INDIAN INSTITUTE OF TECHNOLOGY, DELHI

B.TECH PROJECT THESIS

Refining Index Storage in NoSQL Graph Databases

Author:
Mayank Rajoria
Prakhar Gupta
Prakhar Agrawal

Supervisor:
Dr. Srikanta J. Bedathur

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Technology

in the

DAIR
Department of Computer Science and Engineering

May 18, 2018

INDIAN INSTITUTE OF TECHNOLOGY, DELHI

Abstract

Department of Computer Science and Engineering

Bachelor of Technology

Refining Index Storage in NoSQL Graph Databases

by Mayank Rajoria
Prakhar Gupta
Prakhar Agrawal

Graph databases such as Neo4j, JanusGraph/Titan etc. are increasingly becoming popular due to the flexible data model they provide. However, writing optimal queries on these databases using the dataflow query language, Gremlin, remains an interesting problem. In this project, we explore ways to optimize these Gremlin queries by exploiting the flexibility –viz., schema – of the graph database.

Specifically, we try to achieve this by storing the index structure as a part of the graph database itself (thus removing dependency on third party application like the elastic search or solr). We explore this approach towards the following four-fold objectives: (a) reduce the moving components from a graph database, (b) remove the application layer overheads in terms of using external applications like solr (c) offer flexibility in terms of index structures one can design and play with and (d) explore other unexplored ways of traversal e.g. traveling through index to data and then going up some other index to get cumulative number etc.

Contents

Chapter 1

Introduction

1.1 Motivation

NoSQL and Graph databases are alternatives to the already existing SQL approaches used in web and cloud applications. They have recently gained a wide spread acceptance, both in industry and academia. This has been due the their ability to handle an increasing scale of operations and the fact that they allow data to be stored in a way it is modeled.

These databases do support indexes but require a separate index backend to create and maintain this index. This not only adds more moving components to the database but also hampers the flexibility to use the set of index structures offered by the application alone. This separation of index, from the data, at an application level causes inefficiencies in its usage. This is due to the inter-application overheads involved in the communictaion between the graph database store and the index backend. Hence, exploiting the flexible schema of graph databases, we have devised a way to store the index along with the data, and reduce these additional overheads.

1.2 Approach

Schema flexibility in Graph databases allows us to "rewire" a graph in the database by introducing new vertices and edges/relationships between these and original vertices. We create these new, or "ghost" edges and vertices, based on their utility in making the queries faster. These vertices and edges should ideally be invisible to the end-user and play the role of an *index* in these graph databases.

These will allow us to perform faster traversal in the graph and hence would speed up the queries we wish to run on the graph. We construct these elements and put them as a layer in the graph database itself, as "ghost" nodes and edges. Clearly, this kind of rewiring is going to bloat up the size of the graph, hence we must chose the vertices and edges to be added carefully. This decision is should be made on the basis of a set of static query workloads given initially. Currently we do not have a metric to base our decision regarding the choice of the index but use intuition for all

further experiments stated in the paper.

Considering this flexibility of the graph, a natural way to introduce powerful indexing capabilities within the graph is to store the index itself as a layer in the graph. We utilize the fact that most of the traditional indexes can be represented as trees (for eg B-Tree), which are essentially graphs. For instance, it is possible to store a multi-dimensional index structure such as R-Tree simply in the graph database starting from linking vertices which have overlapping bounding boxes, and going on to construct multiple layers of this relationship in the same manner as a traditional R-Tree.

Finally, to understand the impact of these methods, we need to run test on a benchmark. We choose the LDBC's [1] (Linked Data Benchmark Council) Business Intelligence Workload which consists a various complex structured queries for analyzing the behaviour of various parts of a social network. These workloads help us highlight the effect of our methods on the performance.

1.3 Choice of Benchmark

LDBC (Linked Data Benchmark Council) [1] is an organization that develops benchmarks that allow various Graph Database management systems to be tested for their capabilities and their performance. We use the Social Network benchmark developed by them. This benchmark has vertices and edges in the structure of a typical social network i.e. users with details about them, people they know, their posts and likes and comments on posts. It also has a set of both interactive and business intelligence queries that operate on this data. Also it has a data generator package which can create social network databases of varied sizes.

Social Networks are one of the best examples of applications of Graph databases and therefore they form a great choice for queries which can involve a given number of hops (example a 2 hop from a person would give the people known to the people he knows). The indexes we are creating can thus be tested for efficiency by cutting down on the number of such hops. So social networks are a good choice for benchmarking.

LDBC SNB workload has some really good properties that allows us to create synthetic data which closely mimics the data from real social networks. It enforces on the network structure the power law distribution in terms of various properties as well as the short diameters of the graph that are generally common in social networks. It is also generates property values which are realistic in nature and generated from correlated value distributions of these properties. Temporal correlations and presence of flash mobs is also something that makes this a really valuable benchmark for any system that is to work in a real world scenario.

For our testing, we only deal with the Business Intelligence queries of the Social Network benchmark. This consists of complex structured query workloads. Most of these queries require us to find and group vertices that satisfy certain given constraints. Since we are using the Gremlin query language, we had to convert these queries into gremlin and run them on the database loaded by us. These conversions can be found in our java project codebase [10] .

Chapter 2

Setting things up

This chapter describes how to set up the development environment. It describes all the preliminary tasks that are to be done before we actually even start thinking anything about the in-graph index data-structures. Issues ranging from loading the dataset to the conversion of the English query description to gremlin are described and the query. We further briefly give details of our java codebase that comes in very handy while dealing with this process and streamlines the process of obtaining query timings (and other statistics) into running a fixed set of predefined classes. We also mention the problems faced during each of the tasks so as to give the read an idea of how to deal with the same, should he/she come across similar problems. This chapter is divided into the various sections for clarity. We first describe the basis for the choice of the graph database to perform experiments, then briefly talk about the query language used in the project, the third section is to help the interested reader on how to get the dataset loaded into the graph database (as this can be tricky in several cases). We finally talk about how to translate the English descriptions of the query from the LDBC benchmark into their gremlin equivalents.

2.1 Choice of the Graph Database

2.1.1 Neo4j

Neo4j being one of the most popular Graph DB, was the first choice for testing the suggested changes. Also Neo4j was initially chosen by us due to the fact that it supports both Cypher and Gremlin. Hence, we would be able to compare the performance of the Gremlin queries (both with and without our newly created Indexes) and the ones written in Cypher.

2.1.2 Janus Graph

JanusGraph is another popular Graph Database. It is an open source project and is under The Linux Foundation. It also has active contributions from organizations like Google, Amazon and IBM. JanusGraph does not have any of its own exclusive

query language. Instead, it utilizes the popular Apache TinkerPop which is a one stop solution for most graph processing applications.

The Gremlin DSL is the prime choice for the project since it is more closer written to the imperative fashion than the sql like declarative fashion. This would allow us to manually write traversals using our modifications without directly messing with the code of the core database. Though we started with Neo4J, its quickly out-dating gremlin plugins and downgrading support for Gremlin DSL (which was an important part to begin our project) implied that we shift to the other popular option of Janusgraph. Thus, after encountering some integration difficulties between Neo4j and Gremlin, we decided to move to JanusGraph.

2.2 Query Language: Interacting with a graph database

The two chief candidates for the choice of the query language were Cypher and Gremlin. Cypher is the query language used in Neo4j, the most popular Graph Database. Being declarative, it is closer to SQL, thus making it easier to use. Moreover, it is more advanced than most Graph DB query languages by virtue of having a query planner to help in optimizing the evaluation of the queries.

Gremlin is an open source, vendor agnostic, graph computing frame work. It allows for both, declarative and imperative style queries on graphs. Unlike Cypher, the query planner that Gremlin has is very basic in its functionality. So it is the responsibility of the user to write the best possible way of traversal of the graph.

From the above description, it seems that Cypher is a hands down winner among the two due to ease of use. Still, we decided to go ahead with Gremlin, owing primarily to the fact that Gremlin supports querying all major Graph DBs but Cypher only supports Neo4j. We must also note that Gremlin provides better flexibility to the user (due to the possibility of writing imperative style queries) in terms of deciding the evaluation plan rather than leaving it up to the query planner. This was a key part of the project since it involved modifying the graph traversals to use different kinds of indexes available. We needed a highly granular control over the traversal. This reason overruled all others and we chose to use the gremlin query language[5].

2.3 Loading the dataset

Using the LDBC datagen [3] we generated the dataset with standard scale factor (SF=1) which created about 1GB of dataset in csv files. We used the default berkeley DB backend to start with, which we eventually changed to the Cassandra backend

where we observed an improvement in terms of dataset loading times. Since being able to change and load the dataset was crucial to be able to conduct the experiments, we ended up sticking to the Cassandra backend.

The Janusgraph configuration was set to turn on batch loading with the configuration 'storage.batch-loading=true' and 'schema.default = none' (Note: This turns off the transactions in the graph database and so concurrent writes might cause inconsistencies). Turning off schema auto creation is required as mentioned in the JanusGraph documentation to turn on the batch loading. We use the jars given along with the project code in the repository and then insert the data points using a simple for loop. We flush the memory to enforce that the DB writes to the disk after every 30,000 data points insertion and allow the process to do a garbage collection manually. Without this we faced several crashes and severe slow downs in many cases.
The general load times that we observed for this Scale Factor=1 dataset (which we also call the 1GB dataset) is as shown in the following table. The size of the bloated dataset was around 6.7GB.

Machine	**Dataset Size**	**Loading Time**
Baadal (32 GB, 8 cores)	120 MB	~40 mins
Aryabhata (128 GB, 32 cores)	1GB	~2 hours

One important thing to note here is that the required elastic search indexes for the experiments were also created before loading data during the creation of the schema (we had to do this manually since schema.default=None was set in the configuration). This was done because the re-indexing step required when creating a new index on a pre-loaded dataset always timed out even with substantially large time out limits (in order of days).

Now that we have our dataset in place, we can start looking at the query workloads and their implication for our proposed idea. This is exactly what the next section offers.

2.4 Working with queries in LDBC

LDBC's SNB Business Intelligence workload is a set of complex structured queries analyzing the behaviour of entities on a social network. These are parameterized with certain substitution parameters (eg. Get all persons below age A, where A is an parameter), to be able to obtain the same information in various parts of the social network. These substitution parameters can be secured from the LDBC data

generator that also generates parameters. The queries are given in the LDBC SNB specification [2] are described in simple English with expected inputs and the outputs (along with the sort order and retrieval). Naturally the first step, before testing the proposed in graph index data structure, boils down to translating these queries into the Gremlin DSL. This will eventually help us compare the raw query time (or time of the best raw query with the SOTA existing index structure) to the performance that our index structure attains.

For flexibility, ease of use and better reproducibility we have maintained a Java codebase [10] that contains the gremlin equivalent of almost all the LDBC BI queries (without the ordering steps). These query classes are contained in the Queries module of the project. We have tried to make the interface generic for each query to allow for easy computation of any timing statistics that the query might require. More details can be found on the wiki and the readme of the code repository.

One sample query from the LDBC SNB BI and its corresponding gremlin equivalent (without the order step) is shown below to describe the general conversion. The queries keep getting more and more complex as we long further in the DI workload and the reader can also look at the gremlin documentation to understand the various steps used in the more complicated gremlin query.

> Given a date, find all Messages created before that date. Group them by a 3-level grouping:
>
> 1. by year of creation
> 2. for each year, group into Message types: is Comment or not
> 3. for each year-type group, split into four groups based on length of their content
>
> - 0 <= length < 40 (short)
> - 40 <= length < 80 (one liner)
> - 80 <= length < 160 (tweet)
> - 160 <= length (long)

```
g.V().hasLabel("post", "comment")
  .has("po_creationDate", P.lt(dateVar))
  .group()
  .by{ it.value("po_creationDate").getYear()+1900 }
  .by(group().by(T.label).by(group().by{
    int len = it.value("length");
    if (len < 40) {
        return "short";
```

```
} else if (len < 80) {
    return "one liner";
} else if (len < 160) {
    return "tweet";
} else {
    return "long";
}}));
```

Chapter 3

Index Creation: In Graph Index Structure

Now that we are all set with the basic development environment we can start thinking about the structure of our in Graph index data structure and how to create it. Instead of innovating more on the structure side we decided to implement the already popularised index structures inside the graph database and observe the performance changes followed by error analysis over it. So creating index structure inside the graph DB is the next natural step to follow. This chapter gives a detailed explanation of the same.

3.1 Choice of Index

Essentially, most of the popularly used index structures used are either graphs or can be modeled as graphs. For example B-Tree and R-Tree are trees (a type of graphs. Hash indexes too can be seen as a trees with a single internal node having a leaf-child corresponding to each possible value of the hash function even if they are not graphs in the true sense of the word. This means that in a graph database it is quite natural to store them within the graph rather than keeping them as separate independent structures.

The choice of the index structure to be included in the graph is mainly based on maximising the its utility with a minimum increase in the size of the graph. Therefore, B-Trees and B+ Trees seems to be a reasonable choice. They can have varied depths based on the size of the data being indexed and can be used for answering range queries unlike hash indexes. For example, consider a query requiring us to return all people with age between 20 and 50. If we use a B-Tree index, we can traverse the index tree in a BFS manner keeping only the relevant nodes for further traversal and discarding the others until we finally reach the records. While for Hash indexes, this is not possible. We will have to iterate over all the ages from 20 to 50, and find the records corresponding to each age individually. This though seemingly not so bad in performance however if there are many values for ages with no records then the hash indexes perform really bad. Moreover, consider the extended version of the problem where instead of age, we only have access to the Date of Birth of the people

(i.e. finer granularity on the record values) and the index is present on this value. In this case the the hash function performance degrades considerably as the iteration is now required over all days lying in the span of 30 years whereas the performance of the B-Tree index remains the same.

3.2 Construction of B-Tree/B+ Tree Index

The scope of our work involves creating a framework on top the gremlin query language to create and use B-Tree indexes. However writing B-Tree directly in Gremlin would be really slow to execute. Since we do not need to handle update and modifications in the data, one alternative is to implement the B-tree in some other fast programming language, traverse the tree to determine what all new vertices and edges need to be created in the graph DB, express it in a gremlin script and execute the script. This way we can introduce optimizations for minimizing the number of fetches of vertices and edges in gremlin.

C++ was chosen as the language for creating the index and traversing it. Since the B-Tree index has an edge to each data vertex on which the index has been created, the generated gremlin script must have twice the number of lines as the data vertices (one for fetching the data vertex and another for connecting it to the index) in addition to the creation of index vertices and index_vertex-index_vertex edges (depending on the maximum branching factor chosen for the B-Tree). Thus, the size of the script would be really large for even moderate sized data. For example, the index generation script on a 13.8 MB file containing the 131,566 posts of a social network dataset (generated using LDBC) had 266,039 lines when the maximum branching factor was set to 201. The execution of this script was tried multiple times with inclusion of improvements like use of composite indexes while fetching data vertices, adding intermittent commits and freeing of redundant program variables. However it never completed in 2 days time (tested on Baadal Virtual machine with 32 GB ram, 8 cores).

The need to introduce improvements like reducing the number of fetches of vertices and using gremlin internal optimizations was felt. One way for this was to run the process of index creation in loops rather than having individual query for each data item (so that the gremlin interpreter can contribute something to the improvements). Since each data item is unique, the only good way to implement this was to have all the edge creations (*INDEX_EDGES* and *INDEX_DATA_EDGES*) needed in a csv file having the id of one vertex against that of another. The creation of these edge links can thus be run in a loop. Another factor introduced through this was sorting the edges creation csv files on the ids of the column having fewer distinct values. This helped us to prevent the extra fetching of the same vertex again and again. When run on Janus Graph with batch loading on, the above index creation successfully completed in ~3 minutes. When tested on a file with 1,003,605 posts,

the execution completed in ~17 minutes on aryabhata (128 GB ram, 32 cores).

Machine	**No. of Vertices**	**Loading Time** (gremlin queries)	**Loading Time** (csv files)
Baadal (32 GB, 8 cores)	131,566	> 2 days	~3 min
Aryabhata (128 GB, 32 cores)	1,003,605	-	~17 min

Other index tree structures can be created in a similar way. We have created B+ Tree by making small changes to the code which was used to create the B-Tree. The major change is addition of chain-links (*NEXT_LEAF_EDGE*) between leafs which offer a possibility of speeding up the range queries.

3.3 Super Index

To be able to use an index tree for speeding queries, we must first be able to quickly access its root vertex. This creates the need of having an index on the root of all index trees. We call this structure the super index.

Our super-index is just a single vertex which has outgoing vertices to all index roots. This edge has the name of the index, property of indexing and type of index (B-Tree/B+ Tree). Keeping the metadata of the index on the edge is useful as we do not need to go the index to know whether we would require to use that index for a particular query. Our program creates the super-index while adding the *Management* data for index creation. The edges from super-index to index are created while creating the index tree.

With the super-index, our problem of being able to quickly reach all index roots reduces to just accessing the super-index vertex. One option is to store the vertex-id of this root vertex. However we cannot store this data on the graphdb. So, we should create the super-index root with a special id, like 1 (though custom vertex ids might not be allowed for all Graph DBs). The other way is to give an attribute to this vertex on which a composite index is present. We have used this method. The attribute is *index_id* and the value is -1.

Chapter 4

In Graph Index Structure: Experiments & Analysis

In this chapter we start by describing how to modify existing gremlin queries to use the index structure that was created in the previous chapters. We go in a sequential manner describing small modifications & doing the associated error analysis - slowly building towards the final model that we have in our code repository currently. This will help user appreciate why a particular modification was needed and also help him/her reason about any further optimization that can be brought into the query.

4.1 Modifying the Gremlin

Let us take the example of query from chapter 2 (the one given as a sample).

```
g.V().hasLabel("post", "comment")
  .has("po_creationDate", P.lt(dateVar))
  .group()
  .by{ it.value("po_creationDate").getYear()+1900 }
  .by(group().by(T.label).by(group().by{
    int len = it.value("length");
    if (len < 40) {
        return "short";
    } else if (len < 80) {
        return "one liner";
    } else if (len < 160) {
        return "tweet";
    } else {
        return "long";
    }}));
```

To be able to use our in graph index structure in processing of this query the most obvious way is that of retrieving the vertices based on the index structure first and then passing them to the graph traversal for further processing down the line. Let us assume that we have a function of 'searchRange' already at our disposal that using

our index structure can retrieve all the vertices based on the attribute conditions. Then using this our modified query will look something like below:

```
vertices = (new IndexRangeQuery())
            .searchRange(g,
                  dateFormat.parse(initDateVar),
                  dateFormat.parse(dateVar),
                  indexName);

g.V(vertices).hasLabel("post","comment")
  .group()
  .by{ it.value("po_creationDate").getYear()+1900 }
  .by(group().by(T.label).by(group().by{
    int len = it.value("length");
    if (len < 40) {
        return "short";
    } else if (len < 80) {
        return "one liner";
    } else if (len < 160) {
        return "tweet";
    } else {
        return "long";
    }}));
```

The 'searchRange' function can be written in Java, using the gremlin-java support, that enables us to write a simplistic B+Tree traversal code in Java to traverse the above created index. Reader can look at the code repository to get a deeper understanding of how this was exactly done. The observed performance comparison for this kind of query vs the one using the external indexing application (elastic search in this case) is shown below:

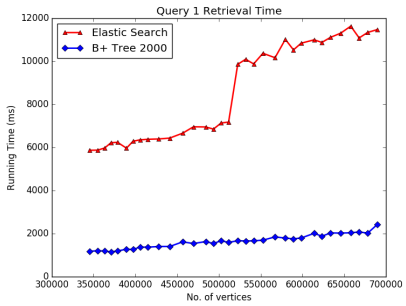

FIGURE 4.1: Vertex Retrieval Time

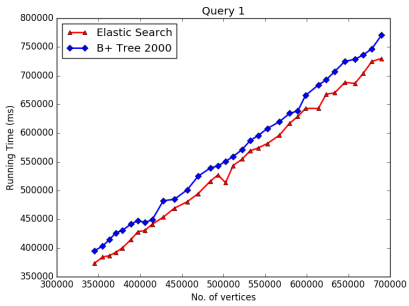

FIGURE 4.2: Query Running Time

From the graph we can see that though the in graph index structure out performs the elastic search index in terms of just the data retrieval (the primary task of the index), the in graph data index has somehow caused the overall query to slow down.

4.1.1 Hyperparameter: Fan-out factor

The in graph indexed used to produce the above results is a B+ Tree with a fan out factor of 2000 (i.e. maximum no. of children of a node is 2000). This is a hyper-parameter that can be searched over to get the most optimal value. For our experiments we did not perform any hyper parameter search but instead chose an intuitive value after checking the query timings with a few values of the fan out factor.

4.2 Error analysis and optimizations

In the above section we observed a that the query utilizing the in graph index data structure was slower than the query utilizing the elastic search index. A prominent reason behind this could be that the in the query using the in graph index data structure, there is an explicit separation of the data retrieval step and data processing step. While on the other hand the query utilizing the elastic search index might derive benefits from pipelining of the two parts of the query. (Note: Pipelining, here, refers to the general definition of pipelining popular in the DBMS community). In order to confirm this hypothesis, a simple experiment would be to separate the query utilizing the elastic search index into two parts and obtain the query timings. Shown below is the query timing graph for such a case.

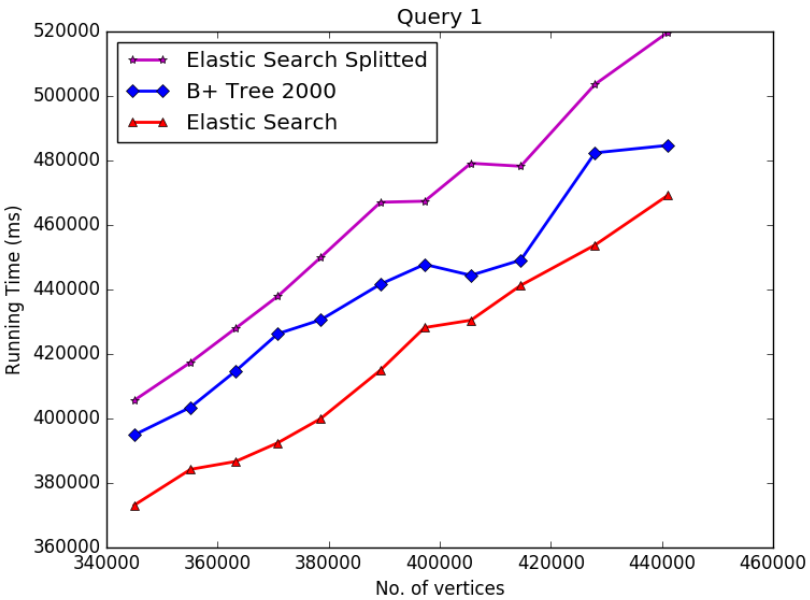

FIGURE 4.3: Query Running Time

While we now have some idea on what might be going wrong, how this can be rectified is still to be resolved. As an attempt to resolve this we try to combine the two parts of the query utilizing the in graph index structure into one. This can be done by

traversing the index to retrieve information in a single gremlin query rather than in Java-like code and then calling the data processing steps on this traversal. We used .repeat() and .until() steps to traverse over our in graph index data structure. The traversal that we came up with to traverse through the B+ Tree in a single gremlin query is shown below:

```
g.V().has("index_id", -1).out().has("name",indexName)     //index root
  .repeat(
    __.outE("INDEX_EDGE")
      .not(__.has("min", P.gte(dateVar)))
      .not(__.has("max", P.lte(initDateVar)))
      .inV()
  )
  .until(__.outE("INDEX_EDGE").count().is(0))
  .outE("INDEX_DATA_EDGE")
  .has("val", P.gte(initDateVar)).has("val", P.lte(dateVar)).inV()
```

Combining this with the data processing step from the sample query we have been using, we get a combined query as shown below:

```
g.V().has("index_id", -1).out().has("name",indexName)
  .repeat(
    __.outE("INDEX_EDGE")
      .not(__.has("min", P.gte(dateVar)))
      .not(__.has("max", P.lte(initDateVar)))
      .inV()
  )
  .until(__.outE("INDEX_EDGE").count().is(0))
  .outE("INDEX_DATA_EDGE")
  .has("val", P.gte(initDateVar)).has("val", P.lte(dateVar)).inV()
  .group()
  .by{ it.value("po_creationDate").getYear()+1900 }
  .by(group().by(T.label).by(group().by{
    int len = it.value("length");
    if (len < 40) {
        return "short";
    } else if (len < 80) {
        return "one liner";
    } else if (len < 160) {
        return "tweet";
    } else {
        return "long";
    }}));
```

The comparison of this query with the one utilizing elastic search is shown below.

The elastic search index is still out performing ours in terms in the full query time.

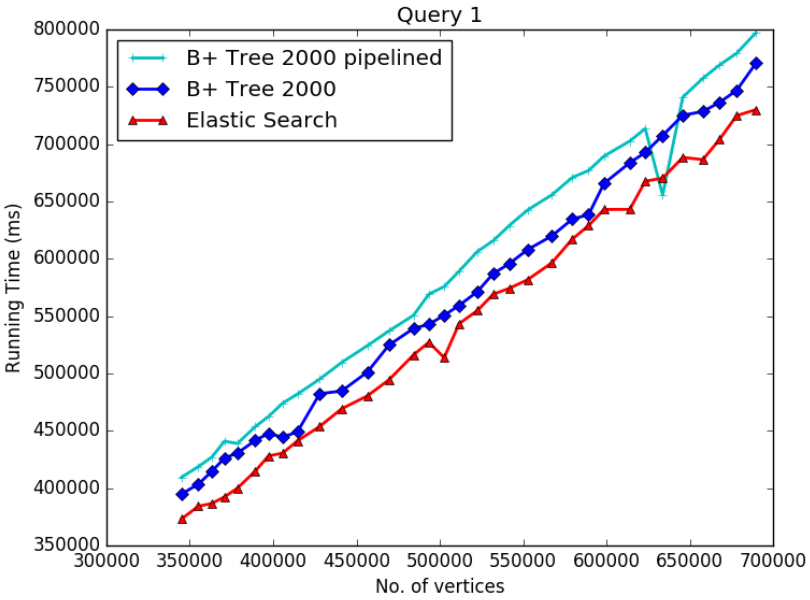

FIGURE 4.4: Query Running Time

Clearly we have not been able to come up with the optimal index traversal gremlin query. We experiment with a few things like if the order of the data retrieved is affecting the overall processing time but have not received any convincing results yet. One prime thing we came across was the bulk optimizations that are internal to JanusGraph. Interestingly putting an explicit barrier step after data retrieval does the greatest improvement to our timings.

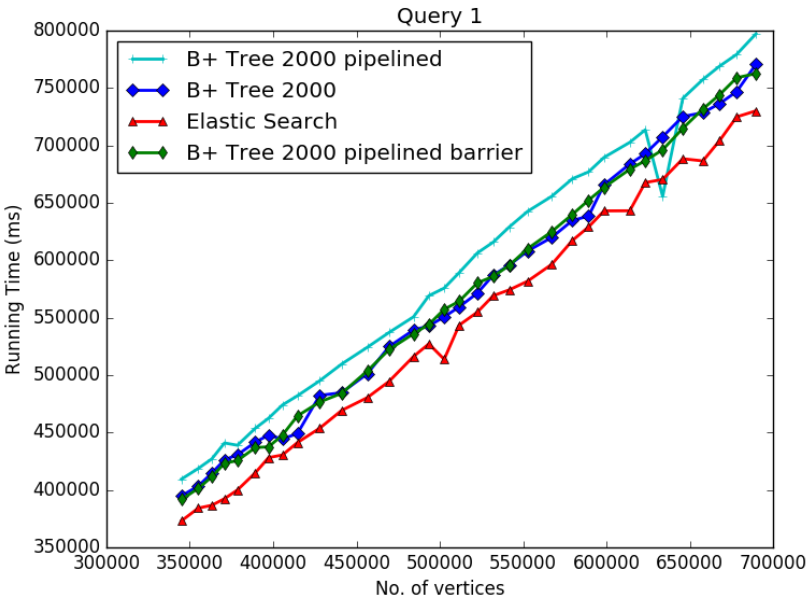

FIGURE 4.5: Query Running Time

In order to obtain greater insights on what was going wrong we started looking at other queries in the workload and surprisingly there were quite a few of them where the queries utilizing our index executed faster than the ones utilizing the elastic search index. This has been shown below:

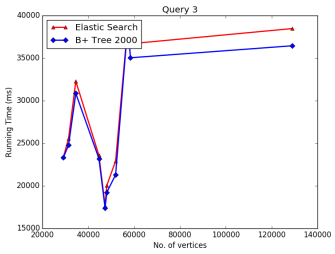

FIGURE 4.6

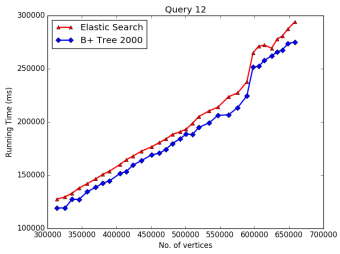

FIGURE 4.7

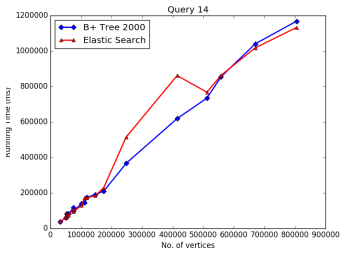

FIGURE 4.8

Why the performance in query 1 degrades is something that remains to the ascertained yet. How can we further improve the performance and make this much more faster than elastic search would be another interesting thing to explore and find out.

4.3 Results

The result sheet

The aforementioned Google Sheet [9] contains all the experimental results we have obtained. The sheets *[Final] Query Running Time* and *[Final] Vertex Retrieval Time* contain our final query timings when we ran them on Aryabhata on the 1GB dataset. We have the timings for queries 1, 3, 10, 12 and 14. These are the queries that used our B+ Tree index on *po_creationDate* property.

4.4 Brief description of the code repository

PerformanceTester

PerformanceTester is a class created to aid the running of queries to get reliable experimental data of run times. This class runs the queries in the Queries folder. These queries are expected to inherit from the *Query* class. The *PerformanceTester* takes the class name of query to run, the parameter file index in *substitution_parameter* folder, the number of lines to skip in the parameter file and the *conf* file to use to make connection to the JanusGraph DB.

Running the *PerformanceTester* runs the specified query using the specified parameter file. It generates a .txt file that has various metrics like the average, median and

standard deviation of the query run times and the index traversal times. This .txt file is generated in a *query_results* folder.

Using the Performance Tester class
Create the jar of the java module. The usage for running the jar is

java -cp Utilities-1.0-SNAPSHOT-jar-with-dependencies.jar main.PerformanceTester <queryClassName(only the endclassname)> <param_number> <skiplines> [<confFile>]

Example:
java -cp Utilities-1.0-SNAPSHOT-jar-with-dependencies.jar main.PerformanceTester BPIndexQuery1_1 1 1 local

The *query_results*, *substitution_parameter* and *conf* folder will need to be present along with the jar when it is run. Sample of these can be obtained from our git repository [10].

The times we obtained by running queries can be accessed at result spreadsheet[9]

Chapter 5

Query Rewrite Framework

So far, we have seen that we are dependent on including Index tree traversal in the query. To compete with Elastic Search indexes, we must be able to seamlessly use our indexes in general gremlin queries. This creates the need to be able to automatically rewrite a gremlin query with index tree code so that we have a true proof of concept of the applicability of our indexes.

Methodology
Our first task in the rewriting procedure is to identify part of the gremlin query to be rewritten. For this, we break to query into a list of *Traversal Steps* using the *GremlinGroovyScriptingEngine*. We then look for *HasStep* which lie immediately after some *GraphStep* which have all vertices in the Graph. The key attribute of this *HasStep* is identified. We then check whether there exists an index applicable for our given attribute. We thus need the metadata of the indexes. Here come handy the edges from the super-index to the index roots which have properies representing the metadata.

Just looking at the first HasStep is not much meaningful. We should consider all consecutive HasSteps after the GraphStep for applying the index. To achieve this, we keep looking at the key attribute in all HasSteps till we find one on which we can use our index. After this, in the next consecutive *.has()*, we only look for the presence of the same key attribute. All these HasStep collectively define a range of values within which the key attribute should be in the vertices of the result. We maitain 2 variables: $minVal$ and $maxVal$. Initiall these values are set to the extremes of the datatype of the key (so as to return all possible values i.e. full range). Every HasStep on the key shrinks this range. Finally, the output query has the GraphStep followed by the index traversal code for the range $[minVal, maxVal]$ on the key attribute and the other HasSteps in the consecutive HasStep list and any other gremlin steps which may follow this list.

Chapter 6

Conclusion

In this chapter we briefly summarize our contributions and put forth several possible future directions for the project that the further extensions might be based on.

6.1 Key Contributions

A key contribution of our work is that it serves as a proof-of-concept showing that in database index structures can be as efficient as the state of the art elastic search indexes. This also in some sense offers a better stability of the complete system by reducing a moving component of a third party application and thus reducing a large number of places where frequent faults or crashes can occur. Not only stability, this concept also offers a very vast flexibility to the user in terms of the design of index structure, which was previously restricted to what is available in elastic search or any other third party indexing application.

With the introduction of rewrite engine we can now in fact take a simple user gremlin query and optimize it ourselves to use the in database/graph index structures if available, which also makes our index structures equally easy to use like any other indexing mechanism like elastic search or solr. This work also opens up a much larger new area for any audience willing to try a new innovative index structure and they will be only restricted by the fact that their index must be representable as a graph. We already support the pre-implemented versions of B-Tree and B+ Tree index structures for the aforementioned infrastructure which are one of the most common index structures in the databases community.

6.2 Further Extensions

This section describes in brief some of the several possible future directions directions of this work.

Selecting amongst multiple available indexes
We currently select the first possible usage of index to rewrite the given query. But,

suppose there are many indexes with different parameters on same key, how to select one? An improvement over this would be to add a metric for selecting the best index amongst a variety on indexes available on different attributes of a gremlin query.

Extending to other databases

Also, this work is clearly not restricted to a single graph database but applies to a large variety of them. Extension implementations of the same for other Graph DBs like Neo4J is likely to generalize these kind of indexes even further. Converting the system into a easy to use plugin for Janusgraph/Neo4J is a direct way popularizing this kind of index in the community.

Adding more index structures or a GiST

Expanding this work into something where any user can be allowed to just add code to a few functions to introduce another kind of new/novel index can greatly accelerate the usage of this kind of index structures. Another possible way to add the power of a generalized search index would be to implement something like the GiST (Generalized Search Tree) mentioned by Hellerstein et. al [11].

Other kinds of traversals

Storing index structures along with the data opens multiple new avenues which can be exploited to optimize the execution of queries on Graph DBs. Apart from the traditional index-to-data traversal, data-to-index and index-to-index movements too are possible now.

Data to Index

Moving from Data to Index can help us easily determine the key on which the index is based given the data vertex. This might at first not seem to be really useful owing to the fact that if the indexing key is already a property of the vertex, then once we have the vertex, we can directly find its value instead of going to the index vertex and adding one extra vertex fetch.

However, consider the case where the index is present on the some aggregation of keys rather than directly the key. For example an index on the percentile of marks received by a student can help to trivially compute his percentile which in the absence of such an index would involve multiple vertex fetches.

Index to Index

Another possible extension to this is the **Index-to-Data-to-Index** traversal which can help us answer queries like what percentile of salary is being received by the students with top x percentile marks.

In this kind of queries, an execution can be framed that using the index structures, filters the data swiftly to get to a specific set of vertices/data nodes. This execution can then use the idea from the data to index traversal to answer the aggregation queries on this filtered set of nodes.

Creating/obtaining a set of workloads / benchmark specifically to observe the effects of these kind of optimizations (and novel use of index structures) is an issue one might have address before jumping right into these kinds of executions/traversal though.

Creation of index structures automatically based on workload
Analysis of workload of a set of queries can help in further refining the indexing in Graph DBs. We can suggest suitable indexes to be created internally without the knowledge of the user and use these indexes to automatically modify the queries for making use of these indexes. Apart from simple suggestion of indexes based on workloads, this requires analysis of each index to judge whether it offers a good trade-off in terms of performance improvement over increase in size of the graph (due to the additional vertices and edges we add to accommodate the index structure).

Bibliography

[1] Benchmark: http://ldbcouncil.org/developer/snb

[2] SNB Queries Specifications: http://ldbc.github.io/ldbc_snb_docs/ldbc-snb-specification.pdf

[3] LDBC Data Generator: https://github.com/ldbc/ldbc_snb_datagen

[4] Gremlin reference book: http://kelvinlawrence.net/book/Gremlin-Graph-Guide.html

[5] Tinkerpop Documentation: http://tinkerpop.apache.org/docs/current/reference/

[6] Tinkerpop Java Documentation: http://tinkerpop.apache.org/javadocs/current/core/

[7] Towards Integrated Graph Algebra for Graph Pattern Matching with Gremlin: Harsh et al

[8] The Gremlin Graph Traversal Machine and Language: Marko A. Rodriguez

[9] Query run times: https://docs.google.com/spreadsheets/d/1lSBP4IWh5VKFXzT6qyYzQJn_R93dUtRdPldVc9AqR38/

[10] Git Repository: https://github.com/Prakhar0409/Ghost-Index-in-Graph-DB

[11] Generalized Search Tree: http://db.cs.berkeley.edu/papers/vldb95-gist.pdf