SLIDE 1

Hello Everyone, my name is Joel Horne. This is my Final Project presentation, A Comparison of Graph and Relational Database Systems, for COMP519 Advanced Topics in DBMS.

SLIDE 2

This course is taught by Dr. Blum for the Fall 2025 semester at Penn State Harrisburg.

SLIDE 3

This project attempted to recreate the methods used in the source paper: A Comparison of a Graph Database and a Relational Database by researchers at the University of Mississippi. It also provided me with a great opportunity to learn the basics of graph databases. I've always enjoyed how visually appealing and understandable graphs are, and I was excited to learn how graph DBMSs store and retrieve data. By setting up graphs to test, and timing increasingly complex queries, I recreated and expanded the original experiment and found new and interesting results.

SLIDE 4

The database design used in the experiment was carried over from the original paper: 12 databases, the product of four different sizes and three different types of payload data, were created for each DBMS. I used each DBMS's Python connector to create and query each database. For each type (Integer, 8 kibibyte string and 32 kibibyte string) a randomly-generated payload was assigned to each node, and a randomly-selected parent was assigned from the list of created parents. This ensures that each node has one, previously inserted parent, which means that each graph contains a set of what are called arborescences, which are directed acyclic graphs where each node has at most one parent. This is actually a bit stricter than the original

study, which specified using any kind of directed acyclic graph. I used arborescences in my graphs because I felt it may speed up traversal query times compared to simple DAGs. While the relational graphs use an attribute to store the ID of their parent, graph databases use a system of edges (or relations) connected to vertexes (which I call nodes). As we'll see, query times scaled for both size of database and complexity of type, so it takes longer for queries to reach deeper graph depths, and it takes longer to find substrings in these huge 32KB string payloads. This is to be expected: the differences in efficiency between the databases makes the experiment interesting.

SLIDE 5

The queries used to test the database were taken directly from the source paper, with slight modifications. S-Queries count the number of nodes that can be reached from each root node for a given depth. These were accomplished using recursive common table expressions, and the natural recursive feature of the cypher graph query language. The I-Queries count the number of nodes with integer payloads according to a predicate. I1 and 2, which match or search or for values greater than an integer respectively were in the original study. I3, which searches for integers that are congruent to 0 modulo another integer was added by myself. C-Queries count the number of nodes that contain a substring of varying length. The source paper used between 4 and 8 characters. I used between 1 and 8 characters to test if any real difference would show up: I'll talk about that later. The R-Query gets the average depth of all the nodes that match some payload. It was created as a combination test of the traversal and search queries. The search queries were tested using randomly generated search values. Each query was run 10 times with python's timeit module and the run times were averaged.

SLIDE 6

The original study compared 2 databases: MySQL and Neo4j. This provided a good side-by-side comparison between two distinct DBMS's, however I thought it would be interesting to update the study with databases and extensions that had been developed and iterated on over the fifteen years since the original study. The two original databases, Neo4j and MySQL, in this case, its open-source fork MariaDB, were kept on with a number of added databases. The original study found that Neo4j outperformed MySQL for traversal queries, but MySQL performed much better for search queries. Originally I used nine DBMSs/extensions: MariaDB, PostGreSQL, and Aerospike (the relational databases), Neo4j, Memgraph, and Dgraph (the graph databases), OQGRAPH, Apache AGE, and Aerospike Graph (the database extensions). OQGRAPH and Apache AGE are extensions of MariaDB and PostGres respectively. After working for a while I realized it was untenable to test all nine systems and pared it down to 6 [MOVE FORWARD, WIPE BOTTOM ROW]. This was done to account for time, and simply because I disliked the query languages developed for Aerospike and Dgraph. MariaDB, PostGres, and OQGRAPH use SQL, Neo4j and Memgraph use the Cypher query language (in fact, they both use Neo4j's python connector infrastructure), and Apache AGE uses a custom plugin that reads in Cypher queries through a SQL extension (which I will discuss in the experiences section).

SLIDE 7

Before and during development of test infrastructure I developed several hypotheses concerning time efficiency between databases with the goal of comparing results with the original study. I mostly expected to replicate, or at least be congruent to, the original study. In comparing databases, I expected that Neo4j would perform fastest on traversal queries as its nature as a graph database should have lent itself to traversal. Memgraph prioritizes computer memory for data storage (on my machine, total memory usage for all databases was about 5 gigabytes). With this in mind I expected Memgraph to outperform other databases at

least with respect to search queries. Since OQGRAPH was queried in the same manner as MariaDB I expected it to perform about as well as its base. When I talk about results it will show how well these hypotheses track.

SLIDE 8

Before I talk about results I want to share some of the issues I ran into during the experiment, most of which were in the engineering of the test infrastructure. As I talked about before I removed Aerospike, Aerospike Graph, and Dgraph from testing. I was not pleased by their query systems, which I found awkward compared to Cypher and basic SQL, and I came to the conclusion that rather than waste precious time testing them I would focus on query systems I was more familiar with. Apache AGE used a plugin system that loaded cypher queries in through SQL queries, introducing limitations such as reduced predication power, which provided an interesting challenge to work around. I noticed close to the end of development of the testing tools that I was using a different schema than what the source paper used. The source paper used two fields per node to store both incoming and outgoing edges, while I stored one for each node’s parent. I think this may have had an effect on query times, as the schema in the source paper allowed for what seem like more efficient queries, as their implementation did not need to use common table expressions to find nodes in the S0 query. Moving on, I encountered a lot of problems using the Apache AGE extension to insert nodes and edges into the databases. Nodes would insert relatively quickly but edges frequently took prohibitively longer to insert for larger databases. I waited an entire 24 hours, although I should have given up after one, for my generator to insert all the edges in the 100K graphs. This, unfortunately, lead to a gap in my data which will be evident in the results. I’ve also noticed that my randomization method of assigning parents randomly did not produce graphs with large depths. At most, I noticed that the 100K graphs have a max depth of 16. This appears in the data for larger S Queries in that

S128 and S256 take about the same time for all databases. On a positive note, from a user interface perspective, I was pleased with some of the tools used for graph database visualization. I was very pleased with Neo4j's visualization console, which functioned as a UI for all of Neo4j's functionality: its a very quick and responsive console. Memgraph's console, on the other hand, was detailed in its presentation but was very slow to load node and edge visualization. Overall, I enjoyed working with Neo4j's tools the best as they felt robust, responsive, and easy to use.

SLIDE 9

This is an example usage of Neo4j's browser console, currently displaying results of a query that selects all the nodes and parent relations from the onek_integer table. Note that the table is selected as an attribute of the node and not the node as a member of the table. This will be touched on in the results section.

SLIDE 10

When I started the experiment I expected the results to conform relatively closely to the original study, and in some ways it was. Neo4j performed much worse for the traversal queries in this experiment than in the original study, but did not scale in the same way it did in the original study, especially for S0. For instance, queries for larger databases took about half as long as in the original. This may be in part due to the Neo4j instance I used relied on a single database with nodes holding a table attribute, rather than multiple tables. I'm unsure if the original study used this architecture but from what I understand this is the only way to accomplish this. It may also be due to how I made my traversal queries for Neo4j, relying on recursive expressions for traversal to specified depths. The original study does not list any of its graph queries, so I didn't have a good way to compare my query methods.

In terms of speed, PostGres performed the best for traversal, and for integer search queries, but at a higher scale during character search typically came behind Neo4j and Memgraph. I wasn’t expecting this, especially for the traversal queries. I also did not expect Memgraph to perform well for large database search in the 32K character databases when it had performed poorly over lower sized tables. The overall losers of the experiment were of course Apache AGE and also OQGRAPH, which performed much worse than MariaDB, sometimes by a factor of 4 for certain queries. These results defied my expectations in a way I really enjoyed.

SLIDE 11

This table shows the results from the S0 query trial for all integer tables in all databases: all results for all tables are in milliseconds. Note that while PostGres and MariaDB are fastest, Neo4j and Memgraph remain constant at scale.

SLIDE 12

This table shows the results from the S16 query trial over the same domain as the previous table. You can see the clear ramp up for the 100K table. My best explanation for this is because the traversal has a much larger domain to search. Take note that as before PostGres has the best time, yet Neo4j and Memgraph scale the best.

SLIDE 13

This table shows the comparison between the C1 and C4 query trials over the same domain as the previous tables. This table shows clearly how the character queries scale and is relatively congruent to the results for the 32K character tables. At this point you can see where Memgraph and OQGRAPH start to break down for larger payload types.

The entire set of results is available to view on the repository under the Graph DB Data markdown file.

SLIDE 14

These tables show the results from the R1 query trials for the integer and 8K character tables. Note that OQGRAPH has mediocre performance in the integer tables and performs outstandingly for the character tables. This will be touched on in a few slides.

SLIDE 15

So as you can see there are a couple things that can be taken away from this experiment. For these kinds of queries PostGres and Neo4j perform best. PostGres handles search and traversal queries best at lower database sizes and Neo4j comes in second. As stated previously, Memgraph is an outlier in higher size character search for higher databases, as it performed the best for the 100K database with 32 kibibyte strings. I learned not too long into experimentation that the team behind Apache AGE had been laid off about a year ago and the project abandoned, and I would recommend against using it. OQGRAPH performed badly in search, but was middle of the path for most traversal and performed best for the R1 search query over character tables using its built-in search algorithms.

SLIDE 16

An important part of scientific study is replication, so I want to take the last part of the presentation to how I think this experiment could be iterated on and improved. This experiment used a schema different from the source paper, and it would be useful to test on a schema similar to the original and notice any difference as a result. The testing method used by the original paper was slightly more complex as well: it removed outliers from query tests before taking the average, which may also be worth replicating in the future. Memgraph was the only tested database that used its own Docker container rather than running as a service. I learned during the experiment that I was

using an instance of Memgraph that was designed for learning purposes, and was not designed for production. Using the production version of Memgraph in the future may produce better results from one of the worse performers. As stated before, Neo4j used attributes to store table info in each nodes: in the future using distinct Neo4j instances to store each table may produce different results, at the expense of having to use some unwieldy test methods. It may be useful to use larger databases in the future, but I also suspect that it will only show the kind of scale shown in the current data, and would not be useful for comparison to the current data. Finally, it may be useful to create more test queries, such as those that probe to deeper graph depths, and those that scan data such as Dates.

To wrap things up, I’d like to thank Dr. Blum for guidance and look forward to working on a similar experiment in the future, maybe with a bit more rigor then. The repository containing the generators, tests, and results are available via the provided link. Thank you everyone for watching!