ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

Graph Mining Extensions in Postgresql

G. Anuradha* and D. Joel Varma
Department of CSE, GMRIT, Rajam-532127, Andhra Pradesh, India; anuradhagovada@gmail.com

Abstract

Objectives: To develop an extension for postgresql which will include the fundamental graph operations required for graph algorithms to allow moderately associated datasets to have both(graph and non-graph) concepts. Methods/Statistical Analysis: The extension in postgresql uses the query optimizations of the SQL, although this can be done manually, so it was avoided. The algorithm used by the extension was a depth first search (DFS) algorithm. It is very simple and was written just to show the speed difference between using SQL and extensions. Findings: Relational databases are hard to work on graphs, as the tabular form cannot represent graphs well which Dedicated Graph databases like Neo4j, Titan and Allegro-graph can and also provide performance when it comes to graph related queries which the relational databases deem complex and sometimes not possible. Common Table Expression (CTE) in postgresql perform graph related operations, but unable to perform more optimized and complex operations. To overcome this problem, we are going to develop extensions for postgresql. Application/Improvement: The time and cost i.e. the memory required is very less compared CTE’s.

Keywords: Allegro, Neo4j, Postgresql, Structured Databases, Titan

1. Introduction

Structure mining or structured data mining is the process of finding and extracting useful information from semi-structured data sets. Graph mining is a special case of structured data mining, it is the process of gathering and analyzing the data represented as graphs. In this new world where data has replaced money, data has been created exponentially and chaotically. Data actually exists in semi structured form. The growth of the use of semi-structured data has created new opportunities for data mining, which has traditionally been concerned with tabular data sets, reflecting the strong association between data mining and relational databases. Much of the world’s mineable and interesting data does not easily fit into relational databases, but through time generations of computer, engineers have been trained to believe this was the only way to handle data, and data mining algorithms have generally been developed only to deal with tabular data. We have used this way of handling data for so long that we have not looked for other ways to look at data. This type of data is used by many social sites and business we can cluster.

These days graph data have become a part of our daily life. Every day we use social networking services, change web pages, make phone calls, and update reference information in our e-document. All of these which are our daily behaviors are represented by graph. Social networking service companies such as Facebook and LinkedIn, internet search companies such Google and Yahoo, and mobile service companies have accumulated enormous amount of graph data and analyzed these graph data for their business purposes. There are several famous graph data mining methods that extract significant features from graph data: Page Rank, connected component, diameter, RWR, degree distribution, Eigen values, and belief propagation. This information is essential for the companies to predict the behaviors of their customers, and to optimize the use of resources.

1.1 Introduction to SQL

A special purpose programming language SQL was designed for managing data held in a RDBMS (relational database management system). SQL consists of a DDL (Data Definition Language), DML (Data Manipulation Language), and a DCL (Data Control Language). The scope of SQL includes data insert, select (query), update...
Graph Mining Extensions in Postgresql

and delete, schema creation and modification, and data access control. It became a standard of the American National Standards Institute in 1986 and of the ISO in 1987. Since then there have been a lot of improvements to suite the new needs. Albeit such standards existed, most SQL code is not entirely portable among different database systems without tweaks.

Though SQL has been modified to be efficient over time with all the query retrieval optimizations, it is still not flexible with this process. The user has no control over the optimization hence in some cases it cannot be so efficient. Especially when doing data retrieval over graph like data stored in a relational database system i.e. tabular form. For example, if the graph data was stored as an edge list and we want to find whether two nodes are connected or not, we need to do a lot of recursive iterations over the edge table going one level of depth at a time. Using SQL, where RDBMS are very fast and writing queries in SQL for graph traversal operations or algorithms will not be easy. Some of the limits of SQL regarding graph related operations are mentioned below including some already mentioned.

1.2 PostgreSQL
PostgreSQL is an ORDBMS (object-relational database management system) with an emphasis on extensibility. As a database server, its primary function is to store data securely while supporting best practices and to allow for retrieval of data at the request of other software applications. It can handle a variety of workloads ranging from small single machine applications to large Internet facing applications with concurrent users. PostgreSQL has a large number of extensions. In addition to the possibility of working with the major proprietary and open source databases, PostgreSQL supports migration from them, by its extensive standard SQL support and available migration tools.

PostgreSQL installation provides a build infrastructure for extensions, called PGXS, so that simple extension modules can be built simply against an already installed server. PGXS is mainly intended for extensions that include C code, although it can be used for pure SQL extensions too. Note that PGXS is not intended to be a universal build system framework that can be used to build any software interfacing to PostgreSQL; it simply automates common build rules for simple server extension modules. For more complicated packages, you might need to write your own build system. These extensions are not just useful to add functionality they are also very efficient as they are written in a programming language. PostgreSQL allows those programming languages to use the SQL functions too which add to the efficiency of the extension.

The three aspects we plan to counter are:

i) Memory in case of Dedicated graph databases:
Graph databases require data structured format which can support graph operations easy. So if we wish to have both relational and graph operations we need two instances of the same database, one in relational i.e. tabular format and in graph supporting structures such as adjacency list or matrix etc. which, doubles the amount of memory we normally require, and also may be feasible for large organizations but may not be for small ones.

ii) Performance in case of relational databases:
- Recursive operations need to apply the query constraints on result set at every iteration to remove the unnecessary data. Which is very expensive, in the worst case the query might be applied on the whole dataset available which we cannot afford in terms of performance.
- If graph operations are performed by the application itself then bottleneck occurs at transfer of data from server to application, as transfer of large data might slow the application down.

iii) Programmability
- Even though SQL queries are very capable, writing queries for graph operations can be very complex or may even be not possible in some cases.

These three problems were rectified as follows:

i) Memory in case of Dedicated graph databases:
- By using an extension we can provide basic facilities of a dedicated graph database on relational database, so we don't need to replicate our database or restructure it.
- We can keep the old structure and get the same performance on relational databases.

ii) Performance in case of relational databases:
- When an application wants to perform graph
operations it can directly pass the parameters and constraints to the database layer and get the results, instead of getting data from server and then processing it, which will save the client from storing all the data and processing it.

iii) Programmability

- As extensions for PostgreSQL can be written in programming languages like C++, Java, Python etc. we have more flexibility and programmability while using SQL.
- This also allows the client to be more specific about the constraints on the kind of data he needs.

2. Methodology

2.1 CTE (Common Table Expressions)
To overcome the drawbacks of SQL regarding recursive operations a new syntactic structure called CTE was introduced. A Common Table Expression (CTE) can be thought of as a temporary result set that is defined within the execution scope of a single SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW statement. A CTE is similar to a derived table in that it is not stored as an object and lasts only for the duration of the query. Unlike a derived table, a CTE can be self-referencing and can be referenced multiple times in the same query.

The Main application of CTE’s useful for graphs is that it allows us to use recursive operations. Using a CTE offers the advantages of improved readability and ease in maintenance of complex queries. The query can be divided into separate, simple, logical building blocks. These simple blocks can then be used to build more complex, interim CTEs until the final result set is generated. CTEs can be defined in user-defined routines, such as functions, stored procedures, triggers, or views.

2.2 Structure of CTE
CTE’s structure is divided into two parts.
- Base query
- Recursive query

  Base query is the query that is executed first and is never again repeated, any type of initialization’s can be done in this.

  The recursive query is the query which will refer to its own result.

  The Base query and Recursive queries are used to form a result set which will again be repeatedly used with the result of Recursive query in the next iteration.

2.3 CTE Example
A graph traversal query as the example mentioned before, this CTE will find all the nodes in a certain depth from a given node.

Create or replace function subgraph (in in_id integer, in in_depth integer, in in_etypes integer[]) returns table(id integer, depth integer, path integer[]) as

```
$body$
with recursive searchgraph(id, link, depth, path, cycle) as
  (select g.aid, g.bid, 1, array[g.aid], false
   from graph g
   where g.aid = in_id and g.etype = any(in_etypes)
   union all
   select g.aid, g.bid, sg.depth + 1, path || g.aid, g.aid = any(path)
   from graph g, searchgraph sg
   where (g.aid = sg.link)
   and not cycle
   and depth < in_depth
   and g.etype = any(in_etypes)
  )
select distinct on (link) link, depth, path
from searchgraph
order by link, depth asc
$body$
```

languagesql volatile
cost 100
rows 1000;
alter function subgraph(integer, integer, integer[])

2.4 C – API for Postgresql
As we have seen extensions are of two types
- Server-side.
- Client-side.

Both of these works differently as client and server have different types of configuration and memory access. Server side has a limitation over memory and hence can
only handle small and medium sized results unless the server highly equipped. Client side extensions are faster and have memory at disposal compared to server side. But server side has to deal with concurrent requests while client does not hence both are very powerful and fast in their own domains.

3. Results and Analysis

We have chosen an operation that has the most impact on majority of graph operations, the idea was to make the primary operations faster to boost the overall time of all the not so primary operations i.e. algorithms such as finding radius or diameter or finding articulation points etc. The operation that most algorithms perform is traversal from nodes to specified nodes or in the form of a search.

3.1 Dataset

The dataset taken was generated by us. The Dataset contains of about 2000 nodes, and the data is stored in postgresql in the form of an edge table. The edges were formed by randomly choosing nodes.

The schema of the table containing the data is given below.

| NODE_ID integer | NODE_ID integer |

The scheme is very simple and the dataset is not very large as the experiments were done on an average personal computer with 4GB ram and 2.4 GHz processor with 500GB hard disk.

3.2 The Extension in Postgresql

The extension in postgresql uses the query optimizations of the SQL, although this can be done manually, there was no need for it hence it was avoided. The algorithm used by the extension was a Depth First Search (DFS) algorithm. It is very simple and was written just to show the speed difference between using SQL and extensions.

The algorithm first forms a prepared statement. Then it uses the prepare statement to query all the nodes adjacent to the key node and remembers them with their node-id's and marks the seen ones as touched. Then it picks a node from the touched ones and repeat's the process. If a node which has already been seen is found it is ignored as it cannot contribute any new nodes i.e. with a lesser depth count we cannot find new nodes from there which cannot be discovered in future iterations from other nodes.

The algorithm for the extension is as follows

```sql
Function friends(key, prepareStatement, Depth, path)
  If(Depth==0)
    return;
  resultSet = runStatement (prepareStatement,key);
  foreach node in resultSet as nodei
    if(!touched.check(nodei))
      map.add(nodei,path);
      path.add(nodei);
      touched.add(nodei);
      friends(key,prepareStatement,Depth-1,path);
    return;
```

Even a simple algorithm like this can perform well when compared to CTE's.

3.3 Comparison of CTE with Code

The time and cost for CTE's is very much more compared to the extension. The given Figure 1 shows the graph that as the depth for the algorithm grows the time difference is very high and significant. It's not just the time and cost i.e. the memory required also is very less. As you can see at a depth greater than 15 there is no graph, that's because the system on which this was tested crashed when CTE's were used to go to depths greater than 15 while the extension did not, the time also did not change much. The extension did even go to a depth of 1000 mile seconds and still did not crash, that in case of CTE would not at all be possible.

![Figure 1. Time and cost difference between CTE and extension.](image)
4. Conclusion

The way to increase the efficiency of graph mining operations over relational databases by using extensibility of postgresql platform i.e. by writing an extension for postgresql which does the basic graph traversal operation. Even though the extension is far inferior to the efficiency provided by dedicated graph databases, it will be a lot less expensive, more affordable and feasible to small and medium projects which require both graph and relational operations. But do not have the need for a fully dedicated graph database management system.

5. References