Scale Independent Query Processing using Optimal Access Schema

1S. Anuja, 2M. Monisha Devi, 3Dr. Radha Senthilkumar

1Mtech Student, Madras Institute of Technology, Anna University, Department of IT, 600044, Chennai, India
2Teaching Fellow, Madras Institute of Technology, Anna University, Department of IT, 600044, Chennai, India
3Assistant Professor, Madras Institute of Technology, Anna University, Department of IT, 600044, Chennai, India

Abstract

Background: Big data is data that exceeds the processing capacity of conventional database systems. Managing large and rapidly increasing volume of data has been a challenging issue for many decades. Querying big data is another challenge, since it requires specific techniques for providing reasonable performance guarantee. While existing optimization strategies are providing such guarantees to some extent, these techniques are insufficient for complicated queries which require reading unbounded amounts of data, resulting in an increasing response time with the increase in the size of the database. In order to minimize the response time, the queries should provide scale independence i.e. answer query Q in big database D within available resources, it also answer Q without performance degradation when D grows making Q scalable. Scale independence can be achieved using various techniques. One of the techniques concentrates on access schema which uses the additional information about the accessing information (dataset) to provide scale independence. An optimal access schema is proposed to retrieve the subset D₀ from the original dataset D, and executing queries on the subset results in a better performance even when D grows.

INTRODUCTION

Big data refers to data sets whose size exceeds the capabilities of the traditional database technology. Big Data is always noisy, dynamic, heterogeneous, and inter-related. In Big Data, the amount of data outgrows with the capabilities of query processing technology used. The major challenge in big data query processing is the amount of data to be consumed because, it is difficult to spend considerable amount of time in tuning the data before processing, as the size exceeds petabytes.

As big data avails scalability, the expense in consuming data for query processing is minimum. In the existing work, two techniques are used which includes incremental query answering and query rewriting using views. In proposed work, the query processing for big data focus on a new direction where subset retrieval becomes predominant. The concept of access schema has been used to achieve scale independence. The increased complexity in graph database with the increase in the size of graph leads to focus on different NoSQL database called mongodb. This leads to focus on an indexing i.e. bitmap indexing, which is easier to build and efficient for querying a large dataset. Since bitmap indexing can’t be performed in neo4j and the complexity lies in providing partition tolerance, consistency in neo4j graph database and a document oriented database called mongodb is used which provides the above mentioned characteristics for achieving scale independence with larger datasets.

Related Work:

Query processing with reasonable efficiency on big data is often being a problem when the size of the data is considered. To provide a better performance when handling a large dataset, scale independence is addressed. Several approaches have been proposed to provide scale independence and most of the approaches select views that may speedup query execution on average.

Michael Armbrust et al. (2013) described a scale independent incremental view selection and maintenance system. Michael Armbrust et al. (2011) proposed a declarative language called Performance Insightful Query...
Language (PIQL) which provides scale independence by calculating an upper bound on the number of key/value store operations for any query. PIQL has an extension to SQL (Structured Query Language) for providing extra bounding information to the compiler. PIQL provides a paginate clause for processing unbounded queries. PIQL in (Michael Armbrust et al., 2010) provided strict bounds on the number of I/O operations for any query to be answered. V. Barany et al. (2013) described query answering with access restrictions and integrity constraints in which the accessible portions of schema are identified for processing. Deutsch et al. (2005) discussed about rewriting queries using limited access patterns under integrity constraints. Chen, Li. (2003) described in detail about binding pattern of a relation used for computing the complete answer for a query.

Our contribution lies in providing scale independence for querying graph database and mongodb in hadoop environment. In order to provide scale independence, there is a need to form an access schema which efficiently retrieves the subset regardless of the underlying size of database. Once the access schema is formed, the attribute for indexing is identified, which can result in optimality.

Proposed Methodology:
A. Query processing using optimal access schema:
Query processing in big data is often expensive and requires special techniques completely different from traditional database. Efficient query processing requires a special type of independence called scale independence to result a reasonable performance. In Proposed work, to make query answering feasible in big dataset, the importance of additional information provided with the dataset in the form of indexing and constraints is highlighted which helps in executing query scale independently.

B. Query processing using graph database:
Graph databases are excel at managing highly connected data and complex queries independent of the size of the total dataset.

I. Neo4j:
Neo4j, a NO-SQL graph database is used which results in high query performance on complex data and uses cypher a declarative graph query language for expressive and efficient querying and updating of graph.

The proposed work with neo4j includes an access schema formation and indexing on the dominant property of a node. Indexing the dominant property provides a bound on the search for retrieving the subgraph. In the first module, the dataset is loaded in neo4j graph database. The second module creates the complete graph with the dataset specifying all the properties and relationships. Then in the third module, the access schema for the user query is formed which specifies the labels, properties and relationships in the graph. Once access schema is formed, indexing is performed on the property, which effectively bounds the graph search according to the algorithm.

An algorithm is given below for indexing, to retrieve subgraph from a large graph dataset.

Algorithm for Subgraph Retrieval(Gq, Gdb, Sq, Sdb, Mq, Qne, Pq, Pd)
Input: Gq is the query graph, Gdb is the database graph, Sq is the set of nodes in the Gq, Sdb is the set of nodes in the database, Mq contains all nodes matching with the query, Qne is the set of nodes matches with the edge for the query, Pq is the set of properties/attributes in a query, Pd is the dominant property in a query
Output: sub graph as Qne
1. for every node Nq in Sq do
2. Ndb = the best mapping of Nq in Sdb
3. if Ndb == NULL then
4. continue
5. endif
6. if Nq has no matches in Mq then
7. return NULL
8. else if (Nq, Ndb) matches with Mq then
9. put (Nq, Ndb) into Qne
10. set index on Pd
11. is Pq has more than one property then
12. set index on low cardinality property
13. return Qne
14. endif
15. endfor
Fig. 1: Snippet of Facebook Friends Relations in Neo4j.

Query Syntax: MATCH (me)-[:FRIEND_OF]-(:FRIEND) WHERE me.name = "Anuja" RETURN FRIEND.pid, FRIEND.name, FRIEND.city

Example: If the user query is to find the friends of a particular person, then the following subset will be the result as per the algorithm. Table 1 shows the resultant subset of the user query after indexing the attribute as per the algorithm.

Table 1: Resulting Subset of Given User Query.

<table>
<thead>
<tr>
<th>Uid</th>
<th>Name</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1153450798</td>
<td>JothiLakshmi</td>
<td>Walajapet</td>
</tr>
<tr>
<td>539656358</td>
<td>Vanaja</td>
<td>Walajapet</td>
</tr>
<tr>
<td>1153450798</td>
<td>Tamilvanan</td>
<td>Chennai</td>
</tr>
<tr>
<td>100003167815317</td>
<td>Supraja</td>
<td>Walajapet</td>
</tr>
<tr>
<td>100000758287135</td>
<td>Lakshmi Narasimman</td>
<td>Walajapet</td>
</tr>
</tbody>
</table>

C. Query processing in mongodb:

MongoDB and Hadoop are a powerful combination for complex analytics and data processing. In certain cases, to perform complex data processing, hadoop support is needed. The entire dataset is stored in MongoDB and HDFS which can be processed in hadoop environment.

I. System architecture for query processing with mongodb in hadoop environment:

The Architecture shows the typical search of a user in social networking sites like facebook, twitter etc. Fig.3 shows the system architecture of the proposed system. As per the architecture, when the user query is posed, query execution is performed based on the access schema and indexing which is done during query processing. The dataset is loaded in hadoop storage and NoSQL MongoDB for faster data processing.

The proposed work with MongoDB and Hadoop includes the access schema formed for the user query and performs bitmap indexing on the attribute from the set of attributes specified in the collections. Query answering can be done efficiently using indexing on an attribute in the set of attributes of collection specified in the access schema. Indexing can be performed dynamically for an attribute which has low cardinality constraint to provide better performance.
II. Scale independent query processing using mongodb in hadoop environment:

There are three modules involved in scale independent query processing using mongodb in hadoop environment. The modules are Dataset Loading, Query Processing without indexing and Query Processing with indexing. The modules are described below.

a. Dataset loading:
The dataset is loaded in to mongodb which in turn stores the dataset in hdfs. The dataset stored in mongodb can be accessed through hadoop environment for query processing.

b. Query processing without indexing:
For processing a query without indexing, user query is given as input and the related collections available in the mongodb are searched sequentially. Finally the answer to the user query which is the subset is stored in an array and displayed as a result.

c. Query processing with bitmap indexing:
If the query has to be processed based on the indexing, then here comes actual proposed work. To process a query based on the user input which is one or more attributes, the access schema is formed with the necessary collections to process the query and a dynamic indexing is performed based on the access schema and relationship cardinality which can provide a better performance. Access schema is formed by identifying the necessary collections and attributes to be accessed. With the access schema, bitmap indexing is performed on an attribute which occurs redundantly in the relationship based collection and result of a query is stored in an array and displayed as a subset of the large dataset.

Algorithm for Bitmap Indexing

Input: $C_i$ is the Collection name, $A_i$ is attribute name, $C_q$ is the collection in access schema
Output: Subset of the query

for each attribute $A_i$ in $C_o$
if $A_i$ in $Q$ is indexed
then $A_o=A_i$
else
return NULL
endif
if $A_o$ has matches in $C_q$
continue
while $A_o$ is in $C_q$
set bitmap index as 1
else
set bitmap index as 0
end if
end for

In this algorithm, collections, collection name specified in access schema and attribute name is given as input. If any of the attribute in the collection is indexed, that attribute is assigned as attribute for bitmap index. Otherwise it returns null. If the assigned attribute exists in the collection specified in the access schema, then the
redundancy of the attributes is checked. If it is redundant then the bitmap index is set 1. Otherwise the bitmap index is set to 0. The collection with bitmap index 1 is the subset of the query.

Example: Query is to find the friends of a particular person who lives in a particular location. Table 2 shows sample dataset for person collection.

Table 2: Person Collection in Mongodb.

<table>
<thead>
<tr>
<th>Uid</th>
<th>Name</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>539656358</td>
<td>Vanaja</td>
<td>Walajapet</td>
</tr>
<tr>
<td>1153450798</td>
<td>Tamilvanan</td>
<td>Chennai</td>
</tr>
<tr>
<td>1000003167815317</td>
<td>Supraja</td>
<td>Walajapet</td>
</tr>
<tr>
<td>100000758287135</td>
<td>Lakshmi Narasimman</td>
<td>Walajapet</td>
</tr>
<tr>
<td>1287104219</td>
<td>Brindha Lakshmi</td>
<td>Bangalore</td>
</tr>
<tr>
<td>100001525300588</td>
<td>Kani Mozhi</td>
<td>Ranipet</td>
</tr>
<tr>
<td>100001618273212</td>
<td>Jagadeesan</td>
<td>Tiruvannamalai</td>
</tr>
<tr>
<td>100001715482582</td>
<td>Mounesha</td>
<td>Arcot</td>
</tr>
</tbody>
</table>

Table 3 shows the friends collection. As per the algorithm, the collection name & attribute name to get the person details and the collection name specified in the access schema is given as input according to user query.

Table 3: Friends Collection in Mongodb.

<table>
<thead>
<tr>
<th>Pid</th>
<th>Fid</th>
</tr>
</thead>
<tbody>
<tr>
<td>690304591055546</td>
<td>539656358</td>
</tr>
<tr>
<td>690304591055546</td>
<td>1153450798</td>
</tr>
<tr>
<td>690304591055546</td>
<td>1000003167815317</td>
</tr>
<tr>
<td>690304591055546</td>
<td>100000758287135</td>
</tr>
<tr>
<td>690304591055546</td>
<td>1287104219</td>
</tr>
</tbody>
</table>

Then the person collection is checked for attribute which is indexed and assigns that as the attribute based on which bitmap indexing should be performed. With that attribute it searches for the friend’s id in friend’s collection as shown in Table 3 which is related to the person and set bitmap index in friend’s collection as 1 for those who’s FriendId has relation with the PersonId which is given as input in algorithm. Then with the FriendId, it searches the friends name and location and other details and displays that as a result of a query as shown in Table4.

Table 4: Sample Resulting Subset.

<table>
<thead>
<tr>
<th>Uid</th>
<th>Name</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>539656358</td>
<td>Vanaja</td>
<td>Walajapet</td>
</tr>
<tr>
<td>1153450798</td>
<td>Tamilvanan</td>
<td>Chennai</td>
</tr>
<tr>
<td>1000003167815317</td>
<td>Supraja</td>
<td>Walajapet</td>
</tr>
<tr>
<td>100000758287135</td>
<td>Lakshmi Narasimman</td>
<td>Walajapet</td>
</tr>
<tr>
<td>1287104219</td>
<td>Brindha Lakshmi</td>
<td>Bangalore</td>
</tr>
<tr>
<td>100001525300588</td>
<td>Kani Mozhi</td>
<td>Ranipet</td>
</tr>
<tr>
<td>100001618273212</td>
<td>Jagadeesan</td>
<td>Tiruvannamalai</td>
</tr>
</tbody>
</table>

Experimental Results:

A. Neo4j graph database:

In neo4j graph database, a node could start with a single property and grow to a few million. The subgraph retrieval is how you query a graph, navigating from starting nodes to related nodes according to an algorithm. Facebook dataset is used for the experimental purpose. Facebook.com is a social networking site and it allows users to enter “friend” relationships with one another. All friendships are indistinguishable with respect to tie strength, and informal reports from Facebook users suggest that users enter these relationships rather casually.

Example Query: 1
Finding all the friends of a particular person.

Query Syntax: MATCH (Person)-[:FRIEND_OF]-:FRIEND WHERE Person.name=" Anuja" RETURN Person-[::FRIEND_OF]-FRIEND

With respect to our algorithm, since the label is mentioned in the query, all the nodes of a label named ‘Person’ are mapped with the nodes in the database. Suppose if the database is empty, it will return null. If none of the nodes are matching with the query then it returns null. Otherwise if all the nodes in the query and the database match with Mq, those nodes are stored as sub graph of the given query and indexing is performed on the property specified in the query. If there are multiple properties in the query then, the one which has low cardinality is chosen for indexing. In our example query, the label named Person is mentioned for displaying the sub graph, and it will take all properties of a label and resulted as a sub graph.

Example Query: 2
Find all our friends of friends.
MATCH (Person)-[:FRIEND_OF]->(A)-[:FRIEND_OF]-(B) WHERE Person.name = "Anuja" RETURN Person, A, B

Example Query: 3
Finding all the friends of a particular person residing in a particular city
MATCH (Person)-[:FRIEND_OF]->(FRIEND) WHERE Person.name = "Anuja" AND FRIEND.city = "Arcot" RETURN Person, FRIEND

As per the experimental analysis with facebook dataset, it has been found that the algorithm gives a reasonable response time when compared to existing query processing techniques. Fig. 3 shows the different response time for with indexing and without indexing using Neo4j Graph database. Queries Q4 and Q5 in Fig. 4 are related queries and hence took less response time in without indexing. As per Fig. 3 the time taken to execute queries with indexing is considerably reduced when compare to the response time for without indexing.

Fig. 3: Elapsed time for Subgraph Retrieval before and after Indexing in Neo4j.

As the data is getting bigger, it becomes little difficult to store and process in neo4j which is experimented during the addition of more nodes and the corresponding relations as edges between the nodes. Hence to handle large dataset, a NoSQL database say mongodb is opted to store and process big data which doesn’t affect the performance with respect to response time when the data size increases.

B. Mongodb in hadoop environment:
Mongodb, an open source NoSQL document store database with hadoop environment is adopted for our experimental purpose. With respect to the algorithm, the collections to be accessed and collection name, attribute name are identified and given as input. Based on the input, it searches the collection to find the indexed attribute and if there is an attribute indexed, it stores that attribute as the attribute to be indexed using bitmap. Otherwise it returns NULL. Then it searches for the existence of the same attribute in the collection specified in the access schema say Friends collection according to our example. If the attribute is there and is redundant in the collection to specify the relationship between the attributes in Friends collection, those are indexed with bitmap indexing. Then the collection is searched for the attributes related to the indexed records with respect to user input and resulted as subset of the query. With Fig. 4, it is clear that bitmap indexing results in a reasonable response time.

Fig. 4: Elapsed time for retrieving the subset with Mongodb in Hadoop Environment for larger dataset.
Conclusion:
Querying big data often encounters problems with performance. The existing techniques are insufficient to provide scale independence. In our work, querying graph database like neo4j with additional information represented as access schema is discussed extensively. Also query processing with respect to mongodb in hadoop environment is discussed which incorporates optimal access schema with bitmap indexing. With better utilization of access schema, the scale independence is made easier without any performance bottleneck even the size of the database scales rapidly. An algorithm is given to perform indexing on a property for retrieving a sub graph from a large graph based on the cardinality constraints. Another algorithm is given for performing bitmap indexing in query processing with mongodb in hadoop environment. The results of query processing are also experimented with indexing and without indexing in Neo4j graph database as well as with bitmap indexing and without bitmap indexing using mongodb in hadoop environment and it results in a considerable performance when compared to existing query processing techniques in big data. In future, this can be extended to experiment the query response time for grouping and aggregate queries with respect to access schema.

REFERENCES
Yan, X., P.S. Yu and J. Han, 2005. Graph Indexing based on Discriminative Frequent Structure Analysis. ACM Transactions on Database Systems, 30(4): 960-993.