RESEARCH PROJECT

William Ehrich: wde220

June 3rd, 2017

EECS 495: Readings in Database Systems, Spring 2017

Prof. Jennie Rogers

Graph Databases

Graph Data Models

Theory

Databases are classified as graph databases if their principle architectural design is based on the graph model, in which entities are represented by nodes and the relationships between entities are represented by edges between the corresponding nodes [2]. In any data model, real world objects and their interconnections are encoded using a fixed set of logic. In graph data models, data and/or schemas are expressed as graphs, or as generalizations of graphs, such as hypergraphs or nested graphs. In one graph model, the whole database is encoded as a single directed graph. In this graph model, external nodes correspond to data and internal nodes correspond to relationships. An alternate approach is to specify relationships and rules using labeled directed graphs. When graph models are used, the underlying data must be manipulated using graph transformations or operations on graph features. Important features of graphs include connectivity, paths, neighborhoods, subgraphs, as well as statistics, including centrality and diameter. As with other data models, graph data models can be subjected to integrity constraints. For example, nodes can require labels with unique names and a specific data types. Moreover, attributes can be subject to domains or ranges of possible values.

With respect to their benefits, graph data models offer a number of advantages. First, a graph data model can be a more instinctive representation of data when the connections between data are highly important. In particular, graph models offer an ease of visibility to users, in which connections are readily apparent. Second, the use of a graph model enables the use of graph operations, such as finding the shortest path, in queries. Finally, in some cases, the graph data model is more efficient from a storage perspective.

Because of these advantages, graph data models have been used in many applications. Probably the most famous application is that of social networks. In these types of applications, nodes correspond to people or groups and the edges correspond to friendships or other connections. Graph data models can also be used to model information networks, such as collaboration and co-authorship in research networks. Additionally, graph data models can be used to model technological networks such as telephone networks, airline routs, electric power grids, and even the internet. Finally, graph data models are well suited to biological networks, including those relating to metabolic pathways, homology relationships and gene regulation.

Implementation

To handle these applications, the graph data model has been implemented in a number of graph database models, which differ in their use of data structures, integrity constraints, as well as query and manipulation languages.

Although all graph database models use graph data structures, some rely on directed graphs and

others undirected graphs. Additionally, others use labeled edges, and others use unlabeled edges. Some graph database models use nested graphs models. In nested graph models, some nodes are hypdernodes, which contain nested graphs or other complex objects. Moreover, some models use hypergraphs, which feature hyperedges (edges that can connect more than two vertices). The advantage to using nested graphs and hypergraphs is an increased expressivity and tolerance for complexity compared to "flat" graph database models.

With respect to integrity constraints, some databases use defined schemas to enforce type checking and rules used to define a consistent state. In this case, all nodes and edges must also occur in the scheme. Other database models, however, do not have such a requirement, which allows for missing data or incomplete information.

Multiple query languages have created for graph databases. Of the many languages, one is the query language G. This language is designed to facilitate the specification of recursive queries. A user inputs a labeled directed graph (query graph) and the system returns all the graphs that contain the subgraphs of the input. The language was further developed into G+. In this updated language, the user inputs a query graph and, additionally, a summary graph, used to reform the query result. Finally, GraphLog is a language based on G+ that is extended to include additional functionality. In particular, GraphLog supports negation, among other features.

Among today's most popular graph database implementations are the following: AllegroGraph, DEX, HypergraphDB, InfininteGraph, Neo4j, and Sones [1]. AllegroGraph is designed for semantic networks and features additional functionality pertaining to social network investigation and GeoTemporal reasoning. DEX is engineered for performance, supports large graphs, and is exposed as Java library. HyperGraphDB, as its name implies, supports hypergraphs, in which an edge can have any number of nodes, a feature that is especially useful when using graphs to model bio-informatics, artificial intelligence, and knowledge representation. InfiniteGraph, as its name suggests, is optimized for large, distributed graphs, often used by businesses, government bodies, and social networks. Neo4j embodies a network model, where relations are objects and data is accessed using an object-oriented API. Finally, Sones is built for data abstraction at a high level and supports its own distributed file system and query language.

Comparison to Relational Model

Since their inception in the 1970s, relational databases have become the most widely deployed type of database [7]. Among the most popular relational databases are Oracle, Microsoft SQL Server, and MySQL. From a theoretical standpoint, relational databases adhere to the principles of atomicity, consistency, isolation, and durability (ACID). The principles of ACID, however, were formulated at a time of relatively simple database applications and small amounts of data. The increasing volume of data and computational workloads in recent years has led to the NoSQL movement that does away with the rigid concept of ACID. Among the different types of NoSQL databases is that of graph databases. Of the different graph databases, the most popular is Neo4j. Neo4j can, therefore, be juxtaposed to MySQL to observe the differences between relational and graph databases. Such an investigation can reveal that relational and graph databases differ greatly in their maturity, ease of programming, flexibility, and security.

Relational databases maintain a commanding dominance of the overall database market in large part because they are considered to be at a more mature state of development than are graph databases. Because of their long history, relational databases have undergone significant amounts of

testing, and are considered safer choices for production environments. The financial backing of large companies such as Microsoft and Oracle reassure database customer that technical support will be available for the foreseeable future. In contrast, much of the technical support for Neo4j revolves around a wiki page on the Neo4j website, and it is not clear if Neo4j, an open source project, would still be supported or patched if the startup company developing the project, Neo Technology, went out of business.

Another advantage of relational databases over graph databases is in their relative ease of programming. Relational databases support a common structured querying language (SQL). On the other hand, different graph databases feature different APIs. With that said, certain operations such as graph traversal are encoded much more easily with the Neo4j API than with SQL, as the required SQL statements often contain complex recursion and looping.

Concerning flexibility, graph databases have advantages over relational databases. Neo4j has an easily alterable schema, whereas relational databases such as MySQL do not allow changes to be made in such a simple way. Moreover, because of their maturity, relational databases often contain additional functionality such as logging, which contribute to larger administrative overheads than those of graph databases.

Finally, with regards to security, relational databases are much more sophisticated than graph databases. Neo4j is designed for trusted environments and requires that security be handled by applications. In contrast, modern SQL databases have robust security mechanisms that selectively handle access and permissions to database objects.

Although graph and relational databases are often compared and contrasted with one another, the line between the two database models is becoming increasingly blurred. Researchers at MIT have developed Vertica, a relational database used as a platform for graph analytics [3]. By integrating the graph and relational models, graph and relational analytics can be performed together on the data. For example, a graph analysis could be used to find paths between nodes, and relational analysis could summarize the results. Moreover, some graph queries are more easily expressed using relational algebra. Finally, such an approach can enable graph analysis with a backend that supports ACID. The same group has also developed Vertexica, a graph analytics tool for relational databases [4]. Because data in industry is often stored in relational databases, this approach removes the requirement that data be copied and imported into a new database management system (DBMS). Also, by building on relational databases, this approach also incorporates ACID support. Finally, the implementation features a vertex-centric query interface and a graphical user interface for ease of use.

Case Studies

Pregel

An important research endeavor is that of Pregel, Google's solution for large scale graph processing in a distributed environment [5]. The inspiration behind the research is crafting a graph engine that can effectively utilize Google's physical data infrastructure and Bigtable database model.

The model of computation requires an input of a directed graph with string vertex identifiers used to specify every vertex. Each vertex in the graph contains a value that can be altered by the user. After the system receives the input, it performs a sequence of supersteps until the algorithm completes and the result is returned to the user. During the execution of the supersteps, the system maintains

global synchronization points. During the execution of any given superstep, multiple vertices are computed concurrently according to the user's demand. A vertex can change its state, modify its outgoing edges, accept messages from the prior superstep, send message for use in the next superstep, or adjust the graph topology. At a high level, Pregel is oriented towards vertices, not edges. To determine when to terminate the algorithm, the system requires that every vertex votes to halt. At the first superstep, every vertex is active and deactivates when it votes to halt. A vertex is said to be inactive if it has no more computation required of it. Vertices can be reactivated if they are triggered by other vertices. For communication among vertices, a message passing model is used. The benefit to this approach is that it precludes the need for remote reads. Additionally, such a choice is associated with strong performance.

Pregel features a C++ API for ease of use. At each superstep, a Compute() function is executed for every vertex. This function can, in turn, obtain the values of a vertex using the GetValue() function, or change the value using the MutableValue() function. Messages in Pregel contain a value and destination vertex name. The destination vertex does not need to be a neighbor of the origin vertex. Because there is overhead associated with sending message, Pregel, contains a feature called combiners that allow multiple messages destined for the same vertex to be sent as a single message. Pregel also contains aggregators, which receive data from multiple vertices and generate output values using operations, such as sum, max, and min. Topology mutations are also supported, in which vertices are added and/or removed. Pregel is designed to be compatible with a number of input graph formats such as text files, relational databases, and Bigtable rows. To do so, the classes Reader and Writer are separate from the internal computation so users can create their own input / output interfaces.

Concerning the implementation of Pregel, Google datacenters consist of thousands of machines running commodity hardware. These machines are organized into clusters that are connected with one another, but are geographically distant. In Pregel, a graph is subdivided into partitions, each of which contains vertices, and the edges of those vertices. By default, vertices are placed into partitions based on their vertex ID using the function 'hash(ID) mod N', where N corresponds to the number of partitions. To perform computation across partitions, a number of steps are taken. First, the copies are made of the user program and are propagated to the cluster machines. One of the machines acts as the master and the others are the workers. Second, the master partitions the graph and determines which worker machines will handle any given partition. Third, user input is assigned to the workers by the master. Fourth, the master conducts a superstep. Finally, after the algorithm terminates, the master optionally tells each worker to save its results. The system allows for fault tolerance by using checkpointing. In checkpointing, the workers are instructed by the master to save partition states to persistent storage. Failures are determined by pinging machines and observing when attempted pings fail. Worker machines are responsible for a given part of the graph. They map vertex IDs to the vertex states and handle a queues of incoming messages. The master manages the actions of its workers by first assigning unique identifiers to them when it is registered. During execution, the master keeps track of the computation statistics such, as the size of the graph. Individual workers contain aggregators and send them values to the master at the end of each superstep. At the beginning of the next superstep, the master sends the global values of the aggregators to the workers.

Trinity

Another important research endeavor is Trinity, a distributed graph engine on a memory cloud, developed by researchers at Microsoft Research Asia [6]. At a high level, traditional disk-based graph engines have not satisfied the performance requirements of graph calculations and, memory-based designs have been constrained by the amount of memory of a single machine. The researchers detail a new approach in which a graph engine is distributed over multiple nodes that share a common memory address space (memory cloud).

From an architectural standpoint, Trinity utilizes a cluster of interconnected machines. These machines are arranged into a memory address space that is distributed and globally accessible, to facilitate large graphs. Trinity supports offline analytics, as well as online query processing. The individual parts of a trinity cluster are classified as clients, slaves, and proxies. Clients are applications that serve as the interface between end users and the Trinity system. These applications invoke Trinity library APIs to communicate with slaves and proxies. Slaves store data and perform computations on them. Proxies are an optional component that can be used to manage messages, but not store data. A typical use case for proxies is that of aggregation. To encode graphs into storage, the Trinity specification language (TSL) is used.

At the heart of Trinity is the distributed memory cloud. Backups of the cloud are stored in a shared distributed file system called Trinity File System (TFS), which bears similarities to Hadoop Distributed File System (HDFS). Data are stored in and retrieved from a key-value store, in which keys are 64-bit unique identifiers and the values are binary large objects.

With regards to data modeling, Trinity is designed to support graph data models from a variety of inputs such as XML, relational databases, and text files. When the value in a key-value store corresponds to model data in a graph schema, that value is said to be a cell and its corresponding key is termed the cell ID. This cell is considered to be Trinity's internal representation of a graph node. To incorporate edges, a given cell can contain the cell IDs of the neighboring nodes. If the graph is directed, there can be a set of cell IDs associated with the outgoing nodes, and another set of cell IDs associated with the incoming nodes. If the edges are rich in data, edges can be given their own cells.

Conclusion

From a theoretical standpoint, graph databases have a clear advantage over other types of data stores, including relational databases. When graphs are stored in relational databases, typically one relation will contain nodes, and another relation will contain edges. Despite this conceptual simplicity, graph traversals in this schema require a large number of costly joins, resulting in high latency.

From a practical standpoint, the actual commercial viability of graph databases is less clear. From an optimistic standpoint, graph databases are benefitting from an open source culture. Neo4j is open sourced on GitHub. Additionally, Apache Giraph, an open source implementation of Pregel is available to the public. Finally, Trinity is open sourced as 'Microsoft Graph Engine' on GitHub. With regards to commercial use, Giraph is used by Facebook to perform analysis on users and their connections. Additionally, Neo4j is used by eBay to rout deliveries and is used by Walmart to provide product recommendations to customers. However, the lack of a standardized query language and support for robust security features may impede widespread adoption. As such, a large real-world impact for graph databases has remained elusive. It appears that, for the foreseeable future, graph

databases will occupy niche markets in which the theoretical benefits of using a graph model over a relational model are clear and the client firm has the resources to manage a project using nascent technologies.

Resources

[1] Angles, Renzo. "A comparison of current graph database models." Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on. IEEE, 2012.
[2] Angles, Renzo, and Claudio Gutierrez. "Survey of graph database models." ACM Computing Surveys (CSUR) 40.1 (2008): 1.
[3] Jindal, Alekh, et al. "Graph analytics using vertica relational database." Big Data (Big Data), 2015 IEEE International Conference on. IEEE, 2015.
[4] Jindal, Alekh, et al. "Vertexica: your relational friend for graph analytics!." Proceedings of the VLDB Endowment 7.13 (2014): 1669-1672.
[5] Malewicz, Grzegorz, et al. "Pregel: a system for large-scale graph processing." Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010.
[6] Shao, Bin, Haixun Wang, and Yatao Li. "Trinity: A distributed graph engine on a memory cloud." Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. ACM, 2013.
[7] Vicknair, Chad, et al. "A comparison of a graph database and a relational database: a data provenance perspective." Proceedings of the 48th annual Southeast regional conference. ACM, 2010.