An Empirical Evaluation of Neo4J and Apache Cassandra for Astronomical Spatial Queries

Glenn Galvizo
University of Hawaii at Manoa

***Abstract*—Asking "What are stars are near S?" is a problem that may involve iterating through millions, if not billions of stars. The Tycho-2 dataset allows us to reduce our search to sky regions instead of the entire star set. The time to answer similar questions using the distributed column family database Apache Cassandra, was benchmarked against the distributed graph database Neo4J. Overall, Cassandra's query times are more tightly distributed than Neo4J times but Neo4J is able to process queries faster if the correct indexes are applied.**

I. Introduction

A cursory glance of a clear night sky reveals roughly thousands of stars. There are in fact, billions more stars that the human eye cannot see. Answering "Which stars are which when one looks up into the sky?" in an efficient manner is a very active field of research.

Astrometry is a branch of astronomy that involves determining where celestial bodies lie at a given time. A seemingly innocent and basic question we can ask involving astrometry is "What stars do we see near some star S?" The steps we take to answer this are as follows:

1) Finding some known catalog of stars, that includes the star S.
2) Computing the angular separation between star S and each star in the catalog.
3) Filtering out stars that do not meet the criteria of "near".
4) Applying an additional filter of brightness, removing stars that we cannot see from Earth.
5) Presenting the resultant.

The naive and slowest method requires iterating through the entire catalog for our star S, then looping through the entire catalog again for stars that are near S and visible from Earth. For n elements in our catalog, this means our time complexity is $T(n) = O\left(n^2\right)$.

Steps (1), (3), and (4) involve searching through lists, which may hold billions of elements. Given $O\left(n^2\right)$ complexity this suggests that our catalog queries will be slow. There are a few ways to get around this (the most effective being reducing the dataset itself to fit the problem better), but the option explored here is rethinking our data model to access the catalog data more efficiently.

In this paper, we stem away from the traditional RDBMS (relational database management system) approach to this problem and look toward *distributed* no-SQL solutions. In particular, we analyze the graph DBMS solution of Neo4J and the column family solution of Apache Cassandra. This research aims to contribute several benchmarks for astronomical querying systems (star trackers, astrometry research), as well as some characterization of graph and column family data models for star catalog access.

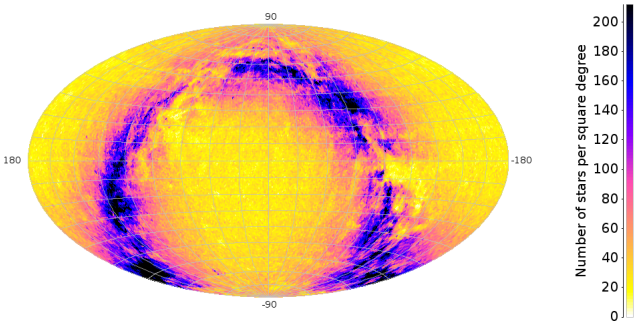

Fig. 1: Heat map of the celestial sphere mapped by the Tycho-2 dataset. There exist roughly 2.5 million stars here. Right ascension varies from left to right, while declination varies from top to bottom.

II. Tycho-2 Dataset

In order to make informed decisions about our data model, we need to understand our data: a star catalog. A star catalog is a collection of stars, their positions, and other traits, enumerated & identified by their catalog numbers. The bulk of the Tycho-2 star catalog comes from the ESA (European Space Agency) astrometric mission Hipparcos. This dataset is available at ftp://cdsarc.u-strasbg.fr/pub/cats/I/259.

Typical star catalogs hold stars as points projected onto a two dimensional map of the sky known as the celestial sphere. Each point is recorded as a point (α, δ) in the *Equatorial coordinate system*. α represents right ascension, which is the celestial equivalent of longitude on Earth. δ represents declination, which is the celestial equivalent of latitude. Figure 1 depicts a heat map of the stars inside the catalog, and is meant to illustrate how positions of stars are recorded.

There exist two main sections of interest here: the catalog files, and the index file. The catalog files hold each star (or astronomical object), their astrometric properties, and their 3-number `TYC` ID. This is formatted as such:

`TYC1`	`TYC2`	`TYC3`
[Region #]	[Running #]	[Component Identifier]

Each `TYC1`, or *GSC Region Number* corresponds to a specific region in space. Region here, refers to a section on

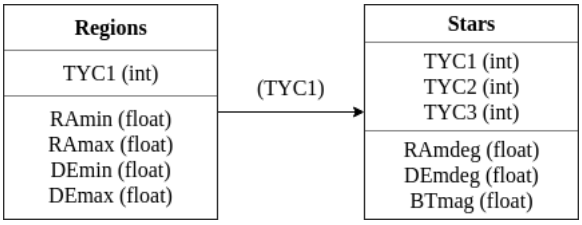

Fig. 2: Depiction of data model used with Cassandra. There exists a *Stars* column family indexed by `TYC1, TYC2, TYC3`, and a *Regions* column family indexed by `TYC1`. Stars exist in regions if they share the same `TYC1` number.

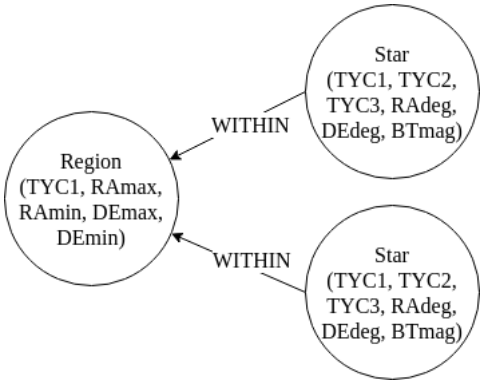

Fig. 3: Depiction of data model used with Neo4J. There exists a collection of *Star* nodes and *Region* nodes, each of which are related by the `WITHIN` relationship.

the celestial sphere. Every GSC Region corresponds to, at most, a $3.75^\circ \times 3.75^\circ$ region of the sky. The running number corresponds to a star inside that region, and the component identifier indicates how many stars exist in that specific star system (typically equal to 1).

Also contained are the proper motions of each star, the epoch (time) associated with the current record, the apparent magnitude (brightness), and errors associated with all of the former. For the queries tested here only the apparent magnitude, the (α, δ), and the `TYC` IDs were recorded with each entry.

The index file is sorted by `TYC1`, and holds the bounds representing each region $(\alpha_{min}, \alpha_{max}), (\delta_{min}, \delta_{max})$. Regions were recorded with each boundary and the appropriate `TYC1` ID.

III. Data Models

Given the Tycho-2 catalog of stars, how can we model our data our data for efficient spatial queries on a static database?

A. Relational Data Model

The relational data model is the most popular data abstraction, and arguably the most natural. Here data is stored in n-dimensional vectors known as tuples. The schema is defined before data is added, meaning that each tuple of a given set has the same length. If thought of as a table, the number of columns is fixed while the number of rows is variable. Interaction between this model involves the use of relational algebra (typically implemented as SQL).

In terms of distributed relational DBMSs, most implementations are consistent, available, but not partition tolerant. This means that every read will always receive the most recent write. Given that our database is static, the biggest concern is partition tolerance. Our system may not operate with message loss across a cluster of computers (nodes).

B. Column Family Data Model

The column family data model can be very loosely thought of as the segmentation of a relational table into several two column tables consisting of a single attribute of the original table and a primary key based off this attribute. These segmented tables can then be grouped into a column family, which represents a group of columns indexed by the same primary key. This segmentation is useful for allowing denormalization, meaning that the schema does not have to be defined before data insertion and that each primary key does not have to map the same amount of columns as other keys. Column family models are known for this denormalization and efficient querying, accessing only the columns required for the query itself. Cons here include restrictive queries, as columns can only be accessed using their primary key.

Apache Cassandra is the column store database that is being tested here, and is a form of a *distributed* DBMS. Cassandra itself is available and partition tolerant, but not consistent. Data in a Cassandra cluster is hash partitioned by the primary key with the additional option of being replicated across the cluster. Cassandra's querying language CQL allows for queries without the primary key, but at the cost of performance.

The data model being implemented with Cassandra involves two column families: *Regions* and *Stars*. The *Stars* family has six columns:

```
TYC1        TYC3        DEmdeg
TYC2        RAmdeg      BTmag
```

where `TYC1, TYC2, TYC3` represent the star's ID, `RAmdeg, DEmdeg` represent the star's position, and `BTmag` represents the magnitude of the star. This is depicted in Figure 2. The primary key for this family is the GSC region `TYC1`, with `TYC2, TYC3` as secondary clustering keys. Efficient queries for stars should involve the primary key `TYC1`.

The *Regions* family has five columns:

```
TYC1        RAmax       DEmax
RAmin       DEmin
```

where `TYC1` represents the region ID, (`RAmin, RAmax`) represent the right ascension limits, and (`DEmin, DEmax`) represent the declination limits. Star exist in the same region if they share the same `TYC1` region ID, which is also the primary key for the *Region* family. Efficient region queries will use only the `TYC1` primary key.

C. Graph Data Model

Relative to the relational data model, the graph data model focuses on the relationship between various tuples. Here data

is stored in nodes as attributes, and relationships are stored as edges between nodes. Data is accessed by searching for nodes or edges, and traversing the graph. Compared to the relational data model, graph data models shine in relationship querying and denormalization. Compared to the column family where data can easily be partitioned across nodes, a graph data model does not allow for the same amount of parallelism.

Neo4J is the graph database that is being tested here, and is also a form of a distributed DBMS. The Neo4J setup that was tested here is available and consistent, but not partition tolerant (like a typical distributed RDBMS). Because data in a Neo4J graph cannot be partitioned like Cassandra, every node in a cluster is a full replica.

The data model being implemented here involves two types of nodes: *Region* and *Star* nodes. A *Star* node has the same attributes as the columns in the Cassandra *Stars* family. *Regions* have all attributes in the columns of the *Regions* family. A `WITHIN` relationship is established between all *Star* nodes that exist in a region.

IV. Methods

The goal of this research was to characterize the running time of various queries under different data models, indices, and architecture.

All experiments were performed on 1–3 node clusters, with each machine of type `n1-standard-1` (2 virtual CPUs, 3.75GB of memory). To normalize the role of the cache (both OS and DBMS), each query set of the list in subsection IV-A was ran 45 times. The last 15 runs were sampled and represents the data in section V. The repository that holds all code and data associated with this project can be found at: https://github.com/glennga/tycho-query

A. Queries

Queries represent read operations on some DBMS. Write operations were not tested here, as stars have not added to this dataset in years. For simplicity, a star is defined as *near* to another star if they share the same region. A star is *naked-eye visible* if its `BTmag` field (apparent magnitude) is below 6.0.

The following queries are to be examined:

1) What are the characteristics of some star S?
2) Which stars are near some Equatorial position (α, δ)?
3) Which stars are near some star S, and what are their characteristics?
4) Which stars are near some Equatorial position (α, δ), and are naked-eye visible?
5) Which stars are near some star S, and are naked-eye visible?

Query 1 is a singular element search across the entire database. Query 2 involves searching for some region that contains our point, and utilizing the data associated with the region to search for stars. This query can be approached two ways: computing the region ID given our point as application logic and searching with this ID, or by searching with the bounds fields associated with each region. Query 3 involves searching for all stars that share the same `TYC1` field. Neo4J has the additional option of searching for the associated *Region* node and walking each edge to the corresponding *Star* node. Queries 4 and 5 are similar to 2 and 3 with an additional brightness filter.

Each general query was translated into 22 different queries for specific stars or locations. When referencing the time to execute query 1 or 2, it is implied that this is the time to execute any of one the specific queries for query set 1 or 2 on average.

B. Cassandra

The multi-node Cassandra clusters were configured with a *Simple Snitch*, a replication factor = 3, and `num_tokens` = 256. No other changes were made to the default configuration. A snitch informs Cassandra about network topology for request efficiency, and the *Simple Snitch* is recommended for single-datacenter deployments. Replication factor describes how many copies of the data exist, and `num_tokens` determines how much of the data each node gets. Given that each node has identical hardware, the `num_tokens` field indicates that all nodes get the same amount of data.

All queries were operated serially and timed within the same session. Cassandra does not allow subqueries (unlike SQL), so the time to execute the first query, feed this into the result of the second, and execute this query was recorded as a single run.

For the first query in subsection IV-A, we are given the exact `TYC1-TYC2-TYC3` ID and are told to search for all attributes associated with this star. Given that our primary key involves this ID, all Cassandra has to do is run these numbers through the same hash function used to partition the data, issue the query, and return the results.

The second query gives us some position instead of the `TYC` ID. Our first approach (denoted as Query 2A) is to treat our column family as a relational table. This involves searching for each "row" in our column family, and returning the `InRegion` field if Equation 1 holds:

$$(\texttt{RAmin} < \alpha < \texttt{RAmax}) \wedge (\texttt{DEmin} < \delta < \texttt{DEmax}) \quad (1)$$

From here, we search for all stars contained in the resultant of the previous query and return the result.

An alternative that avoids iterating through each primary key is to determine `TYC1` before asking Cassandra. This is denoted as (Query 2B). Query 2B involves iterating through a copy of the region data on the querying node, and checking for the region where Equation 1 holds. From here, we perform the same secondary step as Query 2A. A question of interest here is if this exchange for space results in a noticeable decrease in time.

Cassandra's third query takes advantage of our definition of near. Here, we can just search for all stars sharing the same `TYC1`. Queries 4 and 5 involve applying the naked-eye visibility filter to queries 2 and 3. Another question of interest is how a `BTmag` index affects the speed of our query.

C. Neo4J

The multi-node Neo4J clusters were configured as *Causal*, and all queries were performed on the leader node. No other changes were made to the default configuration. All queries were executed serially, operating within the same session.

The first query in subsection IV-A is a search across the entire node list for a singular element given the `TYC` ID. This was tested with and without an index on `TYC1`, to determine how large the speedup is.

For the second query, we are given some position instead of the star ID itself. The first approach, similar to Cassandra's 2A query, starts by searching for regions such that Equation 1 holds. With the region known, we then traverse each connected `WITHIN` edge and return the properties of each *Star* node.

In line with Cassandra's 2B query, we also try to see if the same space exchange results in query speedup. Neo4J's 2B query involves computing the `TYC1` ID in the same manner as Cassandra's query, but then traversing the edges of the region node and returning the properties of each *Star* node once found. A question of interest here is how a `TYC1` index affects Neo4J's 2B query.

In the third query, we are tasked with finding the properties of stars near some star defined by its `TYC` ID. The first approach (Query 3A) involves a similar query to Cassandra: search for all stars sharing the same `TYC1` ID. This is to be analyzed with and without indexes on `TYC1`.

The second approach (Query 3B) involves searching modeling the query in terms of relationships. First, we search some star given the `TYC` ID. Next, we traverse the connected `WITHIN` edge to find our region. Finally, we traverse each `WITHIN` edge connected to our region and return the properties of each star. Again, this was tested with and without a `TYC1` index.

Queries 4 and 5 are variants of queries 2 and 3 with the naked-eye visibility filter. Similar to the Cassandra queries, a question of interest is how a `BTmag` index affects the query time.

V. Results

A. Overall, No Index, 3 Nodes

In Figure 4, all queries are plotted against time for both Neo4J and Cassandra. Each cluster consists of 3 nodes, and no indexes are applied to any column.

Neo4J performs better with queries given some position (α, δ), while Cassandra performs better with queries given the `TYC` ID. Given that the Cassandra families are not indexed by their Equatorial position columns, it makes sense that Neo4J would shine here. For queries 2A and 4A, Neo4J performs an average of 0.88s faster than Cassandra. For queries 1, 3 (3A for Neo4J), and 5 (5A for Neo4J), Cassandra performs an average of 1.27s faster. Cassandra query 3 takes about the same time to run as Neo4J query 3A, at a difference of $\sim$7.9ms.

Neo4J's significant change in running time from query 3A to 5A is interesting, as the only change made is applying a brightness filter. Running both query sets through Neo4J's query planner reveals that query 3A is *compiled* and 5A is *slotted*. A compiled Neo4J query groups operators in the execution plan to optimize performance and memory usage, while a slotted query only optimizes how records are fed into each iterator. It appears that the inclusion of this brightness filter prevents query 5A from being compiled and optimizing performance. Running several queries from query set 1 through the query planner shows that this is also compiled, but is still slower than query set 3A. The main difference here is the inclusion of two more filters (`TYC2` and `TYC3`), which might explain this difference in time.

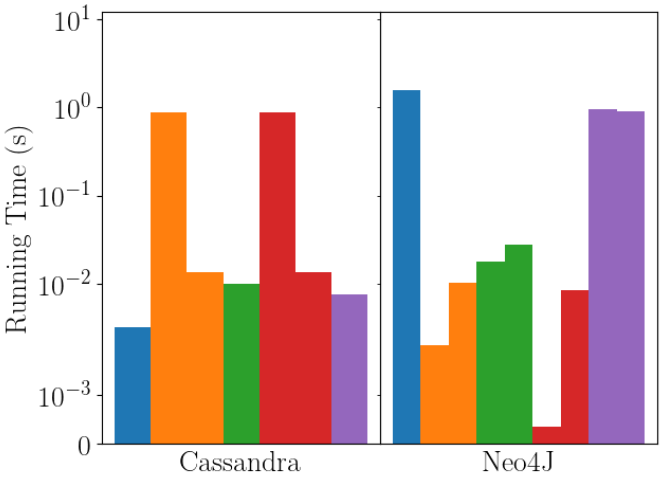

Fig. 4: Running times of different queries for a 3-node Neo4J cluster and a 3-node Cassandra cluster. Each point represents the average of 15 runs. The blue bars represent query 1, the orange represent queries 2A and 2B, the green represent queries 3 (Cassandra), 3A, and 3B (Neo4J), the red represents queries 4A and 4B, and the purple represent queries 5 (Cassandra), 5A, and 5B (Neo4J).

For both Cassandra and Neo4J, queries 2A, 4A, and 2B, 4B differ in their approach to determine the region our star is in. Reading a file containing the region data and determining our `TYC1` in application logic instead of performing the exhaustive search on our *Region* family does yield a time decrease of 0.87s. For the non-indexed Neo4J, this approach has no significant effect on performance (average of 8.31ms increase). Given that no mechanisms are in place for Neo4J to take advantage of this, it makes sense that no speed up is shown.

In Neo4J queries 3B and 5B, we vary our query to include the *Region* node as opposed to their A variants. Both require a full search of the entire *Star* node list, but the B queries require an additional traversal to the matching *Region* node and additional hops to each *Star* node. These additional steps suggest that the A queries should be faster, but the results are mixed. Our results show that query 3A takes the same amount of time to run as query 3B (9.68ms difference), while query 5B is actually $\sim$69ms faster than 5A.

In terms of query time distribution, the average deviation of all Cassandra queries is 20.4ms, while the average Neo4J

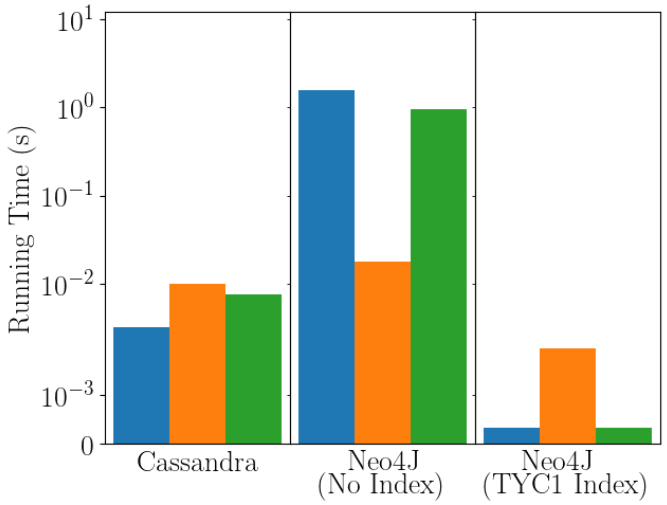

Fig. 5: Running times of queries 1, 3, and 5 for a 3-node Cassandra cluster, a 3-node Neo4J cluster without an index on `TYC1`, and another 3-node Neo4J cluster with the specified index. Each point represents the average of 15 runs. The blue bars represents query 1, the orange represents query 3 (3A for Neo4J), and the green represents query 5 (5A for Neo4J).

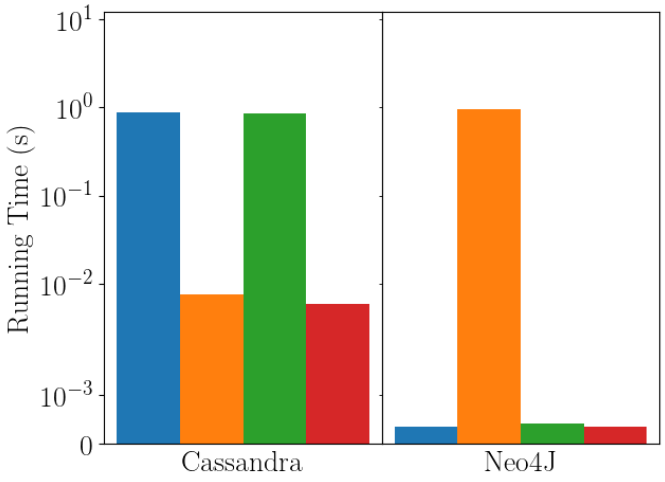

Fig. 6: Running times of queries 4 and 5 for a 3-node Cassandra cluster with and without an index on `BTmag`, and a 3-node Neo4J cluster with and without an index on `BTmag`. Each point represents the average of 15 runs. The blue bars represent queries 4A without the `BTmag` index, the orange represents query 5 (5A for Neo4J) without the `BTmag` index, the green represents queries 4A with the index, and the red represents query 5 (5A for Neo4J) with the index.

deviation is 3.36s. Neo4J queries can take up to 2 minutes to return a single result, and under a second for others. On average, our Cassandra cluster performs all queries faster than our non-indexed Neo4J cluster by 0.13s.

B. *Queries {1, 3, 5}, TYC Index, 3 Nodes*

In Figure 5, queries 1, 3 (3A for Neo4J), and 5 (5A for Neo4J) are plotted for a 3-node Cassandra cluster, a 3-node Neo4J cluster without any indexes, and a 3-node Neo4J cluster with an index on `TYC1`. All of three of these queries involve searching for some star based on the `TYC1` field. The `TYC1` index cannot be applied to our Cassandra cluster because this is the primary key for *Stars* column family.

On average, the `TYC1` indexed Neo4J cluster performs about the same as our current Cassandra cluster (∼6ms difference). Cassandra performs 0.85s faster than a non-indexed Neo4J cluster. The average difference in time between the non-indexed Neo4J cluster and the `TYC1` indexed cluster is 0.85s. The inefficiencies seen in subsection V-A for Neo4J queries 1 and 5A are reduced with the inclusion of an index on the filtered field.

Relative to queries 1 and 5, query 3 was already fast (average of 1.27s vs. ∼18ms). After applying the index, query 3 becomes the slowest of the three, although the difference is fairly small (1.6ms). Query 3B is also more distributed than queries 1 and 5A (0.96ms vs. 0.06ms), suggesting that this behaviour is consistent.

A slight performance boost may be gained for both query 1s of Neo4J if a size restriction filter were placed (i.e. `LIMIT`). For the non-indexed case, this query is of complexity $T(n) = \Theta(n)$ and could be lowered to $O(n)$ if we restrict our resultant to one element.

C. *Queries {4, 5}, BTmag Index, 3 Nodes*

In Figure 6, queries 4A and 5 (5A for Neo4J) are plotted for a Cassandra and Neo4J 3-node cluster with and without an index on `BTmag`. Both of these queries involve searching for nearby stars and applying a brightness restriction filter.

Neo4J with the `BTmag` index performs queries 4 and 5 ∼0.29s faster than Cassandra with the same index. Neo4J query 5 descends from 0.96s to 0.33ms when the `BTmag` index is applied, but there appears to be no significant effect for query 4A (difference of 8.31ms). Given that query 4A was already fast compared to 5A, it follows that any changes in time should be minimal.

Cassandra with the `BTmag` index executes queries 4 and 5 with a very slight increase in time (13.0ms) compared to the Cassandra cluster without the index. Query 4A runs 24.3ms faster with the index, but query 5 runs for the same amount of time (difference of 1.76ms). The `BTmag` index is marginally effective in reducing the query time, but the largest optimization for Cassandra query 4 is reformatting the query for use with the primary key (from 4A to 4B).

D. *Query 1, No Index, 1–3 Nodes*

In Figure 7, query 1 is plotted for Cassandra and Neo4J clusters of varying node size (1 to 3 nodes). For brevity, only query 1 was plotted to determine if how the architecture affects query time.

The running time of Neo4J query 1 is directly proportional to the number of nodes in the cluster. From 1 node to 2, Neo4J experiences an average increase of 1.38s in query time. From 2 nodes to 3, Neo4J experiences an increase of 0.19s in query

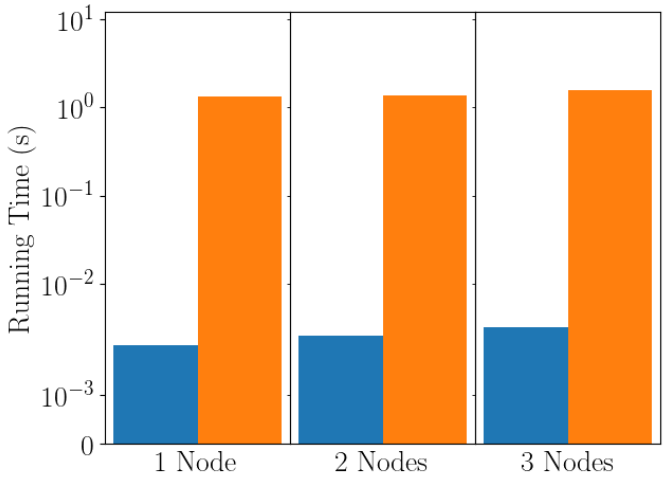

Fig. 7: Running times of query 1 for 1–3 node Cassandra and Neo4J clusters, without any indexes applied. The blue bars represent the Cassandra queries, while the orange represent the Neo4J queries. Each point represents the average of 15 runs.

time. More nodes appear to detrimental to performance here, suggesting that Neo4J may not scale out well.

On the other hand, the number of nodes in our cluster does not seem to have an effect on the running time of Cassandra query 1. The running times here are distributed as: $2.6\text{ms} \pm 0.5\text{ms}$. A more significant result might be seen if a slower query were analyzed, but distributed column stores are known for their ability to scale out. Cassandra should provide consistent and reliable performance as more nodes are added.

VI. Conclusion

A. Future Work

Future work for this project includes running more query variants, modifying each DBMS architecture more, and analyzing the execution plan more.

A large assumption that helped simplify our queries was our definition of *near*. Sometimes, near might involve the angular separation between two stars that span over several GSC regions. As our current data models stand, this would involve iterating through each star family / node set twice at worst. The optimizations used here would not apply to this problem, and we would have to use an entirely different data model.

Clusters of 1, 2, and 3 nodes at the same datacenter were used for each DBMS due to cost limitations. With our current results, it seems that Neo4J should not scale out well and Cassandra should accept more nodes without a problem. Using more nodes across different regions would determine how well Neo4J and Cassandra really scale out. Queries were also executed serially, and more interesting results may have been seen if multiple were run in parallel. Both Neo4J and Cassandra provide availability, and characterizing this would also allow us to test each DBMS's distributed nature.

Query plans were briefly looked at for the non-indexed Neo4J cluster running queries 1, 3A, and 5A. Cassandra does provide an execution planner, and could be compared against Neo4J. Query plans would provide more concrete insight into why certain queries run faster than others. Delving deeper, more explicit analysis could be performed by characterizing the role of the cache for each query (cache misses or hits) or tracking the messages sent to each node.

B. Conclusion

Finding nearby stars is a common stellar query: in star trackers, identifying poorly cataloged images, celestial navigation... Unfortunately there are hundreds of millions recorded stars, and the problem becomes finding nearby stars in an efficient manner. Stars are recorded as 2D points on the celestial sphere in an astronomical catalog known as a star catalog. Using the Tycho-2 star catalog allowed us to reduce this spatial query to finding stars that are contained within a specific region on the sphere.

From here, we divided our approach in two: the Cassandra solution with *Region* and *Star* column families, and the Neo4J solution with *Region* nodes, *Star* nodes, and `WITHIN` relationships. The former is known for efficient, but restrictive queries. The latter is known for its efficient relational queries, but also its inability to partition work like a column family.

Our results show that Cassandra runs faster when Neo4J is not indexed, but Neo4J is able to run just as efficiently (in some cases, more) when the correct indexes are applied. Cassandra queries benefit from taking the time to determine the primary key before asking Cassandra (time-space tradeoff), instead of performing an exhaustive search on Cassandra itself. Neo4J does not experiment this same performance boost. Finally, Cassandra appears to perform the same under a change in cluster size while Neo4J does not.

For consistent performance, Cassandra should be selected. For a more natural abstraction with a higher upper bound on performance, Neo4J should be selected.

References

[1] DATASTAX. Architecture in brief, 2018.
[2] DATASTAX. How are consistent read and write operations handled?, 2018.
[3] DATASTAX. Intializing a multiple node cluster, 2018.
[4] HOG, E., ET AL. Guide to the tycho-2 catalogue.
[5] NEO4J. Recap: Intro to graph datbases, 2011.
[6] NEO4J. Create a new causal cluster, 2018.
[7] NEO4J. Indexes, 2018.
[8] RUND, B. The good, the bad, and the hype about graph databases for mdm, 03 2017.
[9] SADALAGE, P. Nosql databases: An overview, 2014.