A SURVEY OF PERSISTENT GRAPH DATABASES

A thesis submitted
to Kent State University in
partial fulfillment of the requirements
for the degree of Master of Science

by
Yufan Liu

March 2014
Thesis written by

Yufan Liu

M.S., Kent State University, 2014

Approved by

Dr. Ruoming Jin, Advisor

Dr. Javed Khan, Chair, Department of Computer Science

Dr. Janis Crowther, Dean, College of Arts and Sciences
# TABLE OF CONTENTS

**LIST OF FIGURES** ................................................................. vi

**LIST OF TABLES** ............................................................... vii

Acknowledgements ............................................................... viii

1  Introduction ........................................................................... 1

2  Graph data and algorithms enumeration ............................... 4

   2.1  Graph data ................................................................. 4

      2.1.1 Social network ....................................................... 4

      2.1.2 Web graphs/RDF ..................................................... 5

      2.1.3 User/product networks ............................................. 6

      2.1.4 Road networks/Geo networks ................................... 7

      2.1.5 Chemical/Biological networks .................................. 8

   2.2  Graph algorithms ......................................................... 9

      2.2.1 Traversal .............................................................. 9

      2.2.2 Scoring .............................................................. 10

      2.2.3 Clustering .......................................................... 11

      2.2.4 Graph models/generators ....................................... 11
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>file size growth during single insertion of ca-AstroPh figure.</td>
<td>36</td>
</tr>
<tr>
<td>2</td>
<td>Single insertion performance of Amazon-0302.</td>
<td>37</td>
</tr>
<tr>
<td>3</td>
<td>Single insertion performance of ca-Astroph.</td>
<td>37</td>
</tr>
<tr>
<td>4</td>
<td>Average query time of ca-Astroph.</td>
<td>38</td>
</tr>
<tr>
<td>5</td>
<td>Run time of Dijkstra algorithm.</td>
<td>39</td>
</tr>
<tr>
<td>6</td>
<td>Run time of weak-component clustering algorithm.</td>
<td>39</td>
</tr>
<tr>
<td>7</td>
<td>Run time of pagerank algorithm.</td>
<td>40</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>traversal performance comparison between relational database and graph</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>dataset statistics.</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>Batch loading performance (seconds).</td>
<td>34</td>
</tr>
<tr>
<td>4</td>
<td>Comparison of supported features.</td>
<td>35</td>
</tr>
</tbody>
</table>
Acknowledgements

Firstly, I would like to express my deepest gratitude to my advisor, Dr. Ruoming Jin, who has excellent knowledge, patience, and enthusiasm for doing research. His guidance continuously helped me during my research and writing of this thesis. Besides my advisor, I would like to thank the rest professors of my thesis committee, who give me valuable suggestions and questions which help me to improve my work.

My thank also goes to my fellows in the data mining lab in Kent State University, they have helped me since the first day I joined the lab. I would not have this improvement without them.

Finally, I would like to thank my mother, who has supported me and give me encourage throughout my life.
CHAPTER 1

Introduction

A graph can be formally defined as a set of vertices and edges, e.g., $G=(V,E)$. The concept of the graph theory can be attributed to Leonhard Euler, who first solved the Seven Bridges of Konigsberg problem in 1736. The research of graph theory has brought a strong resurgence in both theoretical and applied areas in mathematics, physics, bioinformatics, and computer science ever since. At the same time, the type and quantity of graph data has been grown drastically in recent decade. For type perspective, when represent Wikipedia articles as vertices, there may be several properties belong to one vertex. For quantity, there are Facebook from 2007s 1million to 2012s 1billion users and Twitter from 100 million tweets per quarter of 2008 to 50 million tweets per day in 2010. To structurally store and manage the data, a database has to be used. The relational databases have dominated the industry of information storage since the late 1960s. But there are certain shortages when applying the relational databases to process the graph data. Fortunately, besides the relational databases, we can see many new attempts at providing a database model for large graph data. Especially in commercial areas, graph databases start to be used on dynamic schema and complex data.

Graph database is just a term of databases that specifically store and manage graph data in most of cases. So behind the name, there are many different techniques has been applied to each graph databases. For instance, the trend of NOSQL (not only SQL) databases has brought scalability for handling massive amount of data. The types of
the NOSQL usually include column-based, document-based, key-value-based and graph-based. Some of the graph databases are also implemented base on the technique like column-based or document-based. Besides those kinds of databases, there is another technique called distribute graph process which build a scalable platform on top of the column-based databases. One of the contributions of this thesis is categorized the existed graph databases and processing framework based on their storage behavior.

As the spreading usage of graph database, people from both of the academic and industrial area are paying more attention on research and development on it. From 2012, many researchers have contributed their work to improve the performance and to add features of graph databases. Meanwhile the graph databases have many contributions to open source community: not only some of the graph databases, but also the related tools and query languages are entirely open sourced. For example, Tinkerpop [1], which provides common tools and API for most of the graph databases. Having those graph databases and uniform tools to access them has motivated the comparison. In the experiment chapter of this thesis, we not only have the batch loading performance, but also contribute the comparison of stress test of single node insertion, query, and in-memory graph computation algorithms. Based on the experiment results, we will also have some recommendations on selecting the proper graph database according to applications and data. To cover as many use cases as possible, the datasets we have used include social network, product network, citation network and collaboration network.

The rest of this thesis will be composed as following structure: chapter 2 we will go through some of the typical real-world graph data and the some of the algorithms that
we will usually apply on graph data. In chapter 3, we will go through some state-of-the-art graph databases and cover some essential techniques that been used in those graph databases. In chapter 4, the uniform API and query languages would be introduced to help us access the graph data stored in the database. Chapter 5 will give the performance comparison between selected graph databases, the experiment result would imply us the how to choose the graph database based on the applications. In chapter 6, we will present some commercial use cases and discuss the future trend of graph databases.
CHAPTER 2

Graph data and algorithms enumeration

2.1 Graph data

From the time we step into the 21st century, the enriched logically structure of data and enhanced physical environment help us to collect massive data. Among those different format of data, graph data contains the most comprehensive information for us to analysis and mining. From social network work to protein network, the graph-structured data can be seen exponentially grown anywhere. Meanwhile, networks from different sources have different natures, which not only affect the applications that we want to use them, but also the performance we use them. For example, the scale-free networks, whose degree distribution follows the power-law. If we have special design on storing and processing them in the distribute system, the performance will have huge improvement. Topic about this will be discussed in the third section. In this section we will list some graphs which have the most impact in our real life, we can find that most of these graphs can be fit in power-law distribution.

2.1.1 Social network

In here the social network mainly means online social network service (SNS). Users tend to share their activities (Facebook), contents (Twitter), photos (Instagram), videos (Snapchat), or professional information (Linkedin) with friends on their websites. The
graph representation of the networks usually treats the user as the node and the relationship (friends, follow, etc) as the edge. Considering the business usage of SNS, companies can acquire marketing information from the content or activities from user or advertise to the users immediately on their personal page. To balance the user experience and the business purpose, precisely targeting the advertisement to each user is a primary task in the business model of SNS. Given this, previously people use machine learning techniques such as collaborative filtering [2] to look at the people (nodes) and the entities (node attributes) immediately to provide the recommendation result. Recently, people start to use the relationships (edges) between people together with the people entity information to get a more precise result on the recommendation [3]. This process will heavily rely on mining the graph data itself which requires high computation and storage abilities. The graph databases and graph computing framework to be presented below will provide the ability for academic and industry users.

2.1.2 Web graphs/RDF

Web 2.0 brings us a gigantic graph in which the nodes represent each web page and the edge usually means the hyperlink between each page. Efficiently indexing these web pages, and building search engine on top of the index will help people quickly find the resources they are looking for. Google, the largest search engine provider has made huge impact on processing the graph-like web data. Its techniques range from ranking algorithms PageRank to graph processing framework Pregel. They either provide different tools we can use to further understand graph data or improve the performance on running the mining algorithms. Those techniques will be introduced below.
Beyond web pages, W3C has used Resource Description Framework (RDF) to manage the web resources. RDF is a data model of graphs of subject, predicate, object triples. For example, the subject is the URL to describe a person, then a predicate can be name is, and the object can be John. Using this kind of structure, knowledge can be efficiently stored through the Internet. DBPedia is an RDF version of information from Wikipedia. This kind of way to format the web resources can potentially make querying the web immediately possible, which is the concept of semantic web. There are some websites already use RDF to storage their web resources: DBPedia is a project that Wikipedia created to store the knowledge from website into RDF. It contains data derived from Wikipedia’s infoboxes, category hierarchy, article abstracts, and various external links. By mapping the subject and the object to nodes and predicate to edges, the RDF can be expressed by graph structure, so the process of storing, loading and querying RDF data would have similar manner with graph data.

2.1.3 User/product networks

Online retail companies like Amazon, Ebay or online DVD rental company Netflix flies in these years and meanwhile gathered large amount data of their user and the products they bought. Those data not only include the attribute of each purchase transaction, but also the reviews form other customers. The relationship between customers and products can be mapped to edges and hence those data can be fit into co-purchase network or co-review network. Compare to social network, the advertising result is more obvious: social network has to tracked the number of ads viewing while the online retailers can count the product purchase immediately. This means the precision of the recommendation is
more critical for these companies. Netflix even has opened a competition for 1 million dollars prize on predicting the rating of its movies from 2007 to 2009. From the research from [4], the co-purchase network also has shown the property of power-law distribution. Considering the quantity of the data, there will be benefits from using specific techniques to store and analysis them.

2.1.4 Road networks/Geo networks

The graphs representation of geospatial or location information can be traced back to the beginning of graph theory: Eulers Seven Bridges of Konigsberg problem. In recent years, along with the advanced telecommunication technologies, the location-based service has been a hot spot in both research and business areas. In our daily life, the road networks data with GPS on mobile devices can guide us directly to the place we are going to, the common shortest path algorithms are being executed every time we are looking for a route. So faster response time and higher accuracy are required when running the algorithms on graph data. Because of the constraint of the mobile devices (memory, CPU, network, etc), the requirement of the algorithms is higher than the offline computing on cluster. Besides the traditional usage of the graph data on GPS that has relatively static addresses (nodes) and roads (edges), there are more requirements of the dynamic traffic due to the increasing traffic congestion in the cities all around the world. Waze is the mobile application that provides the service to solve the problem. It can track the traffic condition and provide the suggestions that can lead you to the destinations within the minimal time spent in real time manner. This application requires re-computing the shortest path more frequently following the changes of the traffic conditions. Although
some of the heuristic method can be applied when implementing the application, there still will be much more graph data to be stored and processed.

Besides the navigation usage, geospatial data can be seen more usually in other location-based services. Companies like Yelp and Foursquare provide the recommendations of places like restaurants or shopping malls by users review or check-in data. These data are all labeled by the geo location information. Similar to user-product network, the location-based service can easily embed the advertisement into their services. By combining the user information and geo graph data, applications can provide more reliable recommendation/advertisement to their users. Some researchers also find some interesting result by combining the geo location and social network to provide result that predict the human behaviors [5].

2.1.5 Chemical/Biological networks

The graph structure data mapping is also widely used in the research of fundamental science like chemical and biology. There is a branch in chemical called chemical graph theory. In this area, A chemical graph is a labeled graph whose vertices correspond to the atoms of the compound and edges correspond to chemical bonds [6]. In biology or bioinformatics area, graph theory also can be seen everywhere as a powerful tool to help people discover new features. For example, in a protein-protein network, proteins with similar roles tend to serve similar metabolic functions. Thus, if we know the function of one protein, we can predict that all other proteins having a similar role would also have similar function [7].
2.2 Graph algorithms

Here we will briefly discuss some of the most popular algorithms that have been used on graph data. To efficiently run those algorithms is an essential mission for graph databases. In the 6th section we will have some algorithms tested and performance analysis between selected graph databases.

2.2.1 Traversal

A traversal algorithm can be referred as the steps that visit all the nodes along the edges in the graph. Other algorithms on graphs rely on the basic traversal methods more or less: BFS (breadth first search) and DFS (depth first search). The reachability algorithms answer the questions that whether a node u can reach node v through a simple path of the edges in the graphs. The graph mentioned here commonly means directed graph. Because for undirected graphs, the problem can be referred to identify the connected components: if the nodes are connected, they are all reachable. According to the query manners, there are two classes of reachability algorithms: those require preprocessing and those not. For applications that require multiple queries on the graph repeatedly, using preprocessing can save query time to $O(1)$ by using extra space to save temporary result. While for the applications that require one-time query, using linear BFS or DFS will be sufficient. Recently the researches are mainly focusing on compressing the space using and query time for preprocessing [8].

Similar to the reachability problems, there are preprocessing and non-preprocessing methods. On the non-preprocessing side, there are famous algorithms like Dijkstra algorithms base on basic BFS. On the other hand, the preprocessing algorithms like in [9]
would use the highway heuristics to get an approximate result in very short time slot. Both of them have many applications: find the shortest way in road network, or find the shortest hops between two people in social network (LinkedIn).

2.2.2 Scoring

In many applications people want to give the nodes in the graph in order to index them in a certain order (web pages from search engines, importance of scientific journals by citations, etc).

PageRank/eigenvector centrality Famous for its usage in Google's search engine, PageRank has also been used in ranking journals [10] or even in ranking spaces or streets to predict how many people (pedestrians or vehicles) come to the individual spaces or streets [11]. The basic idea of PageRank is a set of recursive steps that compute the vertices score according to its neighbors scores: 

$$ R(\nu) = \sum_{\mu \in N(\nu)} \frac{R(\mu)}{D(\mu)} $$

where the $D(\mu)$ is the out-degree of vertex $\nu$ and $R(\nu)$ is its ranking score. This process is the same as simple random walk, but PageRank modifies it by adding to the model a probability alpha of jumping to any vertex. If alpha is 0, this is equivalent to the eigenvector centrality algorithm; if alpha is 1, all vertices will receive the same score ($\frac{1}{|\nu|}$). Thus, alpha acts as a sort of score smoothing parameter. Typical values for alpha (according to the original paper) are in the range [0.1, 0.2] but may be any value between 0 and 1 inclusive.

Betweenness centrality Another scoring method is betweenness centrality [12], it measures the number of shortest paths from all vertices to all others that pass through that vertex: 

$$ g(\nu) = \sum_{s \neq \nu \neq t} \frac{\sigma_{st}(\nu)}{\sigma_{st}} $$

where $\sigma_{st}$ is the total number of shortest paths from node $s$ to node $t$ and $\sigma_{st}(\nu)$ is the number of those paths that pass through $\nu$. Development
of betweenness centrality is generally attributed to sociologist Linton Freeman, who has also developed a number of other centrality measures.

### 2.2.3 Clustering

Another important kind of algorithm is clustering or cluster analysis. Its task is to find the groups or clusters by finding the similar data points on their attributes. It is an algorithm that generally used in machine learning, pattern recognition, information retrieval, etc. Some of the clustering algorithms has not considered the relations between each data node, such as the k-means clustering while some of those calculate the connectivities between the data nodes such as linkage clustering.

In this thesis, we will have the experiment of very basic clustering algorithm of computing the weakly connected component. A weak component is defined as the largest subgraph that each node belongs to it has at least one edge to another node belongs to it. The algorithm requires scan the nodes along their edges once, so the running time of it is \( O(|V| + |E|) \).

### 2.2.4 Graph models/generators

Sometimes people want to do the attribute analysis or predict the growth of the graph dataset, they will need to build a graph model that simulate the characteristic of the network. For instance, as mentioned above, the web page network has power-law distribution property, to simulate the process of growth, Barabási-Albert has invented BA-model in 1999 [13]. This model start growing (creating new nodes and edges) from a relatively small network, the new edges tend to link nodes that have high degrees. The probability of linking nodes is: \( P = \frac{d_i}{\sum_{j=1}^{d_i} d_j} \). On top of BA-model, there are LCD model,
Deterministic Scale-Free Networks, etc aim to complete the attributes that BA-model doesn’t have.
3.1 Graph data formats

In this chapter, we will first introduce some of the most used data structures for representing graph data. They would be used in the graph databases we described later in this chapter, and it will help to understand why the graph database has its own advantage to store and access graph data.

3.1.1 Edge list

In some materials edge list may be refer to the adjacency list, but here it will be defined separately, because many data especially the streaming data collected from the Internet are stored in this kind of format. An edge list here is a set of edges $M$, each edge is composed of two nodes $i$ and $j$ and usually be indexed as integer identifier, eg: 1 or 2. If the graph is undirected, the order between two nodes will be the smaller one followed by the larger one. In directed graph, the order can be reversed if the larger one point to smaller one.

This graph structure usually appears in relational database, because it can be easily expressed in table by two columns. There are certain shortages to use it, we will discuss that in later chapters when introducing relational based storage. As it usually to be raw data, people will always transform it to adjacency matrix or adjacency list in their application. The data we will be used to test in graph databases in the chapter 5 are all
formatted in edge list and in plain text.

3.1.2 Adjacency Matrix

We can represent edges with the aid of a two-dimensional array of $n^2$ boolean values, in which $n$ is the number of vertices. The indexes of the array correspond to the node identifiers of the graph, and the boolean value intersect by two node identifiers. If the graph is undirected, then the matrix should be symmetric, which shows that there is an edge between nodes $i$ and $j$. If the graph is directed, $E_{ij}$ indicates an edge that is going from the node $i$ to $j$. The value of the cells in matrix represents the value of edge for weighted graphs.

Although adjacency matrix is very efficient for adding, dropping edges and querying if two nodes are connected because the operation is immediate, it has some obvious drawbacks. First, it takes a fixed quadratic space with respect to the number of nodes, independently of the number of edges, and hence it is not efficient if the graph is sparse, most of the space will be used to store null data. Secondly, the insertion of nodes is expensive because it has to restructure the matrix to add a value to each dimension. In addition, the operation to retrieve all the neighboring nodes takes linear time with respect to the number of vertices in the graph. To get a 2-hop or 3-hop neighbors, it usually has to do matrix multiplications which is too resource consuming. Since traversing neighbors is one of the most common operations, adjacency matrix is not usually used for graph databases.
3.1.3 Adjacency List

Adjacency list uses a set of lists where each one stores all of the neighbors of one node. Each list in adjacency list represent one node, and the members of the list are neighbors to this node. If the graph is undirected and an edge connects nodes i and j, then the list of i contains the node j and vice versa. If the graph is directed, and I points to j, only j will be contained in Is list.

In contrast to adjacency matrix, adjacency list representations are efficient for obtaining the neighbors of a node, and nodes insertion is much cheaper: it just requires add a new list to the structure. More importantly, it is much more compact when storing sparse graph because the edges that do not existed between two nodes wont consume space anymore.

On the other hand, checking if there is an edge between two nodes, adjacency lists is slower than adjacency matrix. Because in adjacency matrix, this process can be done in constant time, while in adjacency lists, it requires O(m), where m is the amount of the nodes neighbors. This can be expensive for nodes with a large number of connections. Another method to enhance this is to keep the adjacency lists sorted by node identifier, which reduces the time to search to O(log(m)). However, if the lists are sorted the insertion time increase from constant to O(log(m)) at the same time.

3.1.4 Encoded adjacency lists

Although the adjacency list save a lot of space on storing the graph, there is still improvement can be done to get a more compact storage. There are many research [14] has shown that the most of the network form Internet still follows the power law
distributions. That means most of nodes do not have huge amount of neighbors, so encoding the nodes in each adjacency list will be a proper way to further compress the graph. There are many encoding technique been developed to compress the graph structure, here we will have two of them: gap encoding [15] and reference encoding [16].

Gap encoding: It stores the neighbors of the graph by differential between node identifiers one by one. This encoding will reduce the size of the overall lists. For example, we have the adjacency list for node x: \( S(x) = (s_1, s_2, s_3, \ldots, s_n) \), the gap encoded list will be \( (s_1 \times, s_2 - s_1, s_3 - s_2, \ldots, s_n - s_{n-1}) \). Since the adjacency list \( s(x) \) is sorted, all the elements except the first are positive numbers. If you want to retrieve the identifier of current node, you have to do a linear scan from the first node to current, then calculate by the stored value of those nodes. It will consume more resource, but due to the power law distribution, the cost will not be too big for most of the nodes.

Reference encoding: The idea of reference encoding is similar to gap encoding, but instead of using the differential between the identifier of each neighbor of one node, it uses the differential between the neighbor of two nodes. For example, there are two nodes A and B. As neighbors are 3, 6, 7, 12, 17; Bs neighbors are 3, 6, 8, 12, 13. Then B can be presented by a bit vector 11010 plus the node identifiers 8,13 corresponding to A. Because A has neighbor 3,6,12, so they are represented by 1, and vise versa. The nodes 8 and 13 do not As neighbor, so they have to be stored explicitly. This encoding is usually used in graph partitioning or clustering, because the nodes have similar neighbors tend to be represented together. But if used in common graph, then the cost will be too big if there are too many nodes.
3.2 Persistence graph databases

The persistence graph databases, or disk-based databases discussed here are not necessarily only persisting their data in-disk. For instance, Orientdb and Titan all support in-memory mode with specific configurations when initializing the database. The reason using term is mainly to differentiate it from the other big category of platform on graph: data distribute graph processing framework. Under the branch of persistence, the graph databases are further categorized by the structure or technique used for data storage. In total we will have graph database of relational storage, NoSQL storage, native graph storage and distribute storage. Notice not all of the graph database has been included in this section, most of selected databases here are already been put into commercial usages.

3.2.1 Graph Databases over Relational Storage

As mentioned in the edge list section, graph data can be stored in a relational table with two columns. Labels and attributes of nodes and edges can be managed separately in other tables and referred by foreign keys. As the dominated database management system form the 60s, RDBMS has its advantages to store graph data: well developed indexing system, sophisticated transaction support, the query language: SQL is a long-established standard and has fast learning cycle.

An example for a graph database implemented over relational storage is GRIPP [17]. It is an index structure for answering reachability and distance queries in graphs. It can be run top of common RDBMS, but requires extra space to store pre-defined indices. Another project called Periscope/GQ, by the University of Michigan implements a graph
database as an application over the PostgreSQL relational database engine. There are also approaches to do graph pattern matching on relational databases [18].

Those projects are mostly experimental oriented. But in general, relational databases are not suitable for storing and operating the graph data. Here are some summarizations about the reasons: Traversal algorithm is one of the most used operations on graph databases; most of other algorithms like page rank are based on it. On relational databases even simple traversal algorithms require costly self joins on the table. If running multiple times of self joins, it may exceed the memory limit quickly and as a result the traversal algorithms cannot be completed. The Index structures in relational databases are usually focused on answering queries over values of attributes. If we want to use index to answer queries corresponding to relationship structure, it still need recursive joins to do the joins. There is a small test been done in [19] by Partner and Vukotic. For a social network containing 1,000,000 people, each with approximately 50 friends.

<table>
<thead>
<tr>
<th>Depth</th>
<th>RDBMS Execution time (seconds)</th>
<th>Neo4j Execution time (seconds)</th>
<th>Records returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.016</td>
<td>0.01</td>
<td>2500</td>
</tr>
<tr>
<td>3</td>
<td>30.267</td>
<td>0.168</td>
<td>125,000</td>
</tr>
<tr>
<td>4</td>
<td>1543.505</td>
<td>1.359</td>
<td>600,000</td>
</tr>
<tr>
<td>5</td>
<td>Not finished</td>
<td>2.132</td>
<td>800,000</td>
</tr>
</tbody>
</table>

Table 1: traversal performance comparison between relational database and graph database.

Another big issue is more and more graph databases now start supporting distribute storage as the increasing size of graph. Instead of partitioning the graph like most of
distribute graph databases do, RDBMS can only use Sharding to split the table horizontally. That will make traversing the graph much more difficult. Considering above issues, for general usage, conventional relational databases are impractical for managing large graph datasets.

3.2.2 Graph Databases over NOSQL Storage

Besides relational databases, new storage technologies have been created to solve the problems relational databases are not good at. There is a term to name those databases: NoSQL. A NoSQL (Not Only SQL) database provides a mechanism for storage and retrieval of data that employs less constrained consistency models than traditional relational databases. Motivations for this approach include simplicity of design, horizontal scaling and finer control over availability (http://en.wikipedia.org/wiki/Nosql). The term NoSQL is usually used to describe all the databases that not in the RDBMS family, so graph databases should also be included, but in this section, we will separate the native graph databases to other NoSQL storage techniques such as column-based and key-value-based storage.

key-value databases allow the application to store schema-less data. (Key-Value stores: a practical overview). The term key is like it in relational database, the value is the data store in the database corresponding to the key. The data is usually primitive type (string, integer, etc) or an object defined by certain programming languages. This replaces the need for fixed data model and makes the requirement for properly formatted data less strict. As a consequence, it provides scalability a better way of graph partitioning. HypergraphDB (www.hypergraphdb.org), using the key-value model in BerkeleyDb.
It stores all the graph information in the form of key-value pairs. Each object of the graph, either nodes or edges, is identified with a single key that is called atom. Each atom is related to a set of atoms, which can contain zero or any number of elements. These relationships create the topological structure of the graph. HypergraphDB also provide an optional hypergraph structure. The edge in hypergraph called hyperedge which connects an arbitrary set of nodes. This would help abstract the graph structure to provide fast way to do query. The key-value structures are stored on an external library, BerkeleyDB, which is supported by Oracle Corporation.

Similar to key-value-based storage, document-based storage also allows schema-less storage, plus it provides higher data complexity. The term document here usually means encoded data in some certain formats, eg: XML, JSON, or even binary forms like PDF, Microsoft Office documents, etc. OrientDB(www.orientdb.org) is a open source Graph Database with a hybrid model taken from both the Document Database and Object Orientation worlds. It can work in schema-less mode, schema-full or a mix of both. Supports features such as ACID Transactions, Fast Indexes. It also highlights advanced features like SQL queries, and easy access to local/remote/in-memory databases. It is a document-based database, but the relationships are managed as in graph databases with direct connections between records. OrientDB uses a new indexing algorithm called MVRB-Tree, derived from the Red-Black Tree and from the B+Tree. According to the report in (XGDBench: A benchmarking platform for graph stores in exascale clouds), the OrientDB provides higher average throughput than Neo4j and AllegroGraph.

Titan (http://thinkaurelius.github.io/titan/) is a scalable graph database optimized for storing and querying graphs containing hundreds of billions of vertices and edges
distributed across a multi-machine cluster. Different from other graph databases, Titan has been implemented on different backend storage systems, eg: Apache Hbase, Apache Cassandra, Oracle BerkeleyDB, etc. This pluggable nature brings Titan: If data already stored in one of the databases above, import graph will not have extra IO cost on data transporting form other databases. Also different backend provide the databases different properties which follow the CAP (availability, consistency and partitionability) theorem.

In addition, Titans developer made another open sourced tool, which called Faunus. It is a Hadoop-based graph analytics engine for analyzing graphs represented across a multi-machine compute cluster. Titan has been tested supporting thousands of concurrent users executing complex graph traversals at the same time.

3.2.3 Native Graph Databases

The term of native graph databases refers to the databases use object model of nodes and relationships to store all of its data. These graph databases will have their own file systems to store the data instead of relying on other storage techniques. As the name implies, these databases are always optimized to store and manage graph-like relationship data. At the same time, they may share some features to fulfill the requirements as high performance database services. Neo4j( http://www.neo4j.org/) is a disk-based transactional graph database that implemented in Java. It is one the most popular graph databases. Besides the advantages in traversing the graph data like other graph databases, it highlights its full ACID transaction support and convenient REST server interfaces. Those features ensure it to be suitable for its enterprise solutions. It also provides its own query language called Cypher, which can handle different kinds of queries
with syntax. At the same time, their developers make APIs for almost all of programming languages to access their graph database. (http://www.neo4j.org/develop/drivers). AllegroGraph (www.franz.com/agraph/allegrograph/) is a high-performance, persistent disk-based graph database. It is initially implemented as an RDF databases, and serves for the SPARQL query language. Similar to Neo4j, it also fully supports ACID and provides REST service. By providing Amazon EC2 AMI (Amazon Machine Image), it can be easily hosted on the cloud. DEX(www.sparsity-technologies.com/dex) is an efficient, bitmaps-based and disk-based graph database model written in C++. The focus of DEX is managing and querying very large graphs with high performance. By encoding adjacency matrices into bitmap, it uses more compact disk space and requires less memory space. In addition to the large data capacity, DEX has a good integrity model for management of persistent and temporary graphs. In contrast to other graph databases like Neo4j, it guarantees consistency, durability and partial isolation and atomicity for its ACID. InfiniteGraph(www.objectivity.com/infinitegraph) is a distributed-oriented system that supports large-scale graphs and efficient graph analysis. It supports a variety of system implementations on server-based systems, cloud platforms, and embedded appliances. When in distribute storage more, InfiniteGraph uses its Advanced Multithreaded Server (AMS) to access the data. AMS is an alternative to native file servers such as NFS or HDFS, and it will be installed as a service to each machine in the cluster.

3.3 Distribute graph processing frameworks

For web applications, scalability is always an initial concern when people are making choices on the database management system. Distributed graph databases focus on
distributing large graphs across clusters and providing high performance access. Although graph databases like Neo4j and OrientDB now start to support running in distributed mode on clusters, they are not designed to be stored in distribute environment like the graph databases introduce in this section.

One main task for distributed graph databases is partitioning graph data. It is a non-trivial problem, optimal division of graphs requires finding perfect subgraphs of the whole graph. For most of the data, the number of edges is too large to efficiently compute an optimal partition. There is always trade-off to partition the graph. Therefore most databases now use random partitioning.

The distribute graph databases provides data persistence by storing graph into distributed file systems. But there are also another set of frameworks just process the graph data with persist them, they are commonly called distribute graph processing. For example, Faunus, a batch graph processing engine, also developed by the company which invented Titan. This engine is built on Hadoop platform and use MapReduce job to batch process the job. It has similar API with Titan, and by leveraging the benefit of MapReduce, It is started from Pregel [20]. Pregel is Google’s internal graph processing platform. It is inspired by Bulk Synchronous Parallel. It makes some graph queries scalable on hundreds of billions nodes and they can be built on distribute systems like Hadoop and HDFS. Other systems such as Apache Giraph and Apache Hama are similar to Pregel, a user cases of Giraph is Facebook is using it to do clustering and ranking on its social graph.

Similar to Pregel/Giraph, GraphLab(http://graphlab.com/) also provides a distribute graph processing framework which is written in C++. It defines its own method of
vertex-cut compared to edge-cut when distributing the graph. In its latest work, they present PowerGraph, which is optimized for the graphs that have power-law distribution property [21]. In their experiment report, they have outperformed Pregel-like systems.

Besides Pregel and PowerGraph, UC Berkeley has developed another system called GraphX [22]. It is built on top of Spark, a MapReduce-like data-parallel computation engine open-sourced also by UC Berkeley. The Spark system differentiates itself from traditional MapReduce on processing the data in-memory. For the applications that can fit their data into distributed memory, Spark can achieve a huge performance improvement than MapReduce by saving IO cost on the file system. Also because the data is kept in memory, it will be easier to query data during the whole computing process, this benefit can help building real-time streaming applications on large data. Meanwhile, for the applications that cannot fit into memory, there won't be any performance benefit on using it.

3.4 Internal Techniques

Each graph database has its own way for implementation. But they do share some ideas on designing the techniques sometimes. Here we will discuss some of the internal techniques been used in the graph databases to acquire its high performance requirements on querying the connected data.

3.4.1 Index-free adjacency

Index-free adjacency (Neo4j) makes the nodes contain direct reference/index to their adjacent neighboring nodes. This means it is unnecessary to maintain a global index for relationships between nodes/edges. This term has similar concept as adjacency list,
except the local indices for each node may also contain the property of nodes and edges. By using Index-free adjacency, time of queries based on traversal the graph will be independent of the total size of the graph.

3.4.2 Vertex Centric Indices

Vertex Centric Indices (Titan) is used to find if a certain node is connected to another node by their unique identifier. As the description in the second chapter, most networks in real world can be fit with power-law distribution. It means most of the nodes do not have large amount of neighbors. But there do exist some supernodes which connect large number of nodes, these supernodes result in the small world phenomena. For example, US president Barack Obama currently has 38,844,920 followers on Twitter. Applying graph term to it, the vertex Obama is considered a supernode in Twitter’s network. Obama send a tweet, it will be sent to more than 38 million accounts on Twitter. To store these supernodes, graph database like Titan will first build an index to classify the edges according to edges label or property, because there may not be just one type of edge for the supernode. Then it sorts the incident nodes of the supernode. By doing these, when querying a vertex’s neighbor, linear scans of incident edge (O(n)) can be reduced to at least O(log(n)).

3.4.3 Bitmaps representation of graphs

Graph database like DEX uses adjacency matrix to multiple small indices to improve the management performance. The indices will be encoded into bitmaps. The first bit represent the first node insert to the graph, the last bit represent the latest coming node. The relationship between each bit and node will be kept in an index, when there is a
new node inserted, the index will be updated. To add an edge between two nodes, it just
needs to update the related bit to 1. DEX also uses key-value maps to provide the index
of full data access to complement the bitmaps. Because the adjacency is usually sparse,
the compressed bitmaps will use the space more efficiently. Another advantage is that
the insert, delete operations are all bit operations, which are much faster than common
operation for objects.

3.4.4 Write Ahead Log

Different from other techniques discussed in this section, Write Ahead Log (WAL)
is a technique that widely used in transactional database. In transactional systems,
atomicity and durability (from the ACID guarantees) are provided by using a WAL. All
modifications during a transactional are persisted on disk as they are requested but are
not actually stored. After the transactional is committed, only then the modifications
of transactions will be really stored on the database and the transactional is removed
from is temporary place. In Neo4j, WAL stores 3 files, a marker file, keeping track of
which of the other two is the current active log file [23]. When the transaction finished
normally, marker file will be marked as healthy and two other log files are removed. But
if the process crashes, the marker file will be indicated as not finished, it will go look at
the last active log. If the log has all of information of the transaction, it will do recovery
by creating a new transaction. Otherwise it will do rollback, which means remove the
transaction files in temporary space. This technique has ensured the atomicity of the
transaction process. In most of graph databases API, we can call commit () to finish the
transaction.
3.4.5 Bulk synchronous parallel

MapReduce computing framework is a widely used in distribute systems, but it does not perform well running graph algorithms: for example, to run the PageRank algorithm on massive graph data, it is difficult to avoid the IO cost on storing the graph structure during every iteration of the computing process. To solve those kind of problem, Google invented Pregel, which we introduced above. Pregel is inspired by Bulk Synchronous Parallel [24], it is a bridging model for designing parallel algorithms. It is based on the concept of superstep. Each superstep consists of three components: concurrent computation. It means the processors use the value store on the local memory and they are independent to each other. Another one is communication, the sending and receiving of each node has be decoupled. The last component is Barrier synchronization: when a process reaches the barrier, it waits until all other processes finished their communication actions. The computation and communication actions do not have to be ordered in time. The barrier synchronization concludes the superstep. Lets use the graph algorithm pagerank as an example. If it is implemented in general MapReduce, one iteration needs one job to complete, after the job, the temporary data would be saved into disk which is IO consuming. While BSP will do one iteration in one superstep, and processes can store the temporary results for each node in-memory, and pass them to other nodes by themselves. By leveraging the features from BSP, the systems like Pregel and Giraph improves the algorithms running on graph data.
4.1 Query languages

As a complete databases management system, query language is a required feature to provide. Sometimes it may even decide if this kind of DBMS is suitable for certain applications. There are general query languages for different graph databases, while there are also some graph database-specified query languages such as Neo4js Cypher. Here we will introduce two widely used query languages.

4.1.1 SPARQL

SPARQL [25] is a query language for RDF data and is recognized as one of the key technologies of the semantic web. It can extract values from structured and semi-structured data and supports unambiguous queries. A SPARQL endpoint accepts queries and returns results via HTTP, the results of SPARQL queries can be returned and/or rendered in a variety of formats such as XML, HTML or JSON. So it can be adapt by web service like REST server easily.

To give an example of how SPARQL works, we will do a query on DBPedia, which we mention in RDF section. This query will return the result of people born before 1900 in China form DBPedia, and sorted by names:

\[\text{SELECT \ ?name \ ?birth \ ?death \ ?person WHERE} \{\]
The first line defines the resource of RDF dataset, then it provides the request result columns and the query condition. SPARQL variables start with a question mark and can match any node (resource or literal) in the dataset. This query can be tested on a SPARQL query explorer on http://dbpedia.org/snorql/ provided by OpenLink Virtuoso. SPARQL is supported by some databases such as AllegroGraph natively. Other graph databases like Neo4j also support SPARQL by applying plugin Linked Data.

4.1.2 Gremlin

Gremlin is a graph query language based on the open source software Blueprints. It can be used on all of the graph databases that support Blueprints. It can be used in various JVM languages such as Java, and Groovy. By working with Blueprint API, Gremlin can be used to manipulate the graph database. It provides a command line prompt to insert/update/delete the nodes and edges. As a query language, it can perform complex graph traversals compactly. To get the same query result, it uses much less code than using Java API. For instance, it just needs 3 lines to get eigenvector centrality of a
Another important feature of Gremlin is that there are many add-ons and connectors on it. For example, the JUNG algorithms library gives Gremlin the ability to compute centrality score, or do a clustering on the graph. The connector LinkedData Sail makes Gremlin easily access RDF just like SPARQL. Besides Gremlin, some graph database vendor also provide their own query language, such as Cypher from Neo4j. There is a performance comparison in [26] that concludes Cypher is more syntax friendly to the SQL users but the Gremlin has a little performance advantage when running the query. Considering its compatibility and pluggable nature, Gremlin should be the initial choice as the query language for graph database.

4.2 Blueprint API

Blueprint is an open source library written in Java, which has a collection of interfaces on graph database. It can be analogue as the JDBC for graph database. Beside accessing it programmatically, Blueprint API can also serve as the interfaces for other services in the Tinkerpop family: Pipes (A lazy, data flow framework), Gremlin (A graph traversal language), Frames (An object-to-graph mapper), Furnace (A graph algorithms package), Rexster (A graph server). It embraces the compatibility to most of the popular graph databases and libraries. Another important feature that the Blueprint API provides is
the wrappers to generate special graphs for certain use cases. For instance, there is *Batch-Graph* which provides the function to batch load the data into database; You can also use *IdGraph* to manage the id of nodes and edges by yourself, the *PartitionGraph* can help you to define your own partitions, and etc. By providing the uniform syntax to access different graph databases, it can shorten the learning curve for programmers from one graph database to another. Here we have some example to use the Blueprint API: to create an database, we can simply use the `Graph graph = new Neo4jGraph(directory);` or `Graph graph = new Neo4jGraph(directory);` To create a node: `Vertex a = graph.addVertex(null);` or an edge: `Edge e = graph.addEdge(null, a, b, "knows");` . When applying the pagerank algorithm from JUNG library, the syntax is also simple: initialize: `PageRank<Vertex, Edge> pageRank = new PageRank<Vertex, Edge>(new GraphJung(graph), 0.15d);` Then doing the computation: `pageRank.evaluate();` . The result score of each node would be stored in the `pageRank` object. The experiments In chapter 5 will leverage the Blueprint API to compose the test.
CHAPTER 5

Performance test and experiment

Comparison between data persistence graph databases In this section, we will have the experiments of comparison between the relatively sophisticated and popular graph databases [27]: Titan (BerkeleyDB, Cassandra and HBase implementation), Neo4j and OrientDB. All of them have been put into commercial practices. The user cases will be presented in the 6th section. Another main factor of the choices is the features they have. In Table 4, we can find all the graph databases we choose support costumed index and transaction which are required in the comparison. Because we are using the Blueprint API 2.4.0, the versions of each graph database will be consistence with the version of Blueprint: Neo4j's version is 1.9.2, OrientDB is 1.5.0 and Titans 0.4.0/0.4.1 version.

5.1 Experiment environment

As mentioned above, most of the experiments use the Blueprints API which is supported of the graph databases as either 1st or 3rd party API. The performance may not be optimal compares to their native APIs, eg: OrientDB has claimed that the performance on Blueprint may be 80% of its optimal performance [28]. But in order to not favor any graph databases by using their tuned API and considering Blueprints widely usage in practical, it is sufficient to test the performance between each database. For the algorithms test on each database, we use the JUNG (Java Universal Network/Graph Framework) [29]. It is naturally supported in Blueprint API.
The machine details: Macbook Pro, Processor: 2.7 GHz Intel Core i7, Memory: 16 GB 1600 MHz DDR3, Hard drive: Solid State Drive 1TB. System environment: Mac OSX 10.8.3, Eclipse kepler, java 1.7. This single machine environment may not leverage the advantage of distribute graph database like Titan. But it is sufficient enough to test the IO performance of basic operation of different graph databases.

The datasets we are using for the experiment include social network (facebook, flickr), product co-purchasing network (amazon), citation network (U.S. patent from National Bureau of Economic Research) and collaboration network (authors papers submitted to Astro Physics category). In Table 2 you can see the basic statistics of the datasets. There are two kinds of formats of raw data source as the data input. One is edge list in plain text, each line has two numbers separate by tab refers to the edge. The other one is the plain text of adjacency list, in which each line has the node and its neighbours. The number stands for id of each vertex.

<table>
<thead>
<tr>
<th>dataset</th>
<th>nodes</th>
<th>edges</th>
<th>average degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>ca-AstroPh</td>
<td>18,772</td>
<td>396,160</td>
<td>21.1</td>
</tr>
<tr>
<td>amazon0302</td>
<td>262,111</td>
<td>1,234,877</td>
<td>4.7</td>
</tr>
<tr>
<td>facebook</td>
<td>63,731</td>
<td>1,545,686</td>
<td>24.25</td>
</tr>
<tr>
<td>cit-Patents</td>
<td>3,774,768</td>
<td>16,518,948</td>
<td>4.3</td>
</tr>
<tr>
<td>flickr</td>
<td>2,302,925</td>
<td>33,140,018</td>
<td>14.4</td>
</tr>
</tbody>
</table>

Table 2: dataset statistics.

5.2 Batch loading operations:

In most of the applications, we would initially load the data into the database. The performance of massive insertion or batch loading process would reflect the basic I/O
throughput to the local disk.

<table>
<thead>
<tr>
<th>dataset</th>
<th>Neo4j</th>
<th>Titan</th>
<th>OrientDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ca-AstroPh</td>
<td>7</td>
<td>19</td>
<td>98</td>
</tr>
<tr>
<td>amazon0302</td>
<td>27</td>
<td>38</td>
<td>171</td>
</tr>
<tr>
<td>facebook</td>
<td>25</td>
<td>35</td>
<td>230</td>
</tr>
<tr>
<td>cit-Patents</td>
<td>587</td>
<td>901</td>
<td>2567</td>
</tr>
<tr>
<td>flickr</td>
<td>1037</td>
<td>1605</td>
<td>6846</td>
</tr>
</tbody>
</table>

Table 3: Batch loading performance (seconds).

By looking at the result in Table 3, we can easily find Neo4j has the best performance while OrientDB falls behind by 5-10 times than Neo4j. The speed for batch loading makes OrientDB not practical to do the one-time job with massive data. At the same time, we can notice that the time increasing is not linear with the edges, the reason is like we introduced in the chapter 3, the graph databases tend to use an index to store each vertex’s neighbour, when the index grows, it takes time to read and write the index to files. For the applications that need to be load graph data frequently to the database, and the graph size is not super large (more than 1 billion edges) Neo4j could be a good fit.

5.3 Single insertion and query operations:

Besides the batch loading as the initial step. We probably need to insert single node to the datasets. As we discussed above, as a commercial database, the capability of transaction data support is very important to save the data if some error happens. Especially when it being used for handling some critical data (financial, health, etc). Most commercial relational databases fully support transaction. But for graph databases implementation, someone doesn’t, and someone partially support. Another constraint
when choosing the comparison target is the support for indices, in most applications we need to customize index for each node/edge in order to retrieve them for querying and updating purpose. There is an comparison of the transaction and index related feature below in Table 4. Our experiments will focus on the single insertion operations with the customized index to the graph databases.

<table>
<thead>
<tr>
<th>graph database</th>
<th>Costumed Index</th>
<th>Key index</th>
<th>transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neo4j</td>
<td>true</td>
<td>true</td>
<td>fully</td>
</tr>
<tr>
<td>OrientDB</td>
<td>true</td>
<td>true</td>
<td>fully</td>
</tr>
<tr>
<td>DEX</td>
<td>false</td>
<td>true</td>
<td>partial</td>
</tr>
<tr>
<td>Titan</td>
<td>true</td>
<td>true</td>
<td>fully</td>
</tr>
<tr>
<td>InfiniteGraph</td>
<td>true</td>
<td>true</td>
<td>fully</td>
</tr>
<tr>
<td>AllegroGraph</td>
<td>false</td>
<td>false</td>
<td>fully</td>
</tr>
</tbody>
</table>

Table 4: Comparison of supported features.

For the insertion operation, in order to test the I/O performance under the extreme circumstance. When finish the transaction for each individual insertion, we use `graph.commit` to flush the data to the disk. At the same time, we give each vertex an id which will be indexed before each transaction. For edge insertion, first we retrieve the vertex according to the ids of both sides, then connect them by using `addEdge` function. There are two aspects we are watching for insertion: the insert speed and the file size that each database use to store the graph on disk. Because the file systems of HBase and Canssandra (HDFS for HBase CFS for Cassandra), the file size for them are not included in the comparison.

For the file size growth, we can find neo4j uses more spaces than the others in Figure 1. By looking at the files in the directory on the disk, they are mostly taken by the file
Figure 1: file size growth during single insertion of ca-AstroPh figure.

"nioneo_logical.log". It is the transactional log that keeps the history transaction event. This log can help the database recover from the abnormal condition (server crush, network down, etc). Though Titan and OrientDB also have similar logs, they are not kept on the disk for long and their size is smaller. Generally the growth of file size should be linear as the inserted data. But OrientDB has some bump during the insertion process. That means some of the data are still be temporarily put in the cache and flush into disk with other data at the same time even we use `graph.commit` after each insertion.

By observing the result of single insertion performance in Figure 2 and Figure 3, we can find that Titans implementation on HBase is not optimized, that makes its performance far behind others. For others performance, the Neo4j actually consumes more time on it, that should be caused by creating the transactional log. Also in any database system especially NoSQL, there is write read tradeoffs. For example, we can optimize
Figure 2: Single insertion performance of Amazon-0302.

Figure 3: Single insertion performance of ca-Astroph.
read performance by using column-oriented data representation and light-weight compression [30]. By investigating the performance of running algorithms listed below, we will find the insertion performance should be engineering tradeoffs in Neo4j to acquire better performance in traversal.

![Query time ca-Astroph](image)

**Figure 4:** Average query time of ca-Astroph.

Without building an index, to lookup a certain vertex in the graph we will have to scan over all vertices (O(n)), with an index it can be shorten to O(logn) or even O(1). We can find from the test in Figure 4, the Titan has the most effective index that helps the lookup operation outperform other graph databases.

5.4 Clustering and scoring computations

In the graph specific algorithms comparison, we have selected PageRank for scoring, Weak-Component for clustering, and Dijkstra for traversal. Other algorithms in each area
Figure 5: run time of Dijkstra algorithm.

Figure 6: run time of weak-component clustering algorithm.
Figure 7: run time of pagerank algorithm.

would have similar performance. The result in Figure 5, Figure 6 and Figure 7 shows that Neo4j has significant advantage on performance in comparing with other graph databases. Considering the insertion performance observed above, we could conclude that Neo4j has been traversal optimized for running the certain algorithms. As a production-ready database, Neo4j has shown its advantage in running algorithms in-memory, also with its well-composed documentation and abundant tools, it has been an initial choose for managing and process small to medium graph-like datasets. Meanwhile although Titan has less performance on the single machine in-memory algorithms, it still has advantages on storing large amount of data by leveraging the benefits from distribute storage platform like HBase and Cassandra. Also its abstraction allows it to be easily adapted to different data types. For the applications that require processing on large datasets, the distribute graph processing frameworks are still needed.
CHAPTER 6

Real world use-cases

There are many companies or organizations have integrated the graph databases into their services or products. By referring those use cases we can realize the graph database which kind of applications. The selected use cases are presented in the graph database conference GraphConnect, which is held every year by Neo Technology.

The financial and tax preparation software company Intuit has used Neo4j in their systems to store the company and product relationship data [31]. The scale of its graph is 29 million nodes, 315 million properties and 48 millions edges. The graph database is served as the real time query engine for applications like recommendation, referrals, and communities detection. Besides the main function, it claim to be benefit from its SQL-like query language Cypher and the REST service.

Different from Intuit, Leap Motion is a small hand-position measurement device which analogous to a mouse, but requiring no hand contact or touching. In [32], they introduced how to use Neo4j to model a novel approach to gesture detection. This allows people to consider each individual gesture as a path on this graph, eliminating the need for constant finger-tip tracking. They have integrated Neo4j’s RESTful API to their client side product. By collecting the data from its device, analyzing them by machine learning algorithms, it can provide more accurate tracking service to its users.
CHAPTER 7

Conclusion and future work

The applications that rely on managing and analyzing graph data has motivated the requirements of graph database. In this thesis we first summarized the different types of graph data, the most commonly used graph algorithms and the structure we use to store the graph data. Then we introduced and categorized different graph databases/processing framework based on their storage behaviors. To better understand the features of graph database, we also introduced some of the techniques that been used under the hood and the query languages and API to access the data. At last we use the blueprint API to compare the commonly used persistent graph database Neo4j, Titan and OrientDB on the performance of batch loading, single insertion with index lookup and some of the in-memory graph algorithms. The comparison results shows Neo4j and Titan both can be used on frequently loading the small to medium size graph. All of them can be used on store and query graph data, while Titan’s cassandra implementation has better performance on querying than its HBase implementation. At last, on the small-scale graph algorithm computation, Neo4j has significant advantage.

Although some graph databases like Neo4j have been already used in industry for certain markets, there are still many space to develop. The common data mining and machine learning tasks on large datasets can not be scaled on them. The distribute processing systems are required to handle the tasks. At this time, the implementations of these systems are in early stage (eg, GraphX and Giraph are still in incubator) and
language to access them varies between each other (scala, c++ and java). It is difficult to put those systems into practical by common users or programmers. But the incoming Tinkerpop3 would solve this problem. Its API and query language would support distributed graph processing like Giraph, HAMA, GraphLab, etc. The future work would then be focusing on comparing the performance of graph algorithms between distribute graph processing platforms on large-scale datasets.
BIBLIOGRAPHY


