only code we made is to test only our data and is unlikely and not recommended to be used in any of their software.

Additionally, we are dealing with technology that we do not fully understand, as only one of our group has any experience in dealing with databases and thus there is a high risk that it will take much longer to understand these concepts and implement them than we expect. The impact of this is dependent on how long it takes for us to work past this obstacle, which can range from the inability to test multiple databases to adding a couple more hours onto our workload to get ourselves up to speed.

These risks indicate a need to be particular and careful in our assessment and testing of the databases before giving any final recommendations to ICR.

V. Definition of Done

Our client's definition of done is to have working code that shows the passing of tests suggested by ICR, fulfillment of requirements provided by ICR, and ultimately our recommendation of a database. These tests must show the efficiency and accuracy of the chosen DB, as well as being able to scale from 10 thousand nodes to 10 million nodes without severely affecting the performance and memory capacity required of the DB system. We also test pulling data connected with different types of relationships, and ensuring that the DB can pull data based on the different properties the data in the nodes contains.

The product will be delivered as a demo of our working code showcasing the ability of chosen DB scheme using the model data we have created.

VI. System Architecture

Test Data Generation: In order to facilitate the testing of the performance of different database schemes, mock relational data is generated and used within the databases. The program depicted in Figure 1 was designed by our team based on the client's data requirements to generate social, object-oriented data with various layers of relationality.

1. First, the program is used to generate datasets with varying numbers of nodes, from 10 thousand to 10 million.
2. Then, using an Amazon RDS database (the client's current database platform), we measured execution time of a set of queries with different levels of complexity to garner time data which is used as a control.
3. Once the control data had been gathered, we repeated the process of using the same datasets and the same set of queries against the chosen graph databases, gathering execution time, and then comparing the results.

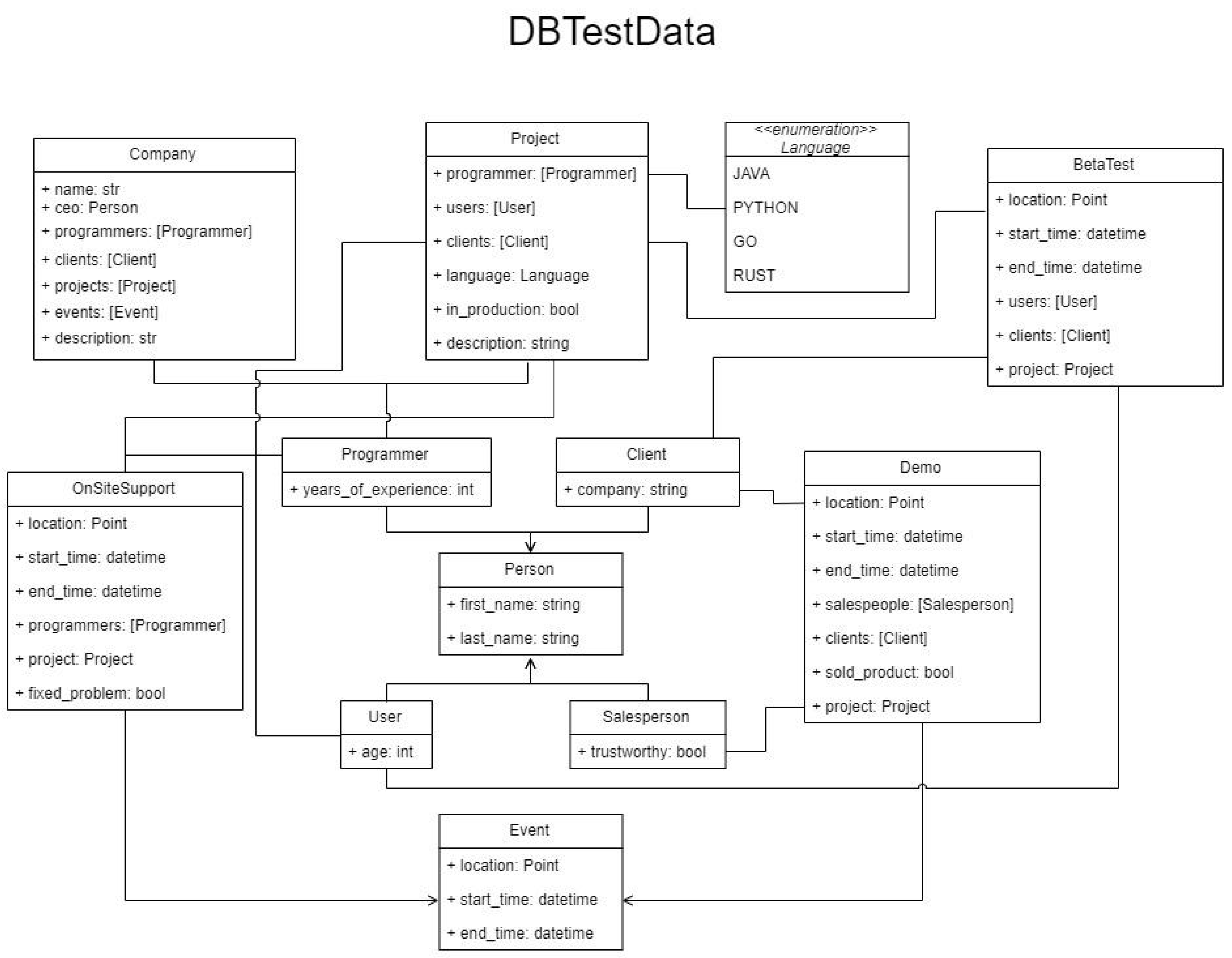


Figure 1: UML of Mock Data

Database Architecture Scheme:

1. Importing data into a graph database is done in a similar fashion as importing data into other relational databases, varying according to the database's API. Typically, the insertion data provided by an importer is stored temporarily in a processing container, where the database management tasks are handled by the database platform (such as in the case of Amazon RDS, Datastax Enterprise Graph, etc.) or processed independently (such as with open source Databases like Neo4j and Dgraph). Once the data has been processed and verified, the data is then loaded into the database.

2. Generally, the storage of the data within a graph database exists in multiple forms, both as tables and as abstract graphs. The graph structure allows for relational data to be mapped and visualized, allowing for the user to gain insight into possible correlations hidden within the relationships in the data, as seen in Figure 2.

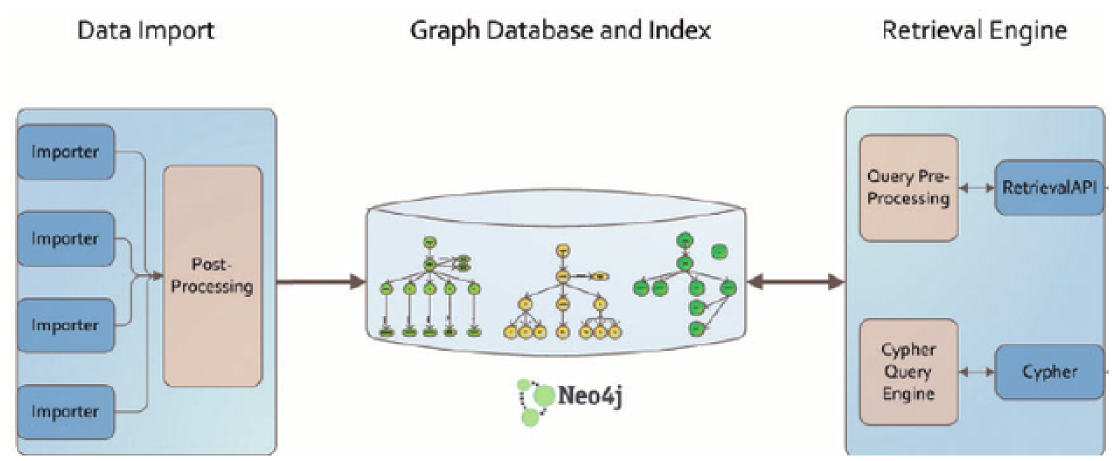


Figure 2: Depiction of Neo4j Database Set-Up

Source: https://www.researchgate.net/publication/273385434_Combining_computational_models_semantic_annotations_and_simulation_experiments_in_a_graph_database

VII. Software Test and Quality

Software Quality Plan:

The software in our project mainly relates to the generation of our tested dummy data. Thus, we need to test that the dummy data was generated correctly. It must have proper randomness and relationships. We must also be able to show that the dummy data is similar to the data it is representing. We can ensure the randomness and relationships by running queries on the data and testing that it matches to what we expect based on the generation algorithm. We use pgadmin to ensure that it's generated randomly and correctly, with the number of results in pgadmin showing the same answers as what is returned by our queries.

We ensure that dummy data is representational by using starter code given to us by our client, ICR, and expanding upon it to make it more relational and dimensional. This was done by adding more classes such as User and Beta Test which both have multiple complex relationships, and by ensuring that each class has several different variables held within them.

Since our code deals primarily in testing databases to determine their differences, we needed to ensure that all other variables besides the database were kept constant. This includes standardizing our test data, controlling the logistics within our computers to make the environment as much as possible, and attempting to standardize our databases such that one does not have a massive difference or discrepancy in how it will be tested versus used. We have done these by:

- conducting tests on group members' laptops to attempt to remove the variable of extra memory or processing power from using a PC or server system
- downloading our data directly from an instance of an amazon rds database, so that there is no difference in the data used
- using the open source version of each database to ensure that we do not utilize the capabilities of a cloud network or extra utility not given to open source users

In general, we have also committed to quality using proper code reviews, frequent documentation and reports of databases, languages, and observations, and the use of statistics. By discussing and documenting our findings, the rest of the group is able to point out inconsistencies and help when one person is struggling. Hearing this constructive criticism elevates our project as a whole, and improves the overall quality of the work we produce, as more than one pair of eyes looks over everything that goes into our project.

VIII. Project Ethical Considerations

- The ACM/IEEE Principles pertinent to our project are 2 (CLIENT AND EMPLOYER), 4 (JUDGMENT), 7 (COLLEAGUES), and 8 (SELF).
    - 2: We must act in the best interest of our client by finding and justifying the best database for their requirements.
    - 4: We must maintain integrity and independence in our professional judgment when choosing which database(s) to recommend.
    - 7: We must respect and work with each other throughout this project.
    - 8: We must participate in lifelong learning by researching databases and getting experience with them.
- The ACM/IEEE Principles that are in most danger of being violated are probably 2 and 4.
    - 2: We may fail to find and sufficiently justify a suitable database which would not be in the best interest of the client.
    - 4: We may be individually biased towards a particular database when collaborating to decide which to recommend.
- Michael Davis Tests
    - Mirror Test: We want to come to a decision that, when we look in the mirror, is a choice that we wholeheartedly believe is the correct option for ICR's specific needs and is not bogged down by significant indecision.
    - Reversibility Test: We must be able to step away from our biases into the shoes of our client and be sure that our choice is the one that most benefits them as opposed to being the one that we are most invested in, even should this mean that we choose the option that is not the one we are biased towards.

The ethical considerations for the project if our software quality plan is not properly implemented are:

- Disruption of the database - as any code we produce is not designed to be implemented into a working database system for ICR, and thus is reliant on our dummy data and thus could add, remove, or change existing data within ICR's system.
- Loss of time and money - should we produce findings that are incorrect and instruct ICR to look into a database that is not correct for them, it is highly possible that this will lead to a waste of their resources in implementing a system that will not work in the way they wish it to.

IX. Results

Summary: The tables and graphs below showcase the results of our tests for both local and remote instances of Neo4j and Dgraph databases (see below for graph-specific analysis). The results for each instance were compared to the control data from the Amazon RDS tests, as well as to each other.

Locally Hosted Database Results: Testing revealed that query response times of a locally hosted instance of both the Neo4j and Dgraph databases performed markedly better than Amazon RDS. Given, this result was to be expected as our response time data is heavily influenced by the lack of network factors with querying a locally hosted database, but does illustrate to our client how much faster the option to locally host a database can be for improving performance. This is not even an option for Amazon RDS.

Remotely Accessed Database Results: Taking an additional step to test remotely accessed instances of Neo4j and Dgraph, we aimed to garner data on how network connectivity may affect query response time and get more comparable results. As expected, the response times fell significantly, but were still 50% + faster than the Amazon RDS control.

Graph Description: The graphs below compare the response times (y-axis) of the given databases for each of our 12 test queries (x-axis). The stark differences in time can be accounted for by the various complexities of the queries (see Appendix B), where the more complex the query, the higher the number of degrees of relationships traversed and thereby the longer the query response time.

	Time (ms)				
Query number	**Amazon RDS (Remote)**	**Neo4j (Local)**	**Neo4j (Remote)**	**Dgraph (Local)**	**Dgraph (Remote)**
1	57.539870	2.355545	31.635129	1.490015	21.889479
2	50.044236	1.534848	30.892503	1.420004	21.003518
3	213.958485	2.826855	34.054840	1.259996	19.155879
4	51.656498	3.819065	61.618361	3.967936	24.595408
5	70.523819	1.869087	27.367992	3.519973	25.331785
6	58.438413	1.201665	34.700837	1.539998	21.409275
7	53.076708	1.265800	26.038775	1.599974	21.045503
8	154.304828	10.113301	24.450519	19.787290	46.203520
9	70.510418	1.201665	36.895380	1.670005	21.205899
10	68.111293	1.189797	28.521855	1.310000	21.297321
11	449.524235	1.016490	23.303397	2.100009	20.579166
12	83.142943	1.917994	34.695182	11.157143	33.949739

Table 1: 10 thousand Node Query Response Time Results

	Time (ms)		
Query number	**Amazon RDS (Remote)**	**Neo4j (Local)**	**Neo4j (Remote)**
1	67.459776	3.056800	57.393992
2	54.423439	2.560296	34.618361
3	9614.941718	4.797168	27.056084
4	55.691928	4.725637	29.485564
5	735.317549	3.798835	22.564104
6	51.617295	2.879810	37.810862
7	53.112971	2.133217	17.313459
8	52.757487	10.027165	23.615339
9	67.853193	3.403976	26.894057
10	252.705779	2.350075	24.553778
11	4150.944991	1.955340	21.435878
12	2303.482697	5.790105	31.729150

Table 2: 100 thousand Nodes Query Response Time Results

Neo4j Data Analysis:

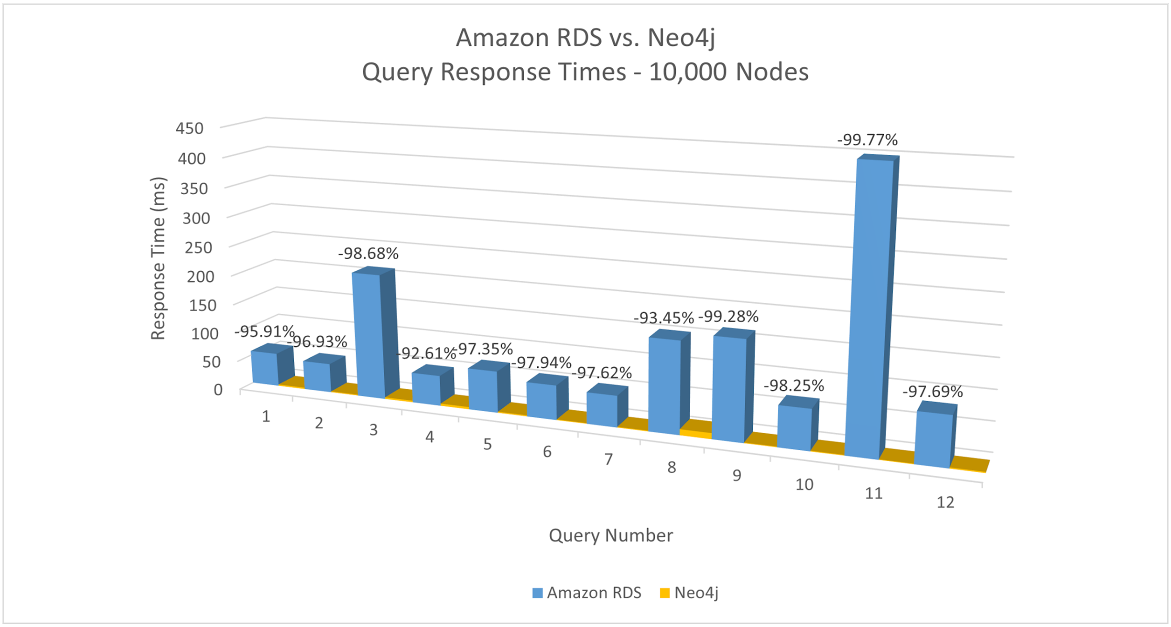


Figure 3: 10k node percent changes in query response time per query from Amazon RDS to a local instance of Neo4j. The results for this test were stark, with query response times decreasing on average 97.12% from Amazon RDS to Neo4j.

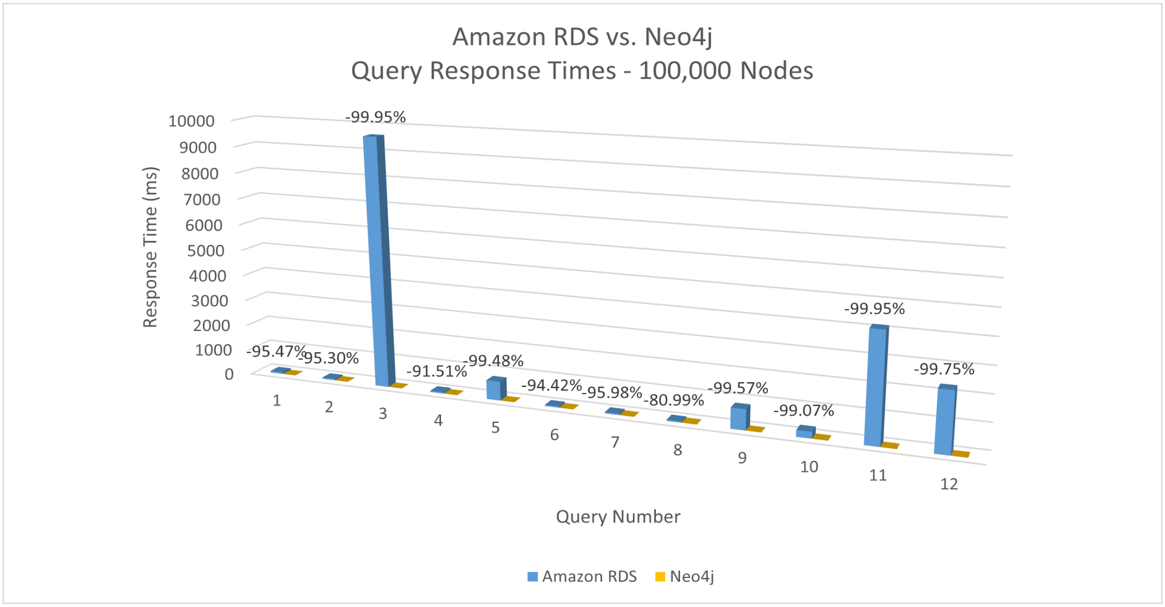


Figure 4: 100k node percent changes in query response time per query from Amazon RDS to a local instance of Neo4j. The results for this test were also stark, with query response times decreasing on average 95.95% from Amazon RDS to Neo4j.

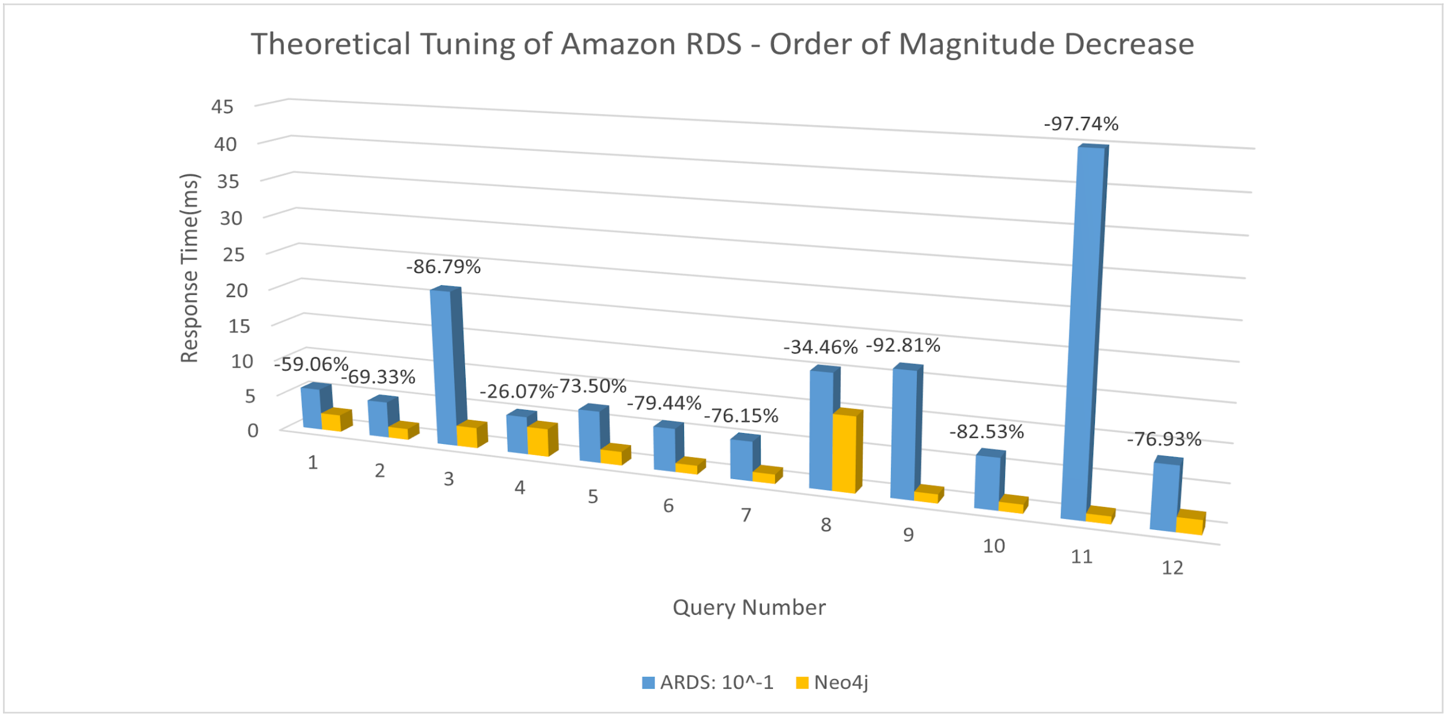


Figure 5: 10k node percent changes per query from Amazon RDS with a decrease by an order of magnitude in query response time to simulate theoretical performance tuning improvements. Result: Avg. percent decrease: 71.23%.

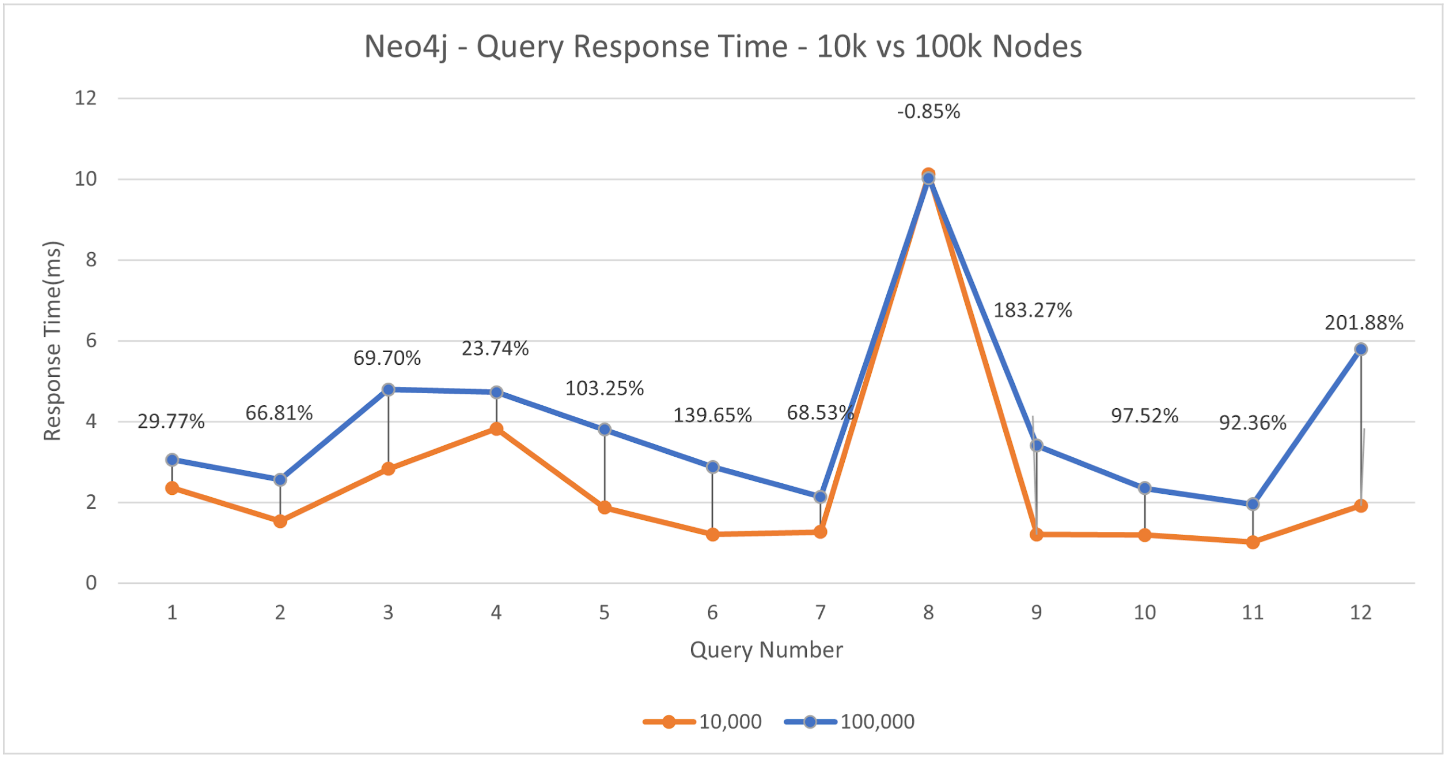


Figure 6: Percent changes per query from 10k to 100k on local Neo4j database instance. Avg. percent increase: 89.64%.

Dgraph Data Analysis:

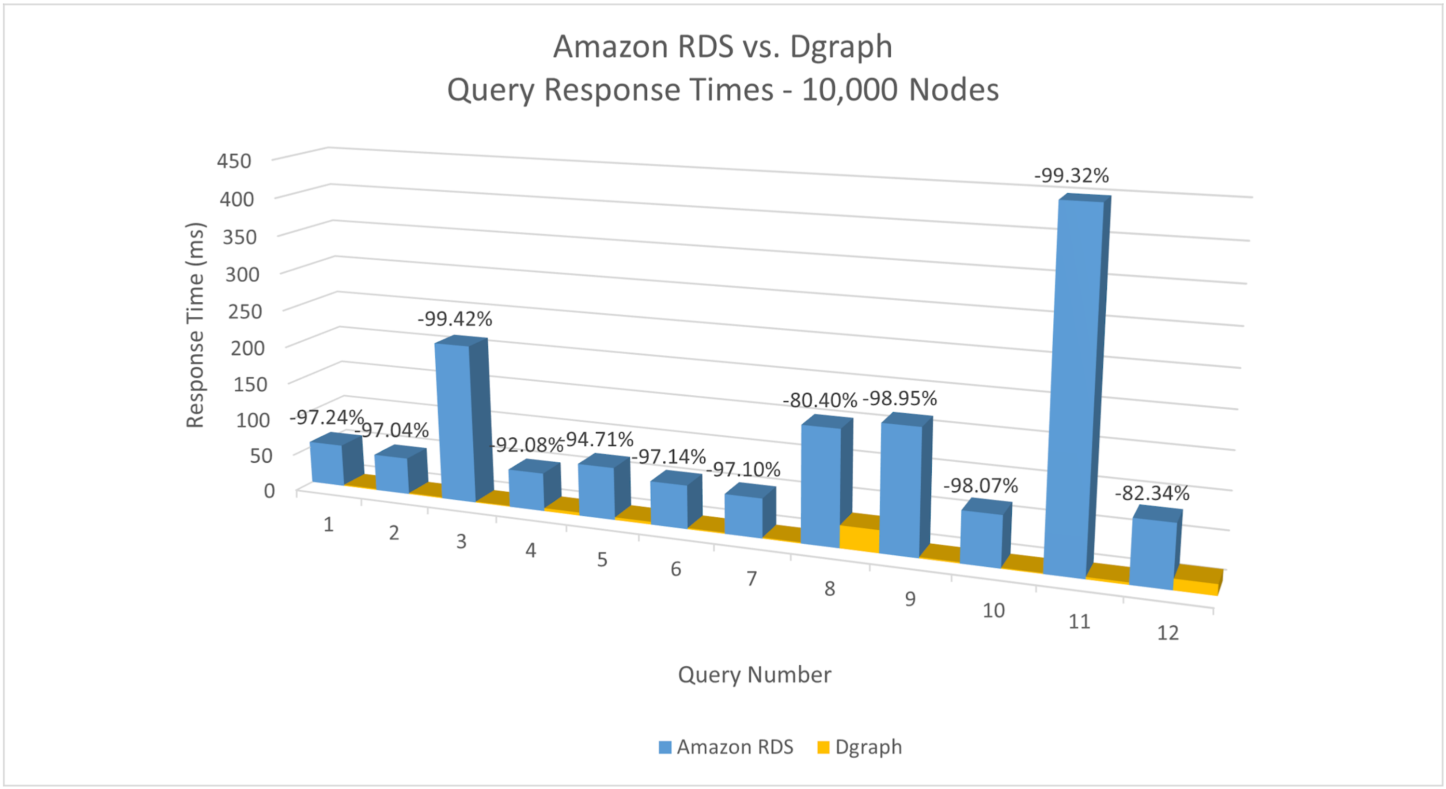


Figure 7: 10k node percent changes per query from Amazon RDS to Dgraph. Result: Avg. percent decrease: 94.48%.

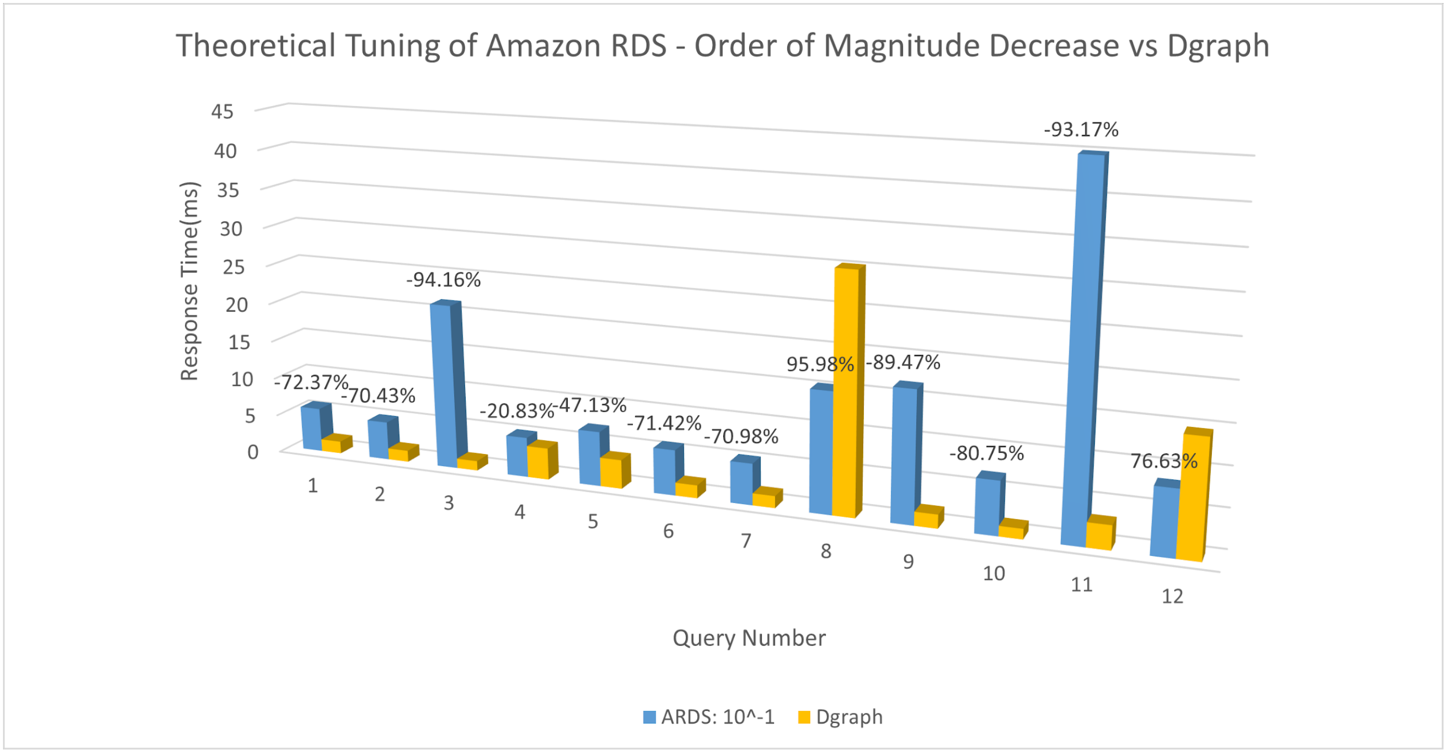


Figure 8: 10k node percent changes per query from "Tuned" Amazon RDS to Dgraph. Avg. percent decrease: 44.84%.

Database Ratings: In order to quantify certain non-tangible, user-oriented aspects of the databases, we categorically rank-ordered each database in order to put a number to other critical factors that played into our final recommendation which could not be qualitatively accounted for. The table below shows our team's experience-based rankings of these intangible but critical factors:

	Neo4j	Dgraph	ArangoDB
Documentation	9	7	5
Community Support	9	9	3
Ease of Use	10	8	7
Data Migration	10	3	3
Installation	8	5	6
Totals	46/50	32/50	24/50

* Rated on a Scale from 1 - 10

X. Project Completion Status

Over the course of our time working on this project, we have been able to:

- generate highly relational and dimensional mock data that resembles the type of data that our client works with in the order of 10 thousand and 100 thousand nodes of data
- load data into Amazon RDS Postgres database
- migrate data to Neo4j from Postgres via ETL tool
- migrate data to Dgraph from Postgres via csv files
- learn and generate complex queries in SQL and Cypher that traverse several relationships and are restricted by several different dimensions
- time the queries and analyze the results

Due to technical obstacles and time restrictions we were unable to:

- generate 1 million and 10 million nodes of data
- successfully migrate data to ArangoDB
- query the same database from multiple computers simultaneously (concurrency)
- use the Java driver for any database
- use Neo4j in Docker and use any database in Kubernetes

XI. Future Work

As this project was highly exploratory, there is much to be done in the future beyond the scope of the project. Our goals for this project were to find databases that fit within the basic requirements put forward by ICR and test them to ensure that they also match the technical requirements. Though we have met most of the requirements, there is still a necessity to test the databases further. We could not generate 1 million or 10 million nodes due to the number of relationships they required. Additionally, we were unable to get remote connections to test concurrency. And we only have four members, which is not enough to get a clear idea of how the database reacts under pressure. These tests may be vital to ICR, due to how much data they need to store and query, and should be run before implementing any databases.

Additionally, though our task was only to research databases and choose one that meets all the requirements from ICR, the logical next phase after our work is implementing the database. Though this is not something that we will deal much in, our documentation and experience in working with these databases will be used in this secondary step in order to guide the creation and transition to the new database.

XII. Lessons Learned

One lesson we learned while trying to implement different softwares was how effective a combination of documentation research and pair programming can be in solving technical problems in our project. When stuck, we were often able to talk through issues with the team or get help searching for solutions on the internet. We also learned a lot about the benefits and limitations of documentation and interviews with professors. In the words of Professor CPW, “One lesson learned is that the more experienced technicians don’t always have the answer”. As a team, we grew to further understand the power of good communication. Any difficulties we experienced in the team were usually solvable via discussion. We also learned how to break down big problem statements into weekly goals. Finally, we learned a valuable lesson in how easy it is to overlook critical dependencies and unintentionally introduce errors into your design. By neglecting the dependency of query response time on variances in network speed, we made the mistake of defining our control data on something that was dependent on network speed, which obviously is not an independent variable. This part of the project really hammered home the ethics lessons we received this semester, knowing that we had to be straightforward and acknowledge / document these errors.

XIII. Team Profile

Manisha Jaiswal
Senior
Computer Science - Data Science/ Mathematics
Hometown: India (but US Citizen)
Work Experience: Undergraduate RA (currently), Computer Science Mines, TA AMS Department Mines, Tutor at Red Rocks Community College, Lakewood CO
Hobbies: Traveling, Cooking.

I am excited to work with my project team and ICR this summer, and also stepping into research as an undergraduate RA at Mines.

Graham Stookey
Senior
Computer Science / Business
Hometown: Littleton, Colorado
Work Experience: Staff Songwriter - Noise Block Music Grp., Private Guitar Teacher, Delivery Driver - FedEx, landscaper, warehouse worker
Hobbies: Grilling, producing lofi-hiphop, fly fishing, backpacking, hiking, snowboarding, folding laundry for my wife.

Since I've been at Mines, I have found a real passion for all things software engineering and hope to work in the aerospace/defense industry after graduation.

Lauren Champlin
Senior
Computer Science
Hometown: Spokane, Washington
Work Experience: Walmart Cashier, Mines DECtech TA, Discrete Math TA, ICR software development intern
Clubs: Society of Women Engineers, ACM-W
Hobbies: Rock climbing, baking, origami

Looking forward to working with ICR again.

Asher Bursnall
Sophomore
Computer Science - Cybersecurity
Hometown: Colorado Springs, CO
Work Experience: Summer Coach for Kids on Bikes, Assistant Camp Manager for Kids on Bikes, Jedi at Colorado School of Mines, Mines CSCI 261 TA
Hobbies: Bouldering, mountain biking, hiking, wallowing in existential dread, traveling, baking, cooking, eating, sleeping.

I am looking forward to working with ICR on this project and learning more about database structure.

Bibliography

[1] “Apache tinkerpop.” [Online]. Available: https://tinkerpop.apache.org/. [Accessed: 17-May-2022].

[2] “Enterprise distributed graph database,” *DataStax*. [Online]. Available: https://www.datastax.com/products/datastax-graph. [Accessed: 17-May-2022].

[3] G. Vogt, “Neptune,” *Amazon*, 2000. [Online]. Available: https://aws.amazon.com/neptune/. [Accessed: 17-May-2022].

[4] “Get started with Dgraph,” *dgraph*. [Online]. Available: https://dgraph.io/docs. [Accessed: 17-May-2022].

[5] *GraphQL*. [Online]. Available: https://graphql.org/. [Accessed: 17-Jun-2022].

[6] “Introduction to arangodb documentation: Arangodb documentation,” *Introduction to ArangoDB Documentation | ArangoDB Documentation*. [Online]. Available: https://www.arangodb.com/docs/stable/. [Accessed: 17-May-2022].

[7] J. Ellingwood, “The different types of databases - overview with examples,” *Prisma's Data Guide*. [Online]. Available: https://www.prisma.io/dataguide/intro/comparing-database-types. [Accessed: 17-May-2022].

[8] Learn the basics of Neo4j and the property graph model, “Take the neo4j fundamentals course with Neo4j Graphacademy,” *Free Neo4j Courses from GraphAcademy*. [Online]. Available: https://graphacademy.neo4j.com/courses/neo4j-fundamentals/. [Accessed: 17-May-2022].

[9] Neo4j Community, “Official release: 3 essentials of neo4j 3.0, from scale to productivity & deployment,” *Neo4j Graph Data Platform*, 24-May-2016. [Online]. Available: https://neo4j.com/blog/neo4j-3-0-massive-scale-developer-productivity/#capabilities-data-size. [Accessed: 13-Jun-2022].

[10] “Neo4j documentation - NEO4J documentation,” *Neo4j Graph Data Platform*. [Online]. Available: https://neo4j.com/docs/. [Accessed: 17-May-2022].

[11] “Neo4j ETL tool - interactive relational database data import - neo4j labs,” *Neo4j Graph Data Platform*. [Online]. Available: https://neo4j.com/labs/etl-tool/. [Accessed: 02-Jun-2022].

[12] *OrientDB Manual*. [Online]. Available: http://www.orientdb.com/docs/last/index.html. [Accessed: 17-May-2022].

[13] “PostgreSQL on Amazon RDS - Amazon Relational Database Service.” [Online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_PostgreSQL.html. [Accessed: 17-May-2022].

[14] Sisense Team, “5 advantages of using a redshift data warehouse,” *Sisense*, 18-Mar-2022. [Online]. Available: https://www.sisense.com/blog/5-advantages-using-redshift-data-warehouse/. [Accessed: 17-May-2022].

[15] *XTDB*. [Online]. Available: https://xtdb.com/. [Accessed: 17-May-2022].

Appendix A – Key Terms

Include descriptions of technical terms, abbreviations and acronyms

Term	Definition
DB	*Database*
RDS	*Relational Database Service*
REST	*Representational state transfer - can be used to interface with web*
ORM	*Object Relational Mapping - a technique that lets you query and manipulate data from a database using an object-oriented paradigm*
Scalability	*Scalability is the degree to which a system can adjust to changes in demand without having to make major changes*
Dimensionality	*Dimensionality refers to how many attributes a dataset has (aka number of columns)*

Appendix B – Queries

1. Return what company the programmer with first name 'fp210' works for
   a. SQL:
   ```
   SELECT * from company WHERE id=(SELECT company_id FROM programmer
   WHERE first_name='fp210')
   ```
   b. Cypher:
   ```
   MATCH (p:Programmer {firstName: 'fp210'})-[w:`works
   for`]->(c:Company) RETURN p, w, c
   ```
   c. DQL:
   ```
   query1(func: allofterms(first_name, "fp210")) {
   works_for { name }
   }
   ```
2. Return all programmers that work for the company with name 'C80'
   a. SQL:
   ```
   SELECT * from programmer WHERE company_id = (SELECT id from company
   WHERE name='C80')
   ```

b. Cypher:
```
MATCH (p:Programmer)-[w:`works for`]->(c:Company{name: 'C80'})
RETURN p, w, c
```
c. DQL:
```
query2(func: allofterms(name, “C80”) {
~works_for { first_name }
}
```
3. Return how many companies have exactly three employees
   a. SQL:
   ```
   SELECT COUNT(*) from company WHERE 3=(SELECT COUNT(*) FROM programmer
   WHERE programmer.company_id = company.id)
   ```
   b. Cypher:
   ```
   MATCH (p:Programmer)-[w:`works for`]->(c:Company)
   WITH c, count(p) as employees
   WHERE employees = 3
   RETURN count(employees)
   ```
   c. DQL:
   ```
   query3(func: eq(count(~works_for), 3)) {
   companies: count(uid)
   }
   ```
4. Return the company with the highest number of programmers and how many programmers that is
   a. SQL:
   ```
   SELECT COUNT(*), company_id FROM programmer GROUP BY company_id ORDER
   BY COUNT(*) DESC LIMIT 10
   ```
   b. Cypher:
   ```
   MATCH (p:Programmer)-[w:`works for`]->(company:Company)
   RETURN company.name, COLLECT(p.firstName) as programmers, count(p) as
   numProgrammers
   ORDER BY SIZE(programmers) DESC LIMIT 10
   ```
   c. DQL:
   ```
   var(func: has(headed_by)) {
   num_employees as count(~works_for)
   }
   query4(func: uid(num_employees), orderdesc: val(num_employees),
   first: {
   name
   count(~works_for)
   }
   ```
5. Return programmers that have more than three years of experience and also manage on site support for a project written in Python
   a. SQL:
   ```
   SELECT DISTINCT programmer.first_name, programmer.last_name,
   onsitesupport.project_id FROM programmer, onsitesupport, project,
   onsitesupport_programmer_association WHERE
   programmer.years_of_experience > 3 AND
   programmer.id=onsitesupport_programmer_association.programmer_id AND
   onsitesupport.id =
   ```

```
onsitesupport_programmer_association.onsitesupport_id AND
onsitesupport.project_id IN (SELECT id FROM project WHERE
language='Python')
```

   b. Cypher:
```
MATCH (oss:Onsitesupport)-[:`provided
for`]-(pj:Project{language:'Python'})
MATCH (oss)-[:`managed by`]-(pg:Programmer)
WHERE pg.yearsOfExperience > 3
RETURN oss.id, collect(pg.id) as Programmers,
collect(pg.yearsOfExperience) as YearsExperience
```

   c. DQL:
```
query5(func: allofterms(language, "Python")) @normalize {

~provided_for {
     managed_by @filter (gt(years_of_experience, 3)) {
     first_name: first_name
     last_name: last_name
     }
}
}
```

6. Return all projects written in Rust for which a specific client attended a demo of the project
   a. SQL:
```
SELECT project.id FROM project, demo, demo_client_association WHERE
1001 =demo_client_association.client_id AND
demo.id=demo_client_association.demo_id AND demo.project_id =
project.id AND project.language='Rust'
```

   b. Cypher:
```
MATCH
(client:Client{id:1001})-[r:attends]-(demos:Demo)-[r2:demoing]-(rustP
rojects:Project{language:'Rust'})
RETURN rustProjects
```

   c. DQL:
```
query6(func: uid(0x1bef)) {
first_name
~attends {
demoing @filter (allofterms(language, "Rust")) {
     uid
     description
     }
}
}
```

7. Return all project currently in production for a given company
   a. SQL:
```
SELECT DISTINCT project.id FROM company, project WHERE
project.company_id = 11 AND project.in_production='1'
```

b. Cypher:

```
MATCH (company:Company{id:11})-[r:`belongs to`]
-(projectsNotInProduction:Project{inProduction:true})
RETURN projectsNotInProduction,company
```

c. DQL:

```
query7(func: uid(0x2ae)) @cascade @normalize {
~belongs_to @filter (eq(in_production, true)) {
project: description
}
}
```

8. Return all clients of companies with at least one project written in Rust who attended a demo of a project written in Rust that was presented by a trustworthy salesperson

a. SQL:

```
SELECT COUNT(DISTINCT client.id) FROM client,
client_company_association as cca, demo_client_association as dca
WHERE client.id=cca.client_id AND cca.company_id IN (SELECT id FROM
company WHERE ( (SELECT COUNT(*) FROM project WHERE
company_id=company.id AND language='Rust') > 0)) AND client.id =
dca.client_id AND dca.demo_id IN (SELECT DISTINCT demo.id FROM demo,
salesperson, demo_salesperson_association AS dsa WHERE
demo.project_id IN (SELECT id FROM project WHERE language='Rust') AND
demo.id = dsa.demo_id AND salesperson.id = dsa.salesperson_id AND
salesperson.trustworthy = '1')
```

b. Cypher:

```
MATCH (companies)<-[r:`belongs
to`]-(rustProjects:Project{language:'Rust'})
MATCH (clients)-[h:hires]->(companies)
MATCH (rustDemos:Demo)-[d:demoing]->(rustProjects)
MATCH (twSalesPeople:Salesperson{trustworthy:'true'})
MATCH ((rustDemos)-[:`presented by`]-(twSalesPeople))
MATCH (clients)-[:attends]->(rustDemos)
RETURN collect(DISTINCT clients)
```

c. DQL:

```
var(func: has(trustworthy)) @filter (eq(trustworthy, true)) {
salespeople as uid
}
var(func: has(designed_for)) @filter (allofterms(language, "Rust")) {
     rust_projects as uid
     all_companies as belongs_to
}
var(func: has(headed_by)) @filter (uid(all_companies)) {
     rust_companies as uid
}

query8(func: uid(rust_companies)) {
     ~belongs_to @filter (uid(rust_projects)) {
           ~demoing {
                presented_by @filter (uid(salespeople)) {
                           ~presented_by {
                                count(attends)
                           }
                }
           }
      }
}
```

9. Return all programmers who manage on site support for projects written in Rust at a specific company
    a. SQL:
    ```
    SELECT DISTINCT programmer.first_name FROM programmer, onsitesupport,
    onsitesupport_programmer_association AS opa,
    project_programmer_association AS ppa, project, company WHERE
    onsitesupport.id = opa.onsitesupport_id AND programmer.id =
    opa.programmer_id AND onsitesupport.project_id = ppa.project_id AND
    ppa.programmer_id = programmer.id AND project.language = 'Rust' AND
    project.company_id IN (SELECT id FROM company WHERE name='C492') AND
    programmer.company_id IN (SELECT id FROM company WHERE name='C492')
    ```
    b. Cypher:
    ```
    MATCH (C492:Company{name:'C492'})<-[r:`belongs
    to`]-(rp:Project{language:'Rust'})
    MATCH (rp)<-[:`provided for`]-(oss:Onsitesupport)
    MATCH (pg:Programmer)-[:`managed by`]-(oss)
    RETURN pg.id as programmer, C492.id as company, rp.id as RustProject,
    oss.id as onSiteSupportID
    ```
    c. DQL:
    ```
    query9(func: allofterms(name, "C492")) @cascade @normalize {
    ~belongs_to @filter (allofterms(language, "Rust")) {
         count(~provided_for)
         ~provided_for {
              managed_by {
                   first_name: first_name
              }
         }
    }
    }
    ```
10. Return how many users use project id 623
    a. SQL:
    ```
    SELECT COUNT(DISTINCT user_p.id) FROM user_p, project,
    project_user_association AS pua WHERE pua.project_id=623 AND
    pua.user_id = user_p.id
    ```
    b. Cypher:
    ```
    MATCH (u:User_P)-[r:`designed for`]-(p:Project {id:623})
    WITH count(u) as Num_Users_of_Proj_623
    RETURN Num_Users_of_Proj_623
    ```
    c. DQL:
    ```
    query10(func: uid(0x21c8)) @cascade {
    ```

```
count(designed_for)
}
```

11. Return all projects that user id 1 uses
    a. SQL:
    ```
    SELECT project.id FROM user_p, project, project_user_association AS
    pua WHERE pua.user_id=1 AND pua.project_id = project.id
    ```
    b. Cypher:
    ```
    MATCH (u:User_P {id: 1})-[r:`designed for`]-(p:Project)
    RETURN p
    ```
    c. DQL:
    ```
    query11(func: uid(0xd17)) {
    ~designed_for {
    description
    }
    }
    ```
12. Return all projects from all companies written in Java or Python which are in production and have more than two programmers managing on site support
    a. SQL:
    ```
    SELECT project.id FROM project WHERE in_production='1' AND
    (language='Java' OR language='Python') AND (SELECT COUNT(DISTINCT
    programmer.id) FROM programmer, onsitesupport,
    onsitesupport_programmer_association AS opa WHERE opa.programmer_id =
    programmer.id AND opa.onsitesupport_id = onsitesupport.id AND
    onsitesupport.project_id = project.id) > 2
    ```
    b. Cypher:
    ```
    MATCH (J_OSS)-[m1:`managed by`]->(jp:Programmer)
    WITH J_OSS,count(jp) as jpCount
    WHERE jpCount > 2
    MATCH (J_OSS:Onsitesupport)-[:`provided
    for`]->(J_proj:Project{inProduction:'true',language:'Java'})
    RETURN J_proj.id as ProjID
    UNION ALL
    MATCH (P_OSS)-[m2:`managed by`]->(pp:Programmer)
    WITH P_OSS,count(pp) as ppCount
    WHERE ppCount > 2
    MATCH (P_OSS:Onsitesupport)-[:`provided
    for`]->(P_proj:Project{inProduction:'true',language:'Python'})
    RETURN P_proj.id as ProjID
    ```
    c. DQL:
    ```
    var(func: has(handled_by)) @filter (gt(count(managed_by), 1)) {
    support_project as provided_for
    }

    query12(func: has(headed_by)) {
    ~belongs_to @filter (anyofterms(language, "Java Python") and
    eq(in_production, true) and uid(support_project)) {
         description
    ```

```
    }
  }
```