Omni Graph Mining: Graph mining using RDBMS

Jin Kyu Kim
Dept. of Computer Science
CMU
jinkyuk@cs.cmu.edu

Alex Degtiar
Dept. of Computer Science
CMU
adegtiar@andrew.cmu.edu

Jay Yoon Lee
LTI & LCCB
CMU
jaylee@andrew.cmu.edu

December 8, 2012

Abstract

In this study, we compare three approaches to implement graph algorithms - SQL, MapReduce, and GraphChi - in terms of performance and cost efficiency. Since MapReduce was introduced, it has been considered as a standard tool to analyze large scale data including graph data. However, we believe that there is a 'sweet spot' in which RDBMS can outperform MapReduce. We think that the performance of these approaches is affected by the size of data and the kind of graph algorithm. The main contribution of this study is to find the most efficient approach for a given data and algorithm.

1 Introduction

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 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: PageRank, connected component, diameter, RWR, degree distribution, eigenvalues, and belief propagation. This information is essential for the companies to predict the behaviors of their customers, and to optimize the use of resources. In this paper, we compare three approaches of graph data analysis: SQL, MapReduce, and GraphChi. Since Google introduced the MapReduce framework in 2006, it has been considered the de-facto standard tool

for large data analysis including graph mining. However, we think that the MapReduce framework is not a single silver bullet to do graph analysis on all kinds of data in terms of performance. We compare the runtime of six graph mining algorithm on three different approaches with varying size of graph data, with a focus on SQL.

The problem we want to solve is the following:

- GIVEN: A set of graph data with different number of edges and vertices.
- FIND: For a given algorithm, and size of graph data, find a best approach among SQL, MapReduce, and GraphChi.
  In modern IT society, companies spend a large amount of money for data management. Sometimes, industry people are blind at the time when they choose a framework for their daily graph data analysis job. Each company might have different size of graph data to analyze. The size of graph data determines which approach is the fastest and most cost-efficient. Through this study, we can give a guidance to the industry. The results of our study will help them select the most performance- and cost- efficient approach for analyzing their graph data.
  The contributions of this project are the following:
  - Our study helps people to select the most efficient approach to analyze a given graph data.

2 Survey

Next we list the papers that each member read, along with their summary and critique.

2.1 Papers read by Jin Kyu Kim

The first paper was "A comparison of Approaches to Large-Scale Data Analysis" by *Pavlo et al.*[1].

- *Main idea*: In this paper the authors compare two large scale data analysis approaches, parallel RDBMS and MapReduce. In the opposition to the public belief that MapReduce is a kind of only one solution that enables large-scale data analysis, they show that parallel RDBMS with fine tuning outperforms MapReduce framework on large size data. They insist that MapReduce framework has several disadvantages compared to parallel RDBMS: initial overhead, contention to disk for shuffling, and data passover between two consecutive invocations of MapReduce. If computation of map function is too small, initial overhead such as Java VM initialization is not amortized. Data analysis job is a collection of complex query that requires the iterative invocations of Map/Reduce. With M-Map tasks and N Reduce tasks, the MapReduce framework generates M*N intermediate files at the end of Map phase. And N files are sent to each reducer. This shuffling step causes the contention over disk as well as network traffic. The implementation of these complex query using MapReduce transfers a large amount of data between two consecutive invocations of Map/Reduce. Parallel RDBMSs have no such problems and well organized in-memory index help them overshadow MapReduce even for selection query.

- *Use for our project*: The experimentation method used in this paper would be a helpful reference when we do our experimentation although they run their experimentation on parallel DBMS. And their explanation about internals of Hadoop and RDBMS would might be useful when we analyze our results.
- *Shortcomings*: One major weakness of this paper is that they do not try real large size data - hundred TB or PB although their cluster is equipped with about 2PB storage space. I believe parallel RDBMS would outperform Hadoop even for that big data thanks to the highly optimized query execution and in-memory index. But the performance gap between Hadoop and RDBMS approaches would be smaller.

The second paper was "Pegasus: Mining Billion-Scale Graphs in the Cloud" by *UKang et al.*[2].

- *Main idea*: This paper introduce a library called GIM-V that can express the five essential graph algorithm. The library consists of three primitives that play a role of glue between graph algorithms and underlying MapReduce framework. Three primitives are combine2, combineAll, and Assign. All of these primitives are for matrix multiplication. In the paper, they implement five algorithms: degree, PageRank, connected component, RWR, and diameter. But the application of GIM-V is not limited to these five algorithms. As far as algorithm can be expressed by the combination of three three primitives, they can be deployed on MapReduce framework. Furthermore, they show that there exists SQL query equivalent to the xG: sequential execution of combine2, combineAll, and assign.
- *Use for our project*: Their implementation of graph algorithms using GIM-V are very good references for us. Theoretically, there exist SQL query corresponding to GIM-V. The things unclear is whether SQL engine allow user defined functions that emulates combine2, and combineAll function of each graph algorithm. As far as we search, PostgreSQL allows user-defined function, aggregate function, and bit-wise operations.
- *Shortcomings*:

The third paper was "HADI: Mining Radii of Large Graph" by *UKang et al.*[3].

- *Main idea*: This paper suggests an approximation method to measure the effective diameter of a given graph. The existing algorithms can find exact results but require O(n^2) space. And those algorithms assume that available memory size is enough to keep all data. But this assumption is not true for large scale graph data. In this paper, the authors apply the Flajolet-Martin algorithm. In their new algorithm, each vertex is given a random bit string encoded by FM algorithm. In each iteration, a vertex receive values from all neighbor vertices and replace its own value with new value if a new value from the neighbors is larger than its own value. Then, a vertex propagate its value to all other neighbors through outgoing edge. The algorithm terminates when there is no vertex that needs to change its value. This algorithm give an approximate results but has only O(log(n)) space complexity.
- *Use for our project*: HADI algorithm would be our choice for diameter algorithm. Thanks to the Bit-wise operation by PostgreSQL, we think we could implement HADI using SQL in a straightforward way.

- *Shortcomings*: The approximated results could be seriously skewed under a certain condition. If a graph has a extremely large number of vertices but is chunked into the large number of very small subgraphs, the results might be much different from the real property of the graph.

2.2 Papers read by Alex Degtiar.

The first paper was "Pig Latin: A Not-So-Foreign Language for Data Processing" by *Olston et al.*[4].

- *Main idea*: Pig is a system that supports a higher-level programming model for analysis of large-scale data than the base MapReduce framework provides. Its abstraction rests at a higher, more declarative level than MapReduce, yet at a more procedural and often intuitive level than SQL. Its performance is currently not much worse than raw MapReduce, yet allows for much easier development and maintenance of large-scale data analytics code than using Hadoop directly.
- *Use for our project*: Pig Latin's somewhat SQL-like syntax and simpler programming model will make it an easier platform on which to implement graph-mining algorithms than on one like Pegasus. It could potentially be used as an additional platform of performance comparison for the algorithms we are experimenting with, in future work.
- *Shortcomings*: Pig typically has worse performance than using the raw Hadoop interface, and so will likely result in worse performance on the graphing algorithms compared to the highly-optimized Pegasus system. Naturally, it does not have any graph mining primitives/algorithms built in, so we will attempt to efficiently implement the relevant graph mining algorithms within Pig.

The second paper was "Unifying guilt-by-association approaches: Theorems and fast algorithms" by *Koutra et al.*[5].

- *Main idea*: Guilt-by-association is a powerful technique in graph mining that classifies nodes via their associations to others, and is used for a variety of purposes. There are several guilt-by-association algorithms that each provide similar results: RWR, SSL, and BP. This paper compares these methods, suggests BP as a preferable algorithm, and proposes a FABP implementation, a fast BP algorithm built on top of MapReduce. This algorithm not only provides convergence guarantees, but is experimentally shown to be more efficient and as accurate or more accurate than the standard BP algorithm.
- *Use for our project*: RWR and BP are both algorithms we intend to implement within our survey of platforms. This paper describes implementation details of FABP, and to a less degree RWR.
- *Shortcomings*: While this paper provides comparisons of several guilt-by-association methods, it does not address the variety of platforms/implementations these can be built on. We will extend the range of comparisons to more algorithms and more platforms.

The third paper was "Patterns on the connected components of terabyte-scale graphs" by *UKang et al.*[6].

- *Main idea*: Finding and analyzing connected component of large graphs has a variety of practical applications. This paper explores models and techniques for studying patterns among connected components in large graphs. It also looks at the patterns by which connected components dynamically grow, as well as those in static graphs. It proposes a number of models and measures that can be used to analyze connected components, including GFD, the 'rebel' probability, and CommunityConnection.
- *Use for our project*: Connected components in large-scale graphs are one of the areas this project will address. We will be looking only at connected component analysis in static graphs, so while many of the models are not directly useful for the project they can help with general understanding.
- *Shortcomings*: While discussing a variety of models and techniques, the paper does not address the challenges of implementation and performance in the analysis of graph components, which is something our project aims to do.

2.3 Papers read by Jay Yoon Lee

The first paper was "Spectral Analysis for Billion-Scale Graphs: Discoveries and Implementation" by *UKang et al.*[7].

- *Main idea*: This paper introduces HEIGEN, an accurate, and highly scalable eigensolver algorithm for large graphs based on MAPREDUCE environment of HADOOP. Top k eigenvalues are useful in analyzing the graph properties such as number of triangles and near-cliques. However, eigensolvers suffer from subtle problem as convergence for a large sparse graph. The paper overcomes these difficulties by selective-reorthogonalization, selective parallelization and blocking and shows its effectiveness and scalability by actually testing on several real-world graphs that are one of the largest publicly available graphs.
- *Use for our project*: The paper suggests ways to reduce or distribute computations by selectively separating them. This separations will be motivation for our team in designing sql codes for eigensolver of large graph. Additionally, this shows how top k eigenvalues can be applied in collecting many graph features and how useful these collected features (triangle count, degree, and combination of those) can be in graph analysis.
- *Shortcomings*: One shortcoming of this algorithm is that it is confined to symmetric matrix. Second shortcoming is that although the selective scheme of HEIGEN motivates division of computations, its high efficiency is under MAPREDUCE environment and it is not certain whether the same scheme would be useful in designing the eigensolver under SQL environment.

The second paper was "GBASE: A Scalable and General Graph Management System" by *UKang et al.*[8].

- *Main idea*: This paper proposes GBASE, a scalable and general graph management system. GBASE includes graph storage method called 'block compression', to efficiently store homogeneous regions of graphs, and seven different types of graph queries using matrix-vector multiplications. This form of matrix-vector multiplication allows query optimization using MAPREDUCE algorithm using the original adjacency matrix without explicitly building the incidence matrix.
- *Use for our project*: Although, storage is not the main part of our Omni-Graph-Mining (OGM), as the graph sizes are getting larger and larger storing graph and reading is becoming burdensome. Therefore as the GBASE paper presented 50x less storage and faster running time, this concentration on homogeneous regions of graph can also be beneficial to our relational database project.
- *Shortcomings*: Block compression would be most utilized when the graph size is big. There should be some burdens, in form of computation and complexity in the code, in block compressing. And at this point, since OGM is targeting mid ranged graph sizes, it is not certain whether this tradeoff is worth implementing for the range of database size that OGM will support.

The third paper was "GraphChi: Large-Scale Graph Computation on Just a PC" by *Kyrola et al.* [9].

- *Main idea*: This paper proposes GraphChi, a disk-based system for computing billion scale graphs. GraphChi enables persistent storage as a memory extension with the novel proposed method of Parallel Sliding Windows (PSW). Basically, PSW loads a subgraph from disk, updates the vertices and edges, and writes the updated values back to disk. To do so, PSW splits the graph into disjoint intervals and associates it with shards. Shards have two types: memory shards and sliding shards. This combination of shards enables parallel updates. PSW can process a graph with mutable edge values efficiently from disk, with only a small number of non-sequential disk accesses and sequential disk block transfers. These properties allow it to perform well on both SSDs and traditional hard disks.
- *Use for our project*: GraphChi is a direct counterpart to our proposed system OGM. By reviewing the surveys and considerations this paper went through, we can design our experiment more carefully. Also, this paper suggests weaknesses of many distributed computational approaches in graph mining, which are equally relevant to our reasoning for using single machine.
- *Shortcomings*: Although GraphChi's preprocessing is I/O efficient, and can be done with limited memory, this preprocessing step needs to satisfy the following condition: the number of splits P that GraphChi makes is chosen such that the shard can be loaded completely into memory. In detail, the number of shards P is chosen such that the biggest shard is at most one fourth of the available memory, to leave enough memory for storing the necessary pointers of the subgraph, file buffers, and other auxiliary data structures. These preprocessing conditions could limit GraphChi's performance in terms of the maximum data size it can run or the running time.

3 Proposed Method

To compare the performance of three approaches of graph mining, we implement eight fundamental graph mining algorithms using SQL, GraphChi, and MapReduce. For the SQL implementation, we use PostgreSQL which is an open source RDBMS with more than 20 years history. PostgreSQL is already equipped with a lot of desirable features such as bit-wise operations, user-defined functions, and aggregates, in addition to ANSI standard features. It also features an advanced storage and search system with highly efficient indices and query optimization. With this engine behind the SQL-implemented algorithms, we hope to outperform a Hadoop-based implementation for a range of data sizes. We also expect the performance to be comparable to other single-machine implementations of the chosen graph algorithms. As a single-machine counterpart implementation, we choose GraphChi to run the chosen algorithms. Since we will be using the included algorithms (a subset of all the chosen algorithms), no actual implementation is involved. For the MapReduce implementation, we use the Hadoop-based PEGASUS implementation in the CMU Open Cloud, where 64 computing nodes are installed.

The algorithms we implement are below. These algorithms were chosen partly due to them already being implemented in PEGASUS, but also because they each represent fundamental and practical graph mining algorithms commonly used in the industry.

- PageRank: Measures the relative importance of nodes via the nodes that link to it.
- Random Walk with Restarts (RWR): Defines a relevance score between two nodes in a graph.
- Connected Components: Partitions the graph into groups of nodes mutually connected via their edges.
- Diameter: Measures the distance of the furthest-apart nodes in the graph.
- Degree distribution: The number of in- and out- edges among nodes of the graph. Figure shows the SQL code of degree distribution.
- Eigenvalues: Finds the eigenvalue decomposition of the graph's adjacency matrix.
- BP/FastBP: An efficient way to solve inference problems based on local message passing.

Detailed description of algorithms

3.1 Common Matrix Operations

One of the core reasons SQL-based graph mining is effective is because most of the algorithms can be distilled into a common set of matrix operations. Matrix operations are typically straightforward and intuitive to implement in SQL, because subsections of the matrix (columns, rows, cells) can be referenced concisely. Furthermore, when dealing with large and sparse matrices, these JOIN operations are efficiently executed in a highly-optimized SQL engine on the right subsection of the matrices and vectors. Thus, SQL is enough to effectively express graph mining operations for most purposes.

The common matrix operations we came across when implementing these algorithms were Saxpy vector update, vector dot product, vector scaling, matrix-vector multiplication, and matrix-matrix multiplication. We note their implementations below, which demonstrate how a few lines of SQL code can be enough to express many relevant graph mining primitives.

Algorithm 1: $Saxpy$ vector update, y = y + alpha*x

```
1: UPDATE y
2: SET y.val = y.val + alpha * x.val
3: FROM x
4: WHERE y.i = x.i;
```

Algorithm 2: $DotProduct$ vector dot product, y^T * x

```
1: SELECT SUM(y.val * x.val)
2: FROM y, x
3: WHERE y.i = x.i
4: GROUP BY x.i;
```

Scale

Algorithm 3: $VectorScale$ vector scaling, alpha * y

```
1: SELECT y.i, a * y.val FROM y;
```

Algorithm 4: $LPNorm$ Vector LP norm, $sum(abs(y)^P)^{1/p}$

```
1: SELECT SUM(ABS(y.val)^P)^1/p FROM y;
```

Algorithm 5: $MatVecMult$ Matrix-vector multipication, A*y

```
1: SELECT matrix.row, SUM(matrix.weight * vector.value)
2: FROM matrix, vector
3: WHERE matrix.column = vector.node
4: GROUP BY matrix.row;
```

Algorithm 6: $MatMatMult$ Matrix-matrix multipication, A*B

```
1: SELECT A.i, B.j, SUM(A.val * B.val)
2: FROM A, B
3: WHERE A.j = B.i
4: GROUP BY A.i, B.j;
```

3.2 PageRank and Random Walk with Restarts

PageRank is the famous algorithm that is used by Google to calculate relative importance of web pages. Although Google is using modified versions, the initial popular equation has the following form with page rank p:

$$p = Wp,\ W = (dA^T + \frac{1-d}{n}U)$$

The rest of variables are a damping factor d(typically 0.85), row-normalized adjacency matrix

A, where row indicates source and column indicates destination, and U, a matrix with all elements set to 1. As seen in the equation$p$ is the eigenvector of the matrix W. To calculate this value, we use power method, a method which iterates through matrix vector multiplication to find eigenvector associated with largest eigenvalue. As our $MatVecMult$ shows, SQL is advantageous for matrix vector multiplication when the matrix is sparse. However, U matrix which does not have any zero value element cause W to be non sparse and thus makes SQL to keep all the indices in the table. To utilize the merit of matrix vector multiplication in SQL, rather than using W matrix directly, we use $p_{next} = dA^T p_{current} + \frac{1-d}{n}E$ where E is vector with all elements set to one. **??** show the steps in detail.

Algorithm 7: $PageRank$

1: Bulk load of dataset to node and edge table
2: Initialize variables ($d = 0.85, max_iteration = 30, \epsilon = 10^{-9}$)
3: Build adjacency matrix A, initialize pagerank p
4: **PowerIteration**
5: **for** $k = 1 \rightarrow maxiteration$ **do**
6: $p_{next} = dA^T p_{current} + \frac{1-d}{n}E$ $(MatVecMul\&Saxpy)$
7: **end for**

Random walk with restarts or personalized pagerank can be done by simply using different vector E at the power iteration. Since E denotes the flyout probability from each node, if all the node fly out to a specific node v, then we say PageRank is personalized to v.

3.3 FastBP

Belief propagation is a popular message passing algorithm that provides solution on inference problems on graph. The influence in graph from neighboring node can be divided into two type. One is homophily, which means that nearby nodes have similar labels, and the other one is heterophily, which means the reverse. The page rank algorithm above is another example of graph algorithm that uses homophile.
Here, we implement $FastBP$, which was proposed by D. Koutra et al.[5], that linearizes belief propagation algorithm and thus enables utilization of $MatVecMult$, sparse matrix vector multiplication, on SQL. In order to linearize and use power iteration instead of matrix inversion, the following are conditions and equation apply to $FastBP$. For SQL procedure, we directly use $MatVecMult$ contrast to using $MatVecMul$ & $Saxpy$ together on $PageRank$. $FastBP$, matrix is still sparse because it only adds nonzero elements on diagonals and this only increases index for n elements for n by n matrix while $PageRank$ if we do not use $Saxpy$, the matrix becomes full without zero terms.

$$[\mathbf{I} + \mathrm{a}\mathbf{D} - \mathrm{c}^{'}\mathbf{A}]\mathrm{b_n} = \phi_\mathrm{h},\ \mathbf{W} = \mathbf{I} + \mathrm{a}\mathbf{D} - \mathrm{c}^{'}\mathbf{A}$$

$$a = 4h_h^2/(1 - 4h_{h_h}^2),\ c^{'} = \frac{2h_h}{1 - 4{h_h}^2}$$

$$h_h < \sqrt{\frac{-c_1 + \sqrt{{c_1}^2 + 4c_2}}{8c_2}},\ c_1 = 2 + \sum_i d_{ii} \text{ and } c_2 = \sum_i {d_{ii}}^2 - 1$$

Algorithm 8: $FastBP$

Bulk load of dataset to node and edge table
Iload prior ϕ and set $belief$ = ϕ and product vector p= ϕ
Build symmetric adjacency matrix $\mathbf{A}$ using JOIN
Build diagonal matrix $\mathbf{D}$ using aggregator SUM()
Calculate h_h using diagonal matrix $\mathbf{D}$ and set coefficients a, $c^{'}$ using h_h
Set $\mathbf{W} = \mathbf{I} + \mathrm{a}\mathbf{D} - \mathrm{c}^{'}\mathbf{A}$
PowerIteration
for $k = 1 \rightarrow maxiteration$ **do**
    p = $\mathbf{W}p$ $(MatVecMul)$
    $belief = belief + product$
end for

3.4 Eigenvalues and Eigenvectors

Eigenvalues and eigenvectors of a matrix are critical properties that many important algorithms depend on, such as SVD, spectral clustering, and tensor analysis. In the context of graph mining, eigenvalues and eigenvectors are used in algorithms such as anomaly detection and calculating triangles and cliques.

We use the Lanczos Selective Orthogonalization algorithm to generate k eigenvalue and eigenvector approximations, by using the intermediate values generated by the power method. The algorithm performs a quick check to estimate which intermediate vectors need to be re-orthogonalized.

Our implementation depends on the fact that the Lanczos algorithm is primarily made up of matrix and vector operations, which can be effectively expressed in SQL. The relevant core linear algebra operations we isolated and expressed in SQL include Saxpy vector update, vector (inner) dot product, vector scaling, matrix-vector multiplication, and matrix-matrix multiplication. Besides the core matrix operations, the rest of the algorithm was implemented iteratively using procedural language extension to Postgres: pgpsql and plpythonu. A summary of the full Lanczos-SO algorithm is listed below.

To improve performance, we added indices to the relevant tables after we initialized the data. Besides the default indices on any relevant primary keys, the one that made the most difference in running time was on step id of the basis vectors.

3.5 Connected components

Connected components of large graphs are another important property in graph mining. We implemented the Hcc connected component algorithm for large graphs in Postgresql to generate this labelling property. The algorithm is as follows:

Algorithm 9: $Eigensolver$

1: Bulk load dataset into adjacency matrix
2: Generate a new basis vector b
3: **for** i until max iteration (or until convergence) **do**
4: Generate a new basis vector b
5: Orthogonalize against previous two
6: Update tridiagonal matrix using scalar components of these vectors
7: Selectively orthogonalize against all previous vectors
8: **end for**
9: Perform a small eigendecomposition of the tridiagonal matrix to get eigenvalues and Q
10: Compute eigenvectors using the relevant portion of Q and the basis vectors

Algorithm 10: $ConnectedComponents$

1: Bulk load dataset into node table and edge matrix
2: Initialize each node with $componentID = self.id$
3: **for** i until max iteration (or until convergence), for each node **do**
4: Update $self.componentID$ to be the minimum of its own and its neighbors
5: **end for**
6: Optionally summarize by counting the number of nodes with each $componentID$

The key step of the algorithm, which is to perform the update, implemented in SQL:

Algorithm 11: $UpdateConnectedComponent$

```
1: UPDATE nodeComponent
2: SET nodeId=id, componentId=newCmptId
3: FROM (SELECT edges.targetId as id,
4: MIN(nodeComponent.componentId) AS newCmptId
5: FROM edges, nodeComponent
6: WHERE edges.sourceId = nodeComponent.nodeId AND (cmptId(edges.targetId) <
   nodeComponent.componentId)
7: GROUP BY edges.targetId) AS updatedComponents
8: WHERE nodeId=id;
```

The edges are treated as symmetric edges, so this operation is repeated again in the reverse direction as well. This tends to converage after a small number of iterations for real graphs.

- Diameter: Measures the distance of the furthest-apart nodes in the graph.

We implement the HADI algorithm[3] with SQL language. We do update Flajolet-Martin bit strings of each vertex using SQL join operation. Three index are built on id of vertex list, and src and dst of edge list for improving the performance of join operation. When the all FM bit strings are

stabilized, the algorithm stops. To support BIT-OR on multiple tuples, one user defined aggregate is created. Algorithm 1 shows the pseudo code for HADI. For the convenience of implementation with SQL, we eliminate all duplicated edges, insert self loops, and assign FM bit strings to each vertex using an external C program before running SQL code.

Algorithm 12: HADI Algorithm in SQL

1 Step1: Preprocessing by C-Code Eliminate all duplicated edges. Insert a self loop to each vertex. Extract a list of vertices and Assign 32 Flajolet-Martin bitstrings to each vertex. Store the list as a separate file.
2 Step2: Create a user defined function and an aggregate for OR operation on FM bit strings. orbitstring: user defined function that performs OR on two bit strings. This user defined function is used by mabp aggregate. mabp: user defined ggregate performs OR on the stream of bit strings from multiple rows. It use orbitstring function as a component function.
3 Step3: Create edge and vertex table and load data in SQL code. Create table E(dst, src) Create table V(vid, FMB[1 32]); Create table T(vid, FMB[1 32]); // tmp table Create index on E.dst Create index on E.src
4 Step4: Iterate updating V table that propagates bitstrings and OR operations within each subgraph.
5 For i=1 to MAXITERATION
6 Insert into T select E.src as vid, mabp(T.FMB[1]),..,mabp(T.FMB[32]); where E, V where E.dst = V.vid group by E.src;
7 Insert into V select E.src as vid, mabp(T.FMB[1],...,mabp(T.FMB[32]); where E, T; where E.dst = T.vid; group by E.src;
8 If all bit strings are stabilized, exit the for loop End Note: plpgsql extension is used for flow control.
9 Step 5: Calculate the estimated diameter using FM bit strings of each vertex. Step 6: Store the diameters of each vertex into output file.

Discussion about implementation of Diameter algorithm.

- Degree distribution: The number of in- and out- edges among nodes of the graph. The count of in- and out- edges are updated mainly by scan operation of SQL. Again, three indexes are built on the id of vertex list, and src and dst of edge list for the better performance. Algorithm 2 shows the SQL code of degree

distribution.

4 Experimental Result

Our main experimentation goal is to examine whether a RDBMS approach can outperform the Map/Reduce framework for a certain range of graph sizes. Therefore, the experimention mainly focuses on comparing the running times of OGM to those of the existing graph mining systems for

Algorithm 13: Degree Distribution Algorithm in SQL

1 Step1: Preprocessing by C-code Extract a list of vertices and store it as a separate file
2 Step2: Create edge and vertex tables and load data in SQL code. Create table E(dst, src) Create table V(vid, indegree, outdegree, total) Loading edge list file into edge table E; Loading vertex list file into vertex table V; Create index on the E.src column; Create index on the E.dst column;
3 Step3: Count In/Out degree of each vertex by SQL join operation.
4 Update V set indegree (select count(*) from E where E.dst = V.id); Update V set outdegree (select count(*) from E where E.src = V.id); Update V set total = indegree + outdegree;
5 Step4: Store V into output file.

the eight major graph mining algorithms: PageRank, connected components, eigenvalues, diameter, RWR, belief propagation, FastBP and degree distribution.

4.1 Existing graph mining systems

- PEGASUS
  PEGASUS is the popular graph mining system under the MapReduce platform. It is a peta-scale graph mining system that runs in a parallel, distributed manner on top of Hadoop. Hadoop is an open source implementation of the MapReduce framework, which was originally designed for web-scale data processing by Google.[1] Since PEGASUS is a tool that has been rigorously tested, we will compare the end result of OGM with that of PEGASUS.

- GraphChi
  GraphChi is a relatively new graph mining system that runs graph computations on a single machine, by processing the graph from disk (SSD or hard drive).[2] Although GraphChi is a relatively new graph mining system, we selected this system as our counterpart in single-machine approach in order to examine the effectiveness of an RDBMS approach. The off-the-shelf graph mining algorithms available for GraphChi are PageRank, and connected components. We will only compare these two algorithms for GraphChi. In GraphChi, the vertices are split into P intervals. Each interval is associated with a shard. One shard of an interval contains all the edges whose destination belongs to the interval. The edges in a shard are sorted on their src. Intervals are well balanced and P is large enough for each shard fit into main memory. Parallel Sliding Window method performs graph computation by processing small portion of one intervals at a time. At first, vertices for the sub interval are loaded into main memory and their edges are loaded into main memory from disk. The number of disk accesses for updating all vertices and edges in the sub interval is limited to P-1 block accesses due to the fact that out-edges for the updated vertices are stored cotiguously in the other P-1 shards.

[1] http://www.cs.cmu.edu/∼pegasus
[2] http://code.google.com/p/graphchi/

4.2 Design of experiment

4.2.1 Accuracy

The accuracy of OGM's algorithms will be tested over JUnit tests and comparison with PEGASUS.

- JUnit tests: To make a cohesive tool out of our different SQL algorithms, we are writing Java code to act as the base that users will interact with. This component can perform verification of input data, invoke PostgreSQL with the various pre-processing and algorithm SQL codes, and implement the UI. This will also let us use JUnit to test our algorithm implementations.
- Comparison to PEGASUS: Since OGM and PEGASUS have exactly the same functions, we will examine OGM's result to that of the well-tested PEGASUS. Among the eight algorithms we proposed to implement, only radius should be somewhat different since it employs Flajolet-Martin lemma.

4.2.2 Runtime

The experiment will compare running time of both the "preprocessing" and the "algorithm" phases of computation. We include the preprocessing time to make a fair comparison, since each graph mining system assumes different input files and each needs preparation steps. PEGASUS distributes files into the distributed file system, whereas GraphChi goes through the process of splitting graphs into subgraphs. Therefore we set the baseline input as the CSV file that consists of who-points-to-whom information.

4.3 Data

The experiment will be executed both on synthetic data and on real graph data. Since access to proper scales of datasets are not guaranteed, we will generate synthetic datasets for running time comparison and real graphs would be used to prove the practicality of the proposed OGM. The size variation will be focused on the edge numbers because the complexity of graph feature generation highly depends on the number of edges. Typically, the synthetic test dataset will consist of 10k, 100k, 1M, 10M, 100M, 1B, 10B number of edges.

4.3.1 Synthetic data

We are considering several graph generation algorithms for use as synthetic data in our performance benchmarks. We require realistic graphs that closely match the type of graphs representing real data. Real graphs have a number of properties that generators seek to mimic, including heavy tails in degree distribution, eigenvalues, and eigenvectors; small diameters, and power law distributions. Kronecker graphs fulfill these properties, and the Kronecker generator uses RMAT-like recursive deepening[10]. However, the number of edges generated may be somewhat higher than expected. The Graph500 graph generator is another option, a parallel Kronecker graph generator

variant similar to the RMAT scale-free graph generation algorithm.[3] One potential issue with this method is that it may generate fewer triangles than expected. A third option we are considering is the Python Web Graph Generator, a variant of the RMAT algorithm for generating scale-free network (power law) graphs.[4] We are surveying these tools and will select the Graph500. We generate synthetic data set with -R e16 varying N from 15 to 26. The Graph500 would generate (2 to N)*16 edges with up to 2 to N vertices. But, the Graph500 generates small number of duplicated edges between two vertices and self loops, and some vertices does not appear in the edgelist. Therefore, we eliminate dulpicated edges and selfloops in preprocessing step before running our SQL code for the case that algorithm requires elimination of duplicated edges.

This will complement the real graph data and fill any potential gaps of graph size in the real data set. The generated data can also be compared to the real data to indicate how realistic we can expect it to be.

4.3.2 Real data

For real datasets, we use YahooWeb, Twitter, LinkedIn, and Patent. These are well-known graph data and therefore lets us easily compare with other groups if needed. The detailed description of each dataset is given under Table 1

Data	# Node	# Edge	Description
Patent	6 M	16 M	patent-patent
Stack Overflow	0.4M	3.3M	asker - answerer

Table 1: Summary of real data to be used in the experiment, M: Million

4.4 Environment

4.4.1 Single machine

For a single machine, we run the experiments on a machine with 2 dual-core Intel Xeon 3.00 GHz, 16 GB memory and 480 GB hard disk, running Red Hat Linux 4.1.2.

4.4.2 Open cloud

Open cloud, provided by parallel data laboratory, consists of 64 worker nodes, each with 8 cores, 16 GB DRAM, 4 1TB disks and 10GbE connectivity between node. This cluster is equivalent to a single computer, has over 1 tera-operations per second, over 1 TB memory, 256 1TB disks, and over 40 Gbps bisection bandwidth.[5]

[3] http://www.graph500.org/specifications

[4] http://pywebgraph.sourceforge.net/

[5] http://wiki.pdl.cmu.edu/opencloudwiki/Main/ClusterOverview

- Advantages of SQL in RDBMS - Low storage overhead for sparse matrix vector/Matrix computation - SQL implementation is very concise. - Good flow control primitives (PLPGSQL) by PostgreSQL - RBDMS engine maintains index structure automatically.
- Disadvantages of SQL in RDBMS - Optimization is not clear with hidden procedures. - Debugging is not easy. - Loading data from a disk into tables of RDBMS usually takes longer latency than loading data from a disk into main memory data structure of C/C++ applications. - Updating a table might incur longer delay than updating data structure of C application because of ACID features of RDBMS.

4.5 Comparison with Pegasus on CMU OpenCloud Hadoop Cluster

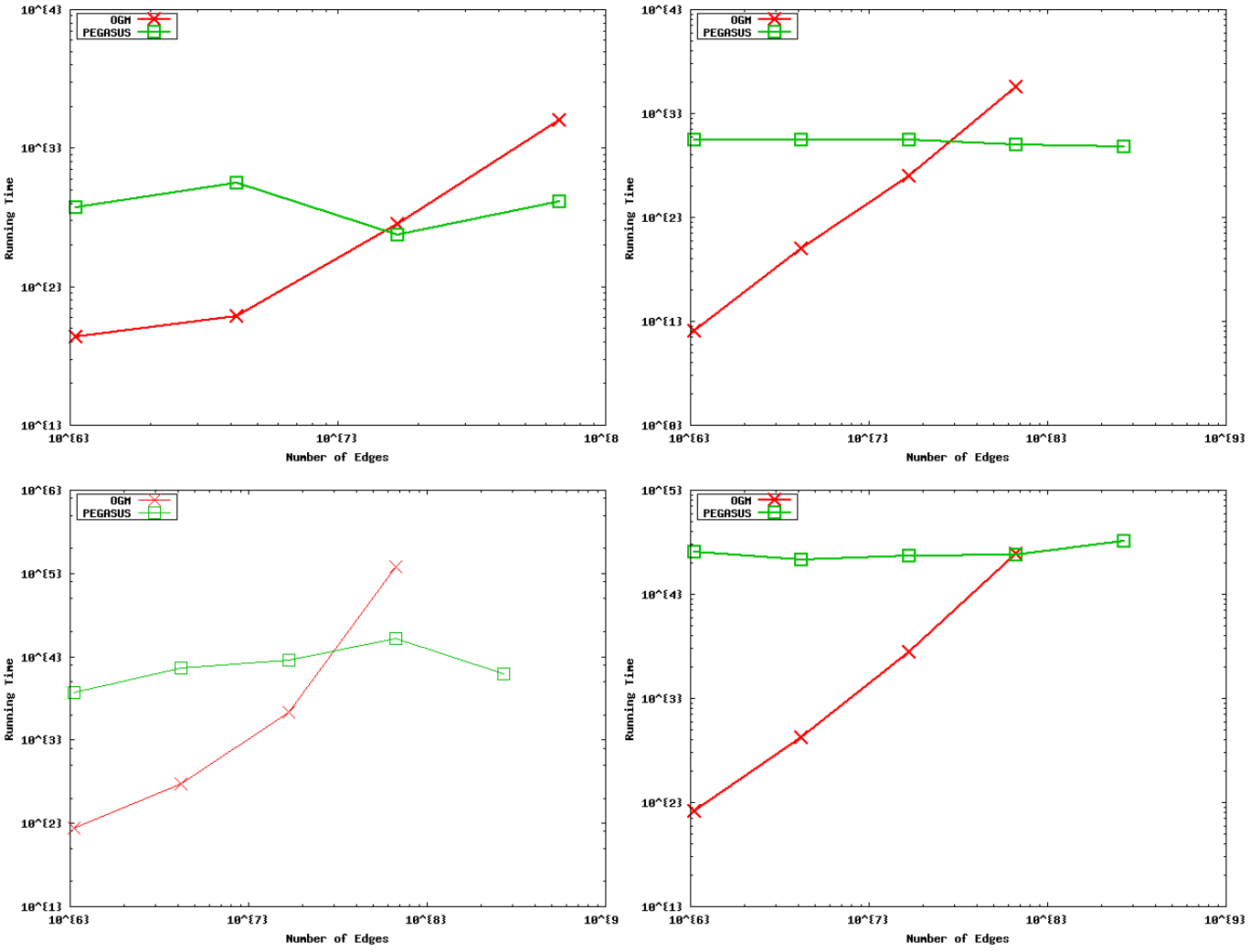
To compare OGM with Hadoop, we run sample code provided by Pegasus library package. Experimentation was performed on CMU OpenCloud. Figure AA depicts performance comparison. For all six algorithms, OGM running on single machine outperforms Pegasus running on 64 core cluster for the data sets that have less than 16 million edges. We try to analyze what factors determine the location of this sweet spot. Based on our observation that CPU utilization of OGM was very low for the data set with 64 million edges while there was no such phenomena observed for the data set with 16 million edges, we hypothesize that OGM run out of memory for the 64 million edges. Though the size of 64 million edge data set is still smaller than 2GB buffer size we configured, we think that index structure and other internal memory use of PostgreSQL cause this IO threshold even before the size of input data reaches the buffer size. We left the verification of our hypothesis as a future work. We think the reason that Pagasus's performance is lower than OGM for the small data sets is that the base overhead of scheduling job and creating Java instance on the cluster machines is not amortized due to the small size. Furthermore, we think that the small data may limit the number of machines running map phase since only the small number of computers hold a partition of input data.

4.6 Effects of RDBMS tuning

To show the effects of tuning RDBMS configurations, we run degree distribution algorithm with default option where fsync is enabled for keeping consistency, and shared buffer size is 32MB. Our machine has 16GB main memory, so that we configure buffer size to 2GB and turn off fsyns option with other few dependent options for better performance. Disabling fsync compromise the reliability and consistency of RBDMS. However, we think that the consistency constraint is not madatory for running graph mining if we do design SQL code carefully. For example, if OGM does not modify data stored in RDBMS and use a temporary repository for running OGM code, disabling fsync would not hurt the consistency of DB. The tuning accelerates the performance by up to 4.2 times.

4.7 Comparison with GraphChi on a single machine

We compare OGM with GraphChi running on a single machine. Figue 2 shows that GraphChi is more than ten times faster than OGM even for a small data set. It's mainly because of the way



(a) Degree (b) Connected Components (c)Pag

Figure 1: Running time comparison (in seconds) OGM and PEGASUS. OGM is faster for smaller sizes but the run time complexity is higher than PEGASUS. OGM run time crosses PEGASUS runtime between edge count of 16 million to 64 million.

of updating graph in GraphChi and the fact that their implementation is more efficient for running graph mining algorithms since GraphChi design does not consider ACID properties at all. In GraphChi, input graph data are sorted and sharded. Run time update is done on one subgraph at a time. Once all necessary input data is loaded into main memory for updating one subgraph, GraphChi does not incur disk access until it finish updating the subgraph. Furthermore, the preprocessing of sharding and sorting limit the number of disk access to P - 1 where P is the number of intervals. By choosing right configuration of intervals, the demand of main memory for updating one subgrap is always kept below the available memory size. And asynchronous disk I/O that overlaps disk I/O overhead with computation hide disk IO latency successfully.We think some of these configurations can be applied to OGM orthogonally. For example, pre-sorting edge table on destination and vetex table on id would improve cache locality of join operation.

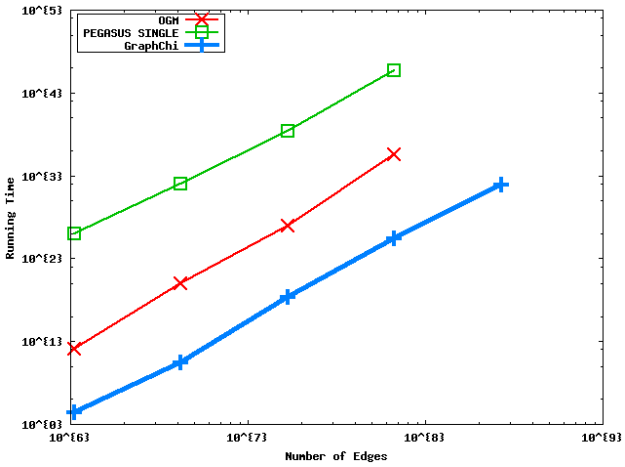


Figure 2: Runtime (log, log scale) comparison of OGM to single machine counterparts: GraphChi and PEGASUS. Runtime complexity of three system is similar with OGM in between.

5 Conclusion and Future work

There are lot of graph data stored in RDBMS. Graph mining with external mining tools requires data transfer between RDBMS and the external tools and sometimes requires data format conversion. At the end of processing, the results have to be sent back to the RDBMS again. To avoid these overheads, we run the mining algorithms in RBDMS without moving data. We implemented six graph mining algorithms such page rank, diameter, degree distributions, eigen values/vectors, fast belief propagation and connected component algorithms using SQL language. We showed that SQL is indeed expressive enough to implement the core components of graph mining, and we run SQL implementations using PostgreSQL on single machine. Our experimentation shows that the Postgres SQL engine can efficiently execute the graph mining algorithms for sparse graphs, and beat PEGASUS for smaller graph sizes. We also found the approximate border for graph size where a SQL-based implementation becomes preferred to PEGASUS.

Building and keeping indices help accelerate the algorithm, but consume substantial amount of main memory. Even though the size of input data including edgelist and vertex list is much smaller than the size of RDBMS buffer, the performance is limited by frequent disk I/O. Based on this observation, we conclude that the size of graph data that SQL engine handles well is much smaller than the size of buffer of RBDMS.

Future work contains further optimization of RDBMS engine and preprocessing the input data before run graph mining algorithms. In this context, the optimization is to make RDBMS compromise consistency and reliability for better performance. Because we do not care about the consistency and reliability of imtermediate status of our algorithms, it's not harmful.We think that GraphChi's idea of presorting edge and vertex lists is applicable to RBDMS. We think that sorting edge list on src and sorting vertex list can increse the data locality during join operation.

6 Reference

- [1] A. Pavlo, A. Rasin, S. Madden, M. Stonebraker, D. DeWitt, E. Paulson, L. Shrinivas, and D. J. Abadi. A Comparison of Approaches to Large Scale Data Analysis. In Proc. of SIGMOD, 2009.
- [2] U. Kang, C. Tsourakakis, A. Appel, C. Faloutsos, and J. Leskovec, Hadi: Fast diameter estimation and mining in massive graphs with hadoop, CMU-ML-08- 117, 2008.
- [3] U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos, Pegasus: A Peta-Scale Graph Mining System - Implementation and Observations. Proc. Intl. Conf. Data Mining, 2009, 229-238.
- [4] C. Olston, B. Reed, U. Srivastava, R. Kumar and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. ACM SIGMOD 2008, June 2008.
- [5] D. Koutra, T.Y. Ke, U. Kang, D. Chau, H.K. Pao, and C. Faloutsos. Unifying guilt-by-association approaches: Theorems and fast algorithms. Machine Learning and Knowledge Discovery in Databases, pages 245260, 2011.
- [6] U. Kang, Mary McGlohon, Leman Akoglu, and Christos Faloutsos. Patterns on the connected components of terabyte-scale graphs. In IEEE International Conference on Data Mining (ICDM), pages 875-880, 2010.
- [7] U Kang, Brendan Meeder, and Christos Faloutsos. Spectral Analysis for Billion-Scale Graphs: Discoveries and Implementation. In Proc. of the 14th Pacifc-Asia Conference on Knowledge Discovery and Data Mining, Hyderabad, India, June 2010.
- [8] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: a scalable and general graph management system. In KDD, pages 1091-1099, 2011.
- [9] A. Kyrola, G. Blelloch, C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. OSDI'12, Hollywood, CA.
- [10] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker Graphs: An Approach to Modeling Networks. In JMLR, pages 985-1042, 2010.

A Appendix

A.1 Labor Division

The team will perform the following tasks

Jin Kyu Kim:

1. By 10/23: Finish Diameter for Postgres. (In progress)
2. By 11/05: Finish Degree distribution for Postgres. (Done)
3. Between 10/23 and 11/12, set up experimentation. (In progress)
4. By 11/20: Finish performance evaluation of diameter and degree.

Alex Degtiar:

1. By 10/23: Finish connected components algorithm for Postgres. (Done)
2. By 11/05: Finish Eigenvalue algorithm for Postgres. (In Progress)
3. Between 10/23 and 11/12, set up experimentation. (Preparation)
4. By 11/20: Finish performance evaluations of connected components and eigenvalues.

Jay Yoon Lee:

1. By 10/23: Finish PageRank for Postgres. (Done)
2. By 10/30: Finish RWR for Postgres. (Done)
2. By 11/05: Finish Belief Propagation for Postgres. (In Progress)
3. Between 10/23 and 11/12, set up experimentation. (In Progress)
4. By 11/20: Finish performance evaluations of PageRank, RWR, and BP.

A.2 Full disclosure wrt dissertations/projects

Jin Kyu Kim: His research is at storage system including Operating System, and flash SSD. Now he is expanding research interest to large scale computing framework for machine learning. Although the scope of his current research is limited to machine learning, he is strongly interested in graph mining. He believe that graph mining algorithms could be the best applications of his new large scale computing framework.

Alex : He is not doing any project or dissertation related to this project.

Jay Yoon Lee: He is not doing any project or dissertation related to this project.

Contents