Computing Complex Graph Properties with SQL Queries

Xiantian Zhou
University of Houston
Houston, TX 77204, USA

Carlos Ordonez
University of Houston
Houston, TX 77204, USA

***Abstract*—Within big data analytics, graph problems are as important as machine learning. There exist many algorithms to analyze large graphs but they are limited by main memory. On the other hand, a lot of data stored on DBMSs that needs to be analyzed as graphs. Moreover, DBMSs can work in parallel and they do not have RAM limitations. In this paper, we propose several algorithms that produce metrics and show properties of the graph as well as help us to understand the graph structure specifically diameter and betweenness centrality and k-betweenness centrality. This work is a big step beyond transitive closure and recursive queries. We propose optimized SQL queries that work on a large graph stored in relational form as triples can compute these in a more flexible and efficient manner. We study how to optimize such SQL queries combining demanding joins and aggregations that remove main memory limitation and also they can work in parallel. Finally, we provide an experimental evaluation to understand accuracy and performance in a columnar DBMS. We compare our algorithms with popular platforms including Python and Spark. We experimentally show our SQL algorithms are accurate and efficient.**

***Index Terms*—Graph Metrics, SQL Queries, Betweenness Centrality, Query Optimization,Columnar DBMS**

I. Introduction

Graph analytics remains one of the most computationally intensive tasks in big data analytics. The main reason is due to the large graph size, structure of the graph and patterns presented in the graph. Large-scale graphs have been widely applied in many emerging areas. Typical graph examples can be social networks, road connecting cities, telecommunication, flights linking airports and so on. On the other hand, relational databases remain the most common technology to store transactional and analytical databases, due to optimized I/O, robustness and security control. A lot of data stored on DBMSs (database management system) can be potentially analyzed as graphs [14]. Even the data is not in DBMS, it is fast to load a large data set into the DBMS. Also, it has been established that columnar DBMS perform better than GraphX or array DBMS for certain kind of graph queries [3], [7]. However, processing large graphs in large scale distributed system has not received much attention in DBMS using relational queries.

Understanding the structure of the graph is more difficult and complex than solving fundamental graph problems like exploring the graph [3] and transitive closure [7]. It has been done in many algorithms [5], but not with queries. With columnar DBMSs, we revisit the problem of solving graph algorithms with SQL queries. We propose several algorithms solved with SQL queries that can help to understand the structure of the graph. Our goal is to prove that DBMS can help us to understand the graph structure with something that is more complicated than what was done before with recursive queries (transitive closure) [15]. Moreover, columnar DBMS provide significant accelerations in Join and Group By queries.

In our work, we propose several algorithms that produce metrics and show properties of the graph such as the diameter of the graph, the betweenness centrality of a vertex and k-betweenness centrality. We study how to express the computation of these algorithms only with relational queries and how we can optimize the queries. Most of the existing graph databases fail when data volume is too large. Also, the usability of these graph databases is comparatively less than relational DBMSs as they have no standard set of rules. Moreover, these graph databases along with many popular platforms including Python and Spark are limited by main memory. We believe efficient graph algorithms for relational databases will avoid wasting time exporting the data or setting up external systems. Also, it has been established that columnar DBMS perform better than GraphX or array DBMS for certain kind of graph queries [3], [7]. In our opinion, even though query optimization is classical, well studied, topic optimization of relational queries on graphs needs further research.

II. Definitions and Background

A. Graphs

Let $G = (V, E)$ be a directed graph with $n = |V|$ vertices and $m = |E|$ edges. An edge in E links two vertices in V and has a direction. Our definition allows the presence of cycles and cliques in graphs. A cycle is a path which starts and ends at the same vertex. A clique is a complete sub-graph of G. The adjacency matrix of G is a $n \times n$ matrix such that the cell i, j holds 1 when exists an edge from vertex i to vertex j.

From database perspective, graph G is stored in table E as a list of edges (adjacency list). Let, table E be defined as $E(i, j, v)$ with primary key (i, j) representing the source and destination vertices and v representing a numeric value e.g. cost/distance. A row from table E represents the existence of an edge. For undirected graph, for each edge (i, j, v), we add reverse edge (j, i, v) meaning both edge direction in both ways. In summary, from a mathematical point of view E is

a sparse matrix and from a database perspective, E is a large and narrow table having one edge per row.

B. Queries used in solutions

We use standard SQL queries to analyze graphs. SQL queries are particularly useful in handling structured data. The standard SPJA includes selection, projection, join, aggregation queries. The SPJ queries are written as σ, π, and $\bowtie$ in relational algebra. The SQL queries we used can be recursive and non-recursive. Non-recursive SQL queries solve common graph exploration problems like counting triangles. A simple example of non-recursive query in terms of relational algebra with aggregation can be $\pi_{i,j,min(v)}(E \bowtie_{j=i} E)$. However, we also use recursive queries to solve harder graph problems such as betweenness centrality.

C. Columnar DBMS

A columnar database stores data in columns instead of rows (such as Vertica, MonetDB [11]). Storing data by column benefits the use of compression. Columnar databases can use traditional database query languages to load data and perform queries. The goal of it is to efficiently write and read data to and from disk in order to speed up the time it takes to return results of a query. The indexing mechanism of columnar DBMSs is significantly different than other traditional DBMSs. For instance, Vertica does not use indexes to find the data like row DBMSs. The data in the table are stored in projections, which can be optimized for different queries. A detailed review of columnar DBMS architectures in given in [1]. We use Vertica as the platform to run our SQL queries.

D. Definition of Problems

1) Diameter: The diameter of a graph is the length $max_{(i,j)}d(i,j)$ of the longest shortest path between any two graph vertices (i,j), where $d(i,j)$ is a graph distance. $d(i,j)$ is defined as the number of edges in a shortest path connecting i and j. When intermediate vertices and edges of $d(i,j)$ are restricted to a particular subgraph H of G, the distance between i and j is denoted $d_H(i,j)$. The diameter of a graph G is d if $max_{(i,j)}d(i,j) = d$.

2) Betweenness Centrality: Betweenness is calculated as the fraction of shortest paths between vertex pairs that pass through the vertex of interest. It is an important class of centrality measures the extent to which a vertex lies on the paths between others. Btweenness centrality can be used to identify critical vertices in a network. High centrality scores indicate that a vertex lies on a considerable fraction of shortest paths connecting pairs of vertices. We define a path from $i \in V$ to $j \in V$ as an alternating sequence of vertices and edges, starting from i and ending with j, such that each edge connects its previous and next vertex. Let $\varphi_{ij} = \varphi_{ji}$ denotes the number of shortest paths from i to j, where $\varphi_{ii} = 1$ by convention. And $\varphi_{ij}(m)$ is the number of shortest paths from i to j that $m \in V$ lies on. The betweenness centrality for vertex m is:

$$C_B(m) = \sum_{i \not\equiv m \not\equiv j \in V} \frac{\varphi_{ij}(m)}{\varphi_{ij}} \quad (1)$$

Define the pair-dependency $\delta_{ij}(m)$ as $\dfrac{\varphi_{ij}(m)}{\varphi_{ij}}$, then standard formula for betweenness centrality can be written as:

$$C_B(m) = \sum_{i \not\equiv m \not\equiv j \in V} \delta_{ij}(m) \quad (2)$$

Besides the full betweenness centrality, $k-$betweenness centrality captures information provided by paths whose length is within k unions of the shortest path length, is also a useful kernel for analyzing the importance of vertices. We will define k-betweenness centrality in the following manner. For a graph $G(V,E)$, let $d(i,j)$ denote the length of the shortest path between vertices i and j. We define φ_{ij_k} to be the number of paths between i and j whose length is less than or equal to $d(i,j)$+k. Likewise, $\varphi_{ij_k}(m)$ is the count of the subset of these paths that pass through vertex m. Therefore, $k-$betweenness centrality is given by:

$$C_B(m) = \sum_{i \not\equiv m \not\equiv j \in V} \frac{\varphi_{ij_k}(m)}{\varphi_{ij_k}} \quad (3)$$

III. Programming Algorithms with Queries

A. Diameter

The diameter of a graph is the length of the longest shortest path between any two vertices (i,j). The exact value of the diameter of a graph is achieved by calculating all pairs' shortest paths (APSP). The algorithms for the APSP problem include matrix multiplication or repeated squaring, the Floyd-Warshall algorithm, and transitive closure of a graph.

Algorithm 1: Transitive Closure

```
n ← |V|;
T_ij(0) ← 0, i, j ∈ V;
for i ← 1 to n do
    for j ← 1 to n do
        if i = j or (i, j) ∈ E then
            T_ij(0) ← 1
        end
    end
end
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            T_ij(k) ← T_ij(k − 1) ∨ (T_ik(k − 1) ∧ T_kj(k − 1))
        end
    end
end
return T(n) ;
```

Fig. 1: Transitive Closure algorithm.

With SQL, we could use linear recursive queries to implement transitive closure algorithm to get all the shortest path of any two vertices. But diameter is different since it focuses on the "longest" shortest path, so it does not have to get the reachability for every vertex. We compute $R_k = R_{k-1} \bowtie E$, and then only keep the shortest path from R_k to continue doing join. Hence, the iterations are of the form $k = 1, 2, 3....$ The base step produces $R_1 = E$. And then $R_2 = E \bowtie E = \pi_{i,j,min(v)}(E \bowtie_{j=i} E)$,... and so on.

When R_k becomes empty since no more rows satisfy the join condition, we reach the maximum iteration. The longest path in R_k is the diameter. The basic join query is shown below.

```
SELECT d+1, R.i, E.j, R.v+E.v
 FROM  R
 JOIN T ON R.j=E.i
 GROUP BY R.i, E.j
 HAVING R.i!=E.j

SELECT MAX(d) AS diameter
FROM RK
```

B. Betweenness Centrality

Fast sequential and parallel algorithms are available for computing betweenness centrality. Brandes described a fast sequential algorithm for computing betweenness in $O(mn)$ time for an unweighted graph [6]. But no one has developed one algorithm which works for DBMSs. One reason is that there are not stack, queue such data structures in DBMSs, so it would be difficult when developing complicate algorithms such as betweenness centrality. Another reason is the table size would grow largely when the algorithm needs multiple times of join, which makes the process take more time. We developed our solution for betweenness centrality problem according to Brandes's algorithm which is shown below.

Algorithm 2: Betweenness centrality in unweighted graphs

$C_B[v] \leftarrow 0, v \in V$;
for $s \in V$ **do**
  $S \leftarrow$ empty stack;
  $P[w] \leftarrow$ empty list, $w \in V$;
  $\sigma[t] \leftarrow 0, t \in V$; $\sigma[s] \leftarrow 1$;
  $d[t] \leftarrow -1, t \in V$; $d[s] \leftarrow 0$;
  $Q \leftarrow$ empty queue;
  enqueue $s \rightarrow Q$;
  while Q *not empty* **do**
    dequeue $v \leftarrow Q$;
    push $v \rightarrow S$;
    foreach *neighbor* w *of* v **do**
      // w *found for the first time?*
      if $d[w] < 0$ **then**
        enqueue $w \rightarrow Q$;
        $d[w] \leftarrow d[v] + 1$;
      end
      // *shortest path to* w *via* v*?*
      if $d[w] = d[v] + 1$ **then**
        $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$;
        append $v \rightarrow P[w]$;
      end
    end
  end
  $\delta[v] \leftarrow 0, v \in V$;
  // *S returns vertices in order of non-increasing distance from s*
  while S *not empty* **do**
    pop $w \leftarrow S$;
    for $v \in P[w]$ **do** $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \cdot (1 + \delta[w])$;
    if $w \neq s$ **then** $C_B[w] \leftarrow C_B[w] + \delta[w]$;
  end
end

Fig. 2: The best known and fast algorithm for Betweenness Centrality.

From the algorithm, we could see that there are two steps: computing the length and number of shortest paths between all pairs of vertices and adding all pair-dependencies. But Data structures such as list and queue are used in the algorithm and also this is a sequential algorithm. Thus there is no way to directly change the algorithm to SQL. We propose a solution to solve the betweenness centrality problem using standard SPJA queries by adding an aided column. And later we optimized it and largely reduced the table size.

For the first step, we could do linear recursive queries to get all shortest paths. Linear recursive are queries of the form: $R = R \cup (R \bowtie E)$, where the result of $R \bowtie E$ gets added to R itself. The iterations are of the form $k = 1, 2, 3....$ The base step produces R_1 = E. And then $R_2 = E \bowtie E$, $R_3 = R_2 \bowtie E$,...It seems similar to transitive closure. But computing betweenness centrality is far more difficult than transitive closure since it needs not only the reachability of each vertex but also the number of shortest paths, and all the intermediate vertices that lie on those paths. One intuitive way to do it is to store the starting vertex, ending vertex and all the intermediate vertices of the path in a row. We also created a new column called $'id'$ by using SEQUENCE that was added to each row to uniquely identify each path and used later to get pair-dependencies. The query to get R_k is shown below.

```
CREATE TABLE Rk as
 SELECT nextval('sequ') as id, k as d,
   t1.i as i, t2.j as j, t1.v + t2.v as v,
   t1.m1 as m1,t1.m2 as m2...
   t1.m(K-2) as m(K-2),
   t2.i as m(K-1)
 FROM R(k-1) t1 JOIN E t2 ON t1.j=t2.i;
```

For k-betweenness centrality, it will go through k depth of join and stop. For betweenness centrality, we should do iterations of join until the size of R_k becomes 0. Now we have $R_1, R_2, ...R_{k-1}, R_k$. it is necessary to union all the temporary tables together to select the shortest paths for all pairs. But the number of columns of R_k is different since the paths in those tables are of different length. For example, all the rows(paths) in R_3 need 2 columns to store intermediate vertices, but they need 3 columns to do that in R_4. Because of that, it is difficult to union all the table together and obtain the pair-dependency. So we changed all the columns storing intermediate vertices into rows before union all the table together. Now each row contains id, the starting vertex i, the ending vertex j, the length of the path v, and one intermediate vertex m. The queries is shown below:

```
CREATE TABLE U AS
SELECT id as id, d as d, i as i, j as j,
    v as v , m1 AS m
FROM RK
UNION ALL
SELECT id as id, d as d, i as i, j as j
    v as v, m2 as m
From Rk
UNION ALL
...
SELECT id as id, d as d, i as i, j as j,
 v as v , m(K-1) AS m
FROM RK;
```

The table $U(id, i, j, m)$ contains all the paths. Then we select all the shortest paths for each pair of vertices from table U and stores in table F. Since id was used to identify each unique path, we can obtain φ_{ij} and $\varphi_{ij}(m)$ by doing GROUP BY different columns. We performed $F1(i, j, \varphi_{ij}) = \pi_{i,j,count(id)}(F)$, $F2(i, j, m, \varphi_{ij}(m)) = \pi_{i,j,m,count(id)}(F)$. Finally, we obtained betweenness centrality for each vertex m using $\pi_{F2.m,sum(\varphi_{ij}(m)/\varphi_{ij})}(F1 \bowtie_{F1.i=F2.i \ and \ F1.j=F2.j} F2)$ to sum up all the pair-dependency.

```
CREATE  TABLE F1 AS
 SELECT i AS i, j AS j, count(distinct id)
 FROM F
 GROUP BY i, j;

CREATE TABLE F2 AS
 SELECT i AS i, j AS j, m AS m,
 count(distinct id) FROM F
 GROUP BY i, j, m;

CREATE TABLE bc AS
 SELECT F2.m AS m,
  sum(F2.count/F1.count) AS bc
 FROM F2
 JOIN F1 ON F2.i=F1.i
  AND F2.j=F1.j
 GROUP BY F2.m;
```

IV. Optimizing Queries

A. Optimization strategies for all solutions

1) Data Encoding and Compression: Columnar DBMSs uses encoding and compression to optimize query performance and save storage space. Data Encoding converts data into a standard format and increases performance because there is less disk I/O during query execution. There are several encoding strategies used in columnar DBMSs depending on data type. It also passes encoded values to other operations, saving memory bandwidth. Compression is fundamental in a columnar DBMS. It transforms data into a compact format. Compression allows a column store to occupy substantially less storage than a row store. In general, the efficient storage methods that columnar DBMSs uses makes it possible to maintain and process more data in physical storage. These optimizations are done by the system itself.

2) Projections: Projections store data in a format that optimizes query execution. Different from row DBMSs, columar DBMSs store the actual data in projections. When data is loaded into a table, the system creates or updates column-store projections. Then when a query is submitted, the query optimizer automatically assembles a query plan and choose a more efficient projection to process the query according to the properties of projections. So the columnar DBMSs could optimize data access.

3) Partitioning: Partitioning divides one large table into smaller pieces based on values in one or more columns. It improves the performance of queries whose predicate is included in the partition expression. This optimization is very effective when processing data in parallel. The graph should be partitioned in such a way that reduce uneven data distribution and costly data movement across the network which makes the parallel join occurs locally on each worker node possible. So partitioning provides opportunities for parallelism during query processing.

We perform the partitioning by vertex in our solutions. All the neighbors of a vertex are stored on the same machine. We assume the graph does not contain one high degree vertex but it might contain a few high degree vertices.

4) Duplicate Elimination: The operation of $\bowtie$ produces many duplicate edges especially for dense graphs. Here We push aggregation to eliminate duplicate edges by grouping rows. Also we avoid cycles in the path using HAVING $R.i! = E.j$. In diameter solution, a GROUP BY aggregation on R_k was performed after each iteration of join, grouping rows by edge with the grouping key i, j.

B. Specific Optimization Strategies for Diameter and Betweenness Centrality

Both diameter and betweenness centrality algorithm need to do recursive join. It will take a long time for large graphs because of fast growing table size. So, we introduce several optimization techniques to reduce the table size in the recursive join process.

Only the shortest path is needed in both algorithms. Unnecessary paths are kept even we did GROUP BY after each iteration. For example, the shortest path for vertex pair (i, j) exists in depth 2. But there is another path for this pair in depth 3. Obviously, the second path is not the shortest one, but it is still kept and explored. This increases both space and time complexity. To keep only the shortest paths, we maintain a table $S(i, j, v)$ which contains all the shortest paths found so far, and delete unnecessary paths from R_k after each iteration. At the same time, we update table S by the addition of new shortest paths. Since DELETE query is slower than the CREATE query, we created a new table instead of deleting unnecessary paths. The queries for diameter solution are presented below. And also the steps summary is shown in Figure 3.

```
CREATE TABLE RT_K AS
 SELECT t1.i AS i, t2.j AS j,
    min(t1.v + t2.v) AS v
 FROM R(K-1) t1
 JOIN E t2 on t1.j=t2.i
 GROUP BY t1.i, t2.j
 HAVING t1.i!=t2.j;


CREATE TABLE  RK  AS
 SELECT i, j, min(v) AS v
 FROM RT_k WHERE (i,j,v) NOT IN
      (SELECT RT_k.i, RT_k.j, RT_k.v FROM  RT_k , S
       WHERE RT_k.i =S.i AND RT_k.j = S.j
       AND RT_k.v > RT_k.v)
 GROUP BY i,j;
```

Algorithm 3: Our optimized SQL solution for diameter

$k \leftarrow 1$;
Create table E from source table ;
Create table S (stores all shortest paths found so far);
$R_1 \leftarrow \pi_{i,j,min(v)} E$;
while $|Rk| > 0$ **do**
  $k \leftarrow k + 1$;
  $RT_{k+1} = R_k \bowtie E$;
  //delete all the unnessary edges ;
  $R_{k+1} = RT_{k+1} \bowtie S$;
  // insert into table S all the shorest paths found in R_{k+1} ;
  update S by $S \bowtie R_{K+1}$;
end
diameter = $\pi_{max(v)} \mathrm{R}_k$;
return diameter ;

Fig. 3: Our algorithm using standard SQL queries for diameter.

```
INSERT INTO S
 SELECT  i,j,min(v) AS v
 FROM Rk
 GROUP BY i,j;
```

For betweenness centrality solution, the difference with diameter is that the intermediate vertices are kept. So GROUP BY was not allowed in this part. The optimized queries for betweenness centrality are shown below. The summary of betweenness centrality solution steps is shown in Figure 4. For k-betweenness centrality, for example $k = 4$, just need to change the line '$where|R_k| < 0$' to '$k <= 4$', all others are the same.

```
CREATE TABLE RT_k AS
SELECT nextval('sequ') AS id, k AS d,
    t1.i AS i, t2.j AS j, t1.v+t2.v AS v,
    t1.m1 AS m1,t1.m2 AS m2,...
    t1.m(n-2) AS m(n-2), t2.i AS m(n-1)
FROM R(k-1) t1 JOIN E t2 ON t1.j=t2.i;


CREATE TABLE Rk AS
SELECT * from RT_K
WHERE (i,j,v) NOT IN
 (SELECT RT_k.i, RT_K.j, RT_K.v
  FROM  RT_k , S
  WHERE RT_K.i =S.i
  AND RT_K.j = S.j AND RT_k.v > S.v);


INSERT INTO S
 SELECT  i,j,min(v) AS v
 FROM Rk
 GROUP BY i,j;
```

C. Time Complexity

The most challenging part to compute betweenness centrality and diameter is performing join multiple times. The basic operation of iteratively perform is to multiply E by itself: $E \cdot E ... E$. For the graphs stored in a database, $E \cdot E$ is matrix-matrix multiplication. The shape, density and connectivity of a graph will impact the complexity of join. Time and space complexity was analyzed for iterative matrix-matrix multiplication regarding different graph structures in the original and optimized solution.

Algorithm 4: Our optimized SQL solution for betweenness centrality

$k \leftarrow 1$;
Create SEQUENCE;
Create table E from source table ;
Create table S (stores all shortest paths found so far;
$R_1 \leftarrow \pi_{i,j,min(v)} E$;
while $|Rk| > 0$ **do**
  // $k \leftarrow k + 1$;
  $RT_{k+1} = R_k \bowtie E$;
  $R_{k+1} = RT_{k+1} \bowtie S$;
  update S by $S \bowtie R_{K+1}$;
end
create table U ;
for $c \leftarrow 1$ *to* k **do**
  //change all the intermediate columns of table Rk to rows;
  R_k to R'_k;
  U = U $\cup \mathrm{R}'_k$;
end
//table F contains all the shortest path for eash vertex pair and also intermeidate vertices ;
create table F by $U \bowtie S$;
$F1(i, j, \varphi_{ij}) = \pi_{i,j,count(id)}(F)$;
$F2(i, j, m, \varphi_{ij}(m)) = \pi_{i,j,m,count(id)}(F)$;
table bc = $\pi_{F2.m,sum(\varphi_{ij}(m)/\varphi_{ij})}(F1 \bowtie_{F1.i=F2.i \ and \ F1.j=F2.j} F2)$;
return table bc ;

Fig. 4: Our algorithm using standard SQL queries for betweenness centrality.

First, we will focus on the $O()$ of $|R_2|$. For a tree graph, $|E \bowtie E| = O(n)$ since it was necessary to exclude the leaf nodes and the parents of leaf nodes. For a complete graph, $|E \bowtie E| = O(n^3)$ as there are $n(n-1)$ pairs of vertices, and there are $n - 2$ paths for each pair. The ultimate goal was to understand $|R_k|$, where $R_k = E \bowtie E \bowtie ... \bowtie E$. To get the betweenness centrality value, it was necesssary to go through k depth where $k >= p$, p is the longest shortest path for graphs(p equal to d for directed and connected graphs). The worse case is when the graph is a list, then $p = O(n)$. So p is the second aspect impacting $|R_k|$. And $|R_k|$ grows exponentially as k grows.

The optimization of betweenness centrality and the second optimization of diameter keep the shortest paths at each depth. Assume the number of shortest paths for each pair is 1, So the total number of shortest paths for every pair is $O(n^2)$. Then $|R_k| = O(n^2)$.

For the time complexity of the join operator, it can range from $O(m)$ to $O(m^2)$ in each iteration. We checked the query plan in the DBMS, and it always used a hash join. The time complexity of the hash join could be as bad as $O(m^2)$ for a very dense graph but is $O(m)$ on average. For the second optimization method of diameter, since a logarithmic join is performed, there are d times join in the original solution, that is at most $\log_2(d) + 1$ times join in the second optimization method.

TABLE I: Summary of data sets.

Dataset	Type	n	m	Skewed Vertices
tree10m	Synthetic	10M	10M	Low
cliqueLinear100k	Synthetic	2.3K	100k	High
wiki-vote	Real	8k	103.6K	Low
webgoogle	Real	875K	5.1M	Low

TABLE II: Experimental proof for Diameter

Data Set	m	n	Python	DBMS	Relative error(%)
denseWebgoogle	37.3k	7.4k	4	4	0
sparseWebgoogle	22.3k	5.6k	11	11	0
subWikivote	34k	3.6k	5	5	0
cliqueLinear100k	100.2k	2.3k	133	133	0

V. Experimental Evaluation

In this section, we propose experimental validations of our algorithms. First, an overview of the experimental setup and benchmark data sets are presented, followed by accuracy validation, evaluation of the impact of optimization and the performance of optimization of betweenness centrality algorithm.

A. Experimental Setup

1) DBMS Software and Hardware: All the systems were run on eight node clusters that each had an Intel Pentium(R) CPU running at 1.6 GHz, 8 GB of RAM, 1TB disk, 224kb L1 cache, 2MB L2 cache and running Linux Ubuntu 14.04. For the experiments conducted in parallel computation, the total RAM size is 64 GB, and total disk memory is 8 TB. We used the Vertica DBMS supporting ANSI SQL to execute our queries. However, our queries are standard SPJ queries and work on other DBMSs too. We used Python as the host language to generate SQL queries and submit the queries to the DBMS as it is faster than JDBC.

2) Data Sets: Both synthetic and real graph data sets were used for experimental evaluation. For synthetic graph data sets, graphs were generated with varying complexity. Generated graphs with varying clique sizes used a uniform distribution where clique sizes increased linearly. The Stanford SNAP repository was used for real data sets. The data sets are listed in Table I. All the time measurements in this section are taken as the average of running each query five times and excluding the maximum and minimum values.

B. Accuracy Validation

In this section, we show our solution is computationally accurate since the results of our solution for different types of data sets both in Python and DBMS were identical.

1) Diameter: There is a function incorporated in Python-NetworkX ($diameter(G, e)$) that gives diameter values. Since Python runs on single node and only works for simple and small graph, we selected two types of subsets from synthetic and real graphs to do the accuracy validation: graphs with larger indegree and graphs with small indegree. For the dense subsets, we selected a vertex with large indegree first, then we extended the graph by choosing all the vertices connected to this vertex and repeated this process several times. For the sparse subsets, we selected one vertex whose indegree is only 1 and then extended the graph as what we did for the dense graph. Finally for the other subsets, subWikivote, we randomly selected one vertex and then expanded it as sparse and dense subsets. The comparison results are showed in Table II along with the details of the data sets. We can see that the diameters given by our solution are the same as given by the Python function.

2) Betweenness Centrality: The definition of betweenness centrality could extend naturally to directed or disconnected graphs [6]. Along with the previously used subsets, another subset was added that is a disconnected subset named sparseWebgoogle2. This was created by randomly selecting some vertices whose indegree were only 1. A function ($betweenness_centrality(G, k, normalized, weight, endpoints, seed)$) was incorporated in Python-NetworkX that gave the shortest-path betweenness centrality for vertices. The maximum relative error among all the vertices in the test graphs is shown in table III. The depth was set equal to the length of the longest shortest path to get the correct betweenness centrality for each vertex. From table III, it can be seen that the relative errors of our algorithm are lower than 0.0003 for all the data sets. The relative error results from rounding, different adding orders when adding a very sparse number with a small number.

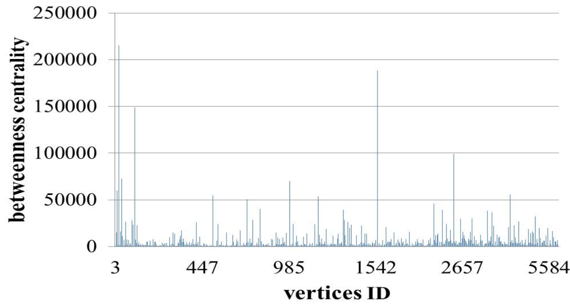
TABLE III: Experimental proof for Betweenness Centrality.

Data Set	m	n	Depth	relative error
denseWebgoogle	37.3 k	7.4 k	22	1E-9
sparseWebgoogle	22.3 k	5.6 k	5	0
sparseWebgoogle2	100 k	153.4 k	5	0
subWikivote	34 k	3.6 k	5	1E-05
cliqueLinear100k	100.2 k	2.3 k	133	0.0003

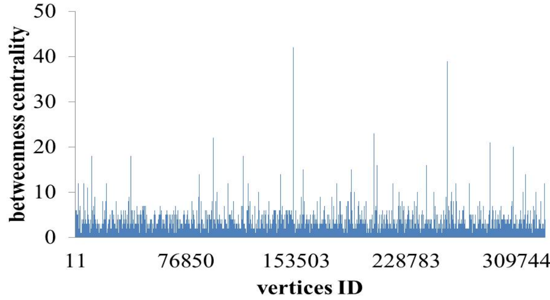
Fig. 5 shows the detail of betweenness centrality results for different data sets. If a vertex's betweenness centrality value is 0, then this vertex id is not shown in the figures. We already know that subWikivote is a tree structure graph. While leaf nodes of a tree have 0 value of betweenness centrality, the root has the maxium value. As the depth of the tree increases, the number of nodes significantly increases, while the betweenness centrality for these nodes significant decreases. This is exactly the situation demonstrated in Fig. 5a. For the sparseWebgoogle2 dataset, the difference of betweenness centrality among all the vertices are very small, and most of the vertices have low betweenness centrality. Fig. 5b demonstrates sparseWebgoogle2 is a sparse graph.

C. Evaluation of the Impact of Optimizations

1) Diameter: In this part, we will compare the optimized algorithms with the original one for diameter. Table IV shows the comparison results. We put a "Stop" sign in the table if



(a) subWikivote graph



(b) sparseWebgoogle2 graph

Fig. 5: The Betweenness centrality values for different data sets.

TABLE IV: Evaluating the impact of optimization for Diameter (time in seconds)

Data Set	with	without	diameter
tree10m	350	Stop	22
subWebgoogle4	1750	Stop	22
denseWebgoogle	165	170	4
cliqueLinear100k	246	1016	133

the time is more than 30 minutes. From the table, we could see that the optimized algorithm greatly reduced the costing time for different graphs.

2) Betweenness Centrality: We perform experiments both with and without optimization for betweenness centrality algorithm. To save time, we use k-betweenness centrality where k is less than the diameter. We put a Stop sign in the table if the time is more than 30 minutes. Table V shows that the solution with optimization takes almost the same time with the method without optimization for sparse and small graphs, but much less time is needed for dense and large graphs, because, the number of unnecessary paths is small for sparse graphs and large for dense graphs. Deleting those unnecessary paths, the performance for dense graphs was significantly improved. Based on the experiments, this is our default optimization for betweenness centrality algorithms. Fig. 6a shows the table size at different depths for the original solution and the optimized solution for complicated graphs. In Fig. 6a shows the table size grows dramatically with increasing depth in the original solution. The table size stays the same or only grows slightly in the optimized solution. $|R_k|$ in the original and optimized

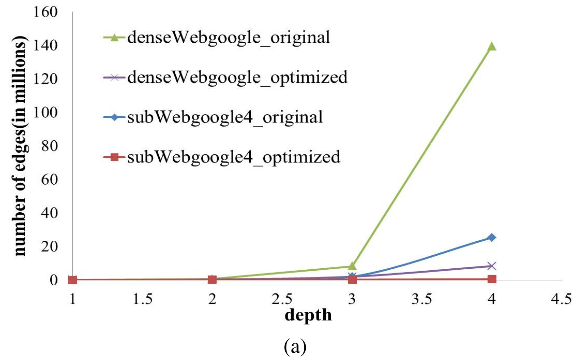
TABLE V: Evaluating the impact of optimization for Betweenness Centrality (time in seconds).

Data Set	without opt	with	depth
tree10m	Stop	1612	22
webgoogle	Stop	Stop	4
wiki-vote	Stop	1538	4
sparseWebgoogle2	6	9	5
subWikivote	Stop	345	5

TABLE VI: The average value of x for different data sets.

Data set	n	m	x in original	x in optimization
webGoogle	875K	5.1M	12	6
wikivote	103.6K	8k	43	16
subWebgoogle4	5.2k	18.4k	11	2.8
denseWebgoogle	7.4k	37.3k	15.5	6

solution could be expressed as $x^k n$ where k is the depth of join, and n is the number of vertices in the table E. The values of x for different graphs are listed in Table VI. $|R_k|$ grows much slower in the optimized algorithm when compared with the original solution, Table VI. So our optimization could largely reduce the table size. In general, our optimization works well for large and dense graphs.



(a)

Fig. 6: How table size changes when doing join in the original and optimized solution.

D. Performance Comparison

1) Diameter: We compare our algorithm with Python solution running on a single machine of the server, then with Spark running on the 8 node cluster. We use the same data sets as the accuracy validation part and the default optimization.

Table VII shows the results for both in single and parallel machines. From the table, we could see that our solution is faster than Python for all data sets except cliqueLinear100k as the number of joins required for this graph is very high. For parallel processing, there is a function incorporated in Spark-Graphx(shortestpath()) calculating the shortest paths of all pairs of vertices. Then we choose the longest shortest path as the diameter. From table VII, we can see that DBMS is faster than Spark for sparse graphs. And for dense or large graphs, DBMS could get results in a short time while Spark crashed during computation because of running out memory.

TABLE VII: Time to compute Diameter in Single and in 8-node Parallel machines(time in seconds).

	Single Machine		Parallel Machines		
Data set	DBMS	Python	DBMS	Spark	Diameter
denseWebgoogle	396	1472	165	Crashed	4
sparseWebgoogle	207	751	60	135	11
subwikivote	138	434	47	Crashed	5
cliqueLinear100k	600	305	246	Crashed	133

TABLE VIII: Time to compute betweenness centrality in single machine: DBMS vs Python(time is seconds)

Data Set	m	n	Python	DBMS
tree10m	10M	10M	Stop	3420
sparseWebgoogle2	100k	153.4k	Stop	3
denseWebgoogle	37.3k	7.4k	621	846
subWikivote	34k	3.6k	76	346

2) Betweenness Centrality: While Python can only give the value of betweenness centrality, Our solution can provide both betweenness centrality and $k-$betweenness centrality. To compare our solution against Python, the betweenness centrality was used. We use the same simple graphs as the accuracy validation part since Python would crash when running on big graphs. Table VIII shows the comparison results. A "Stop" was placed if one computation did not finished in 60 minutes. Table VIII shows that Python was faster than DBMS for small graphs. However, for large graphs, Python spent a lot of time on reading large number of vertices and edges into main memory while DBMS got the results in a very short time. DBMS gave results for sparseWebgoogle2 very quickly because there was few edges when doing join. So our solution works better for large graphs than Python. And also our solution could give k-betweenness centrality for large graphs which is also very important, but Python could not. So our solution is better.

In case of parallel processing, the available betweenness centrality solution in Spark was compared to our method. Both solutions were run on the eight node cluster. It was possible to compute $k-$betweenness centrality in Spark. So we use k-betweenness centrality to compare where depth k is 4,5,6. And also we use big graphs. Table IX shows the calculate time. "Crashed" was inserted when Spark crashed because it ran out of memory. From table IX, it can be seen that Spark was slower than DBMS for sparse graphs and crashed for dense and large graphs. DBMS computed $k-$betweenness centrality for both dense and sparse graphs in a reasonable time.

In summary of this section, we compared our proposed algorithms implemented with SQL queries in DBMS with popular existing systems and we proved that our algorithms are accurate and faster.

VI. Related Work

There are many applications related to graphs. Recent work on graphs offers a vertex-centric query interface to express many graph queries [12]. A novel method called core labeling was proposed to handle reachability queries for massive, sparse graphs [9]. Abughofa et al., 2018 studied processing dynamic graphs in real-time and proposed an end-to-end framework which allowed graph updates in real-time and supported efficient complex analytics in addition to online transaction processing (OLTP) queries [2]. However, querying from large graphs stored on a DBMS using relational queries has not received much attention. Malewicz et al., 2010 proposed a system named Pregel for large-scale graph processing [13]. Pregel is designed for sparse graphs where communication occurs mainly over edges, and the entire computation state resides in RAM. How relational database management systems (RDBMSs) can support graph processing at the SQL level was revisited in [16]. The authors proposed new relational algebra operations.

The diameter is an important parameter to understand the graph. And many attempts were made to seek efficient algorithms that approximate the diameter. In [8], two solutions were exhibited which could achieve a better approximation for the diameter, one running in $O(m^{3/2})$ time and the other running in $O(mn^{2/3})$. Other works learn about the approximation of the diameter when adding edges. But no now has developed an optimized SQL solution which works for big graph stored in a DBMS.

Betweenness centrality was introduced independently by Anthoisse and Freeman in [4], [10], many works have been done about making it faster. Brandes, 2001 developed a fast algorithm that runs in $O(n+m)$ on unweighted graph and $O(mn+n^2log(n))$ time on weighted graphs, where n is the number of vertices and m is the number of edges in the graph [6]. We developed our query solution of betweenness centrality according to Brandes algorithm. Recently, many works focusing on how to obtain rough approximations of betweenness centrality have been done. Bader et al., 2007 presented a novel approximation for computing betweenness centrality of a given vertex, for both weighted and unweighted graphs. The approximation algorithm was based on an adaptive sampling technique that significantly reduced the number of single-source shortest path computations for vertices with high centrality. The random sampling algorithm gave good betweenness approximations for biological networks, road networks, and web crawls. But all those works are about getting approximation of centrality since obtaining the exact value of betweenness centrality for all vertices are time consuming and also impossible due to RAM limitation. In our solution, we could get exact value of betweenness centraliy and also k-betweenness centrality.

VII. Conclusions

We represented the graph in terms of database perspective and stored them as a form of triplets. DBMSs are fast to load/import large external graph data set. SQL queries are good enough to compute complex graph metrics. They are efficient, elegant and short. In this work, we compute diameter and betweenness centrality of a large graph inside DBMS. We showed that DBMSs can indeed help us to compute something that is complicated than transitive closure and recursive

TABLE IX: Time to compute betweenness centrality in 8-node parallel machines: DBMS vs Spark (time in seconds)

	k=4		k=5		k=6	
Data set	DBMS	Spark	DBMS	Spark	DBMS	Spark
tree10M	80	Crashed	93	Crashed	112	Crashed
webgoogle	Stop	Crashed	Stop	Crashed	Stop	Crashed
wiki-vote	1538	Crashed	1790	Crashed	Stop	Crashed
sparseWebgoogle2	9	34	10	39	12	44
cliqueLinear100k	25	Crashed	29	Crashed	37	Crashed

queries. We expressed the computation of these algorithms with queries and proposed several optimization methods on the queries. Optimizing the query performance helps to compute the algorithms faster than usual. Our experimental results show that our queries perform better than Python in one machine and Spark-GraphX in parallel machines in most cases. We also provide accuracy proof to show that our solution is computationally accurate. However, one limitation of our proposed solution would be: too many join operations for betweenness centrality slows down the computation in a very dense graph.

For future work we have the following: computing circumference of a graph (longest cycle), detecting two or more disconnected subgraphs, checking if the graph contains cliques of size at least $k \geq 3$, counting maximal cliques, parallel speedup meaning how the queries perform when we vary the number of machines and discovering complex patterns beyond paths. Moreover, we plan to optimize the algorithms in Spark.

REFERENCES

[1] Abadi, D., Boncz, P., Harizopoulos, S.: Column oriented database systems. PVLDB **2**(2), 1664–1665 (2009)

[2] Abughofa, T., Zulkernine, F.: Sprouter: Dynamic graph processing over data streams at scale. In: DEXA(2). pp. 321–328 (2018)

[3] Al-Amin, S.T., Ordonez, C., Bellatreche, L.: Big data analytics: Exploring graphs with optimized SQL queries. In: Proc.DEXA Conference. pp. 88–100 (2018)

[4] Anthonisse, J.: The rush in a directed graph. Stichting Mathematisch Centrum. Mathematische Besliskunde, No. BN 9/71. pp. 163–177 (1971)

[5] Bertolucci, M., Lulli, A., Ricci, L.: Current flow betweenness centrality with apache spark. In: Algorithms and Architectures for Parallel Processing. pp. 270–278 (2016)

[6] Brandes, U.: A faster algorithm for betweenness centrality. Journal of Mathematical Sociology pp. 163–177 (2001)

[7] Cabrera, W., Ordonez, C.: Scalable parallel graph algorithms with matrix-vector multiplication evaluated with queries. Distributed and Parallel Databases **35**(3-4), 335–362 (2017)

[8] Chechik, S., Larkin, D.H., Roditty, L., Schoenebeck, G., Tarjan, R.E., Williams, V.V.: Better approximation algorithms for the graph diameter. In: Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 1041–1052 (2014)

[9] Chen, Y.: On the evaluation of large and sparse graph reachability queries. In: DEXA. pp. 97–105 (2008)

[10] Freeman, L.C.: A set of measures of centrality based on betweenness. Sociometry pp. 35–41 (1977)

[11] Idreos, S., Groffen, F., Nes, N., Manegold, S., Mullender, K., Kersten, M.: MonetDB: Two decades of research in column-oriented database architectures. IEEE Data Eng. Bull. **35**(1), 40–45 (2012)

[12] Jindal, A., Rawlani, P., Wu, E., Madden, S., Deshpande, A., Stonebraker, M.: Vertexica: Your relational friend for graph analytics! Proc. VLDB Endow. **7**(13), 1669–1672 (Aug 2014)

[13] Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. pp. 135–146. ACM (2010)

[14] Megid, Y.A., El-Tazi, N., Fahmy, A.: Using functional dependencies in conversion of relational databases to graph databases. In: DEXA(2). pp. 350–357 (2018)

[15] Ordonez, C., Cabrera, W., Gurram, A.: Comparing columnar, row and array dbmss to process recursive queries on graphs. Information Systems **63**, 66–79 (2017)

[16] Zhao, K., Yu, J.X.: All-in-one: Graph processing in rdbmss revisited. In: Proceedings of the 2017 ACM International Conference on Management of Data. pp. 1165–1180. SIGMOD '17 (2017)