Job references, hyperlinks, phone calls, friend requests, emails, pings, meetups, family, financial transactions and the list goes on and on — data is not only getting bigger, it is becoming more connected. Not all connections are made equal.

Many of your connections say something interesting about you. Consider the simple example, in which a Person can be connected to a Building by various edge types, and then think about what those edge types mean.

This is the value that relationships have. They might seem to represent mundane activities, but can reveal patterns of behaviour that give significant value to the owners of the data. What if the person eats at a fast food restaurant frequently? They may be interested in coupons for other fast food restaurants. Conversely, what if the person is very well connected and they shop at a store? It is possible that their friends may be interested in that store. In the same way, if that person is a known bank thief and they frequent a deli

<table>
<thead>
<tr>
<th>Edge Type</th>
<th>Might Indicate about the person</th>
<th>Might indicate about the Building</th>
</tr>
</thead>
<tbody>
<tr>
<td>LivesAt</td>
<td>They have a home</td>
<td>It is an apartment building</td>
</tr>
<tr>
<td>EatsAt</td>
<td>They like to eat out</td>
<td>It is a restaurant</td>
</tr>
<tr>
<td>WorksAt</td>
<td>They have a job</td>
<td>It is a company</td>
</tr>
<tr>
<td>ShopsAt</td>
<td>They have money to spend</td>
<td>It is a store</td>
</tr>
</tbody>
</table>

Table 1: Graph inferencing
next to a bank, this might be a valuable lead for the police. These three scenarios highlight very popular use cases for graph databases: advertising/marketing, social networks or network management, and fraud/criminal detection.

But just what is a graph database? A graph database is a native storage engine that enables efficient storage and retrieval of graph structured data. Graph databases are typically used:

1. When the data source is highly connected.
2. Where the connections are important (add value to the data).
3. When the user access pattern requires traversals of those connections.

Graph databases are different than other types of databases because they have a unique data model (vertices and edges) and they are used in unique applications built around graph use cases. Unlike graph analysis engines or visualization frameworks, graph databases are optimized around concurrent access of persisted data, so users can navigate the data as it is being added or updated. Consider a recommendation engine based on a social graph (Figure 1) where the purchasing patterns of your “friends” or “neighbours” are used to suggest merchandise that you might be interested in buying.

In order to identify popular items to recommend, the recommendation engine could traverse the connections in the data to find interesting patterns of purchases. You might only be interested in data pertaining to certain types of People: students, perhaps, or women over the age of 40 who own a home. This navigational query can be quite complex, but it is possible to execute it using an API designed for navigational queries.

Reading Distributed Graph Data

Distributing data is the practice of storing data in multiple physical locations. There are many reasons why someone would want to distribute their data using a distributed database technology, but the most notable are scalability and data localization. Data localization represents the idea of storing data where it most makes sense, which is usually local to the applications that are accessing it most frequently. Also, the ability of a database to distribute data into partitions is critical to their ability to horizontally scale.

Any distributed graph database engine faces a difficult problem though. Accessing highly connected data in a distributed environment commonly means going back and forth between partitions of data to retrieve neighbouring data. This can significantly reduce the performance of a navigational query.

To curtail this issue, there are two common strategies. The first is to partition your graph to isolate highly connected sub-graphs to minimize the number of calls across the network and utilize local cache. In other words, place the vertex objects in a way that reduces the number of edges that cross between hosts. This subgraph isolation can be achieved with custom placement strategies.

The second strategy involves using a distributed navigator. A distributed navigation engine will distribute the processing load for traversing subgraphs to the partition on which they are located.

Custom Placement

InfiniteGraph is a distributed graph database that is designed to handle large data sets and get the highest performance for accessing your data. Using your own data model, InfiniteGraph lets you build a custom placement model that can achieve the isolated graph partitioning that is so important for achieving high-performance navigational queries in a distributed environment. Consider the case where you are placing medical data for hospitals and patients. Knowing that doctor data is related to and accessible through hospitals, it makes sense to physically co-locate instances of hospitals and doctors. Likewise, you would want to physically co-locate the patients and their visits. This would then give both a logical and natural type of placement strategy and would achieve a high degree of isolation for graph partitioning. This sort of custom placement is easy to implement in IG.

Distributed Navigation

Google’s Pregel is a standard for large scale graph processing in a distributed environment that was originally used for
the Google cluster. For a number of years, performing large-scale graph analysis in distributed systems was done mostly by pulling in all remote data and processing it locally. The problem with this approach is poor performance and a lack of scalability. To tackle this problem, Pregel enabled the execution of graph algorithms (like PageRank and Shortest-Path) in a map-reduce fashion by:

1. Enabling each vertex object to send/receive messages.
2. Executing an algorithm in a sequence of super-steps where a function would run at each vertex in parallel.
3. Possibly synchronizing the results at each step.
4. Passing a message to the next vertex in the path until the algorithm terminates and returns the results.

This system of graph analysis in a series of steps, where each vertex sends a message to the next vertex in the path, is scalable and optimized in a distributed system. If the vertex is located on a remote host, the processing will be done on that remote host. At the same time though, this sort of processing can be limited because it relies on the distributed data architecture. Even if the algorithm is only required to process locally, it uses the same logic of sending messages from vertex to vertex, which is only optimal for use in a distributed environment. The IG navigation engine utilizes the Pregel logic for graph processing in distributed environments while at the same time sidestepping the limitations of processing locally.

Similar to Pregel, when the path continues on another host, the distributed navigation engine can send the partial path to a remote query server to continue the navigation on that host and aggregate the results at the end. But unlike Pregel, if the navigation is localized to the host that the process is currently running on, the traversal will continue to be executed in memory instead of sending messages. Also, in an effort to maximize the performance of the navigation, the query engine will intelligently cache remote objects when they are accessed frequently.

**GraphViews**

Another unique and very powerful feature of IG’s navigation API is the ability to execute a complex query using a GraphView. A GraphView enables you to build filtered views of the data on top of which you can execute a traversal. The GraphView can be constructed simply by excluding/include vertex and edge objects based on their type. The following shows how to include a base type (Person), but exclude a derived type (Student).

Alternatively, using a GraphView, you can exclude/include vertex and edge types based on their type and a predicate. The following shows how to include a type (Person) based on a predicate. Note: The fields “gender”, “age” and “ownHome” are all members of the Person class.

GraphViews effectively reduces the size of the data set by allowing the navigator to ignore paths that don’t lead to desired results. When a GraphView is used inside of InfiniteGraph’s accelerated and distributed graph traversal engine, it
allows users to get real-time results over large and highly connected data sets.

In these examples above, “myGuide” is a Guide which defines ordering behaviour like Breadth-First or Depth-First traversal and “pathQualifier” and “resultQualifier” are instances of a Qualifier which defines behaviour around qualifying paths and results, respectively. Also, “myPolicies” is a PolicyChain, set of navigational policies, that can be used to define the global behaviour of the navigator. For example, the MaximumResultCountPolicy sets the maximum number of paths returned and the EdgeTraversalPolicy defines which edge types can be traversed. Finally, the “resultHandler” is an instance of the NavigationResultHandler which processes the paths that are returned by the navigator. It will outline actions around the results that are returned which may include things like writing to a file, aggregating and analysis, or kicking off another navigation.

### Supernodes

In Graph Theory, a “supernode” is a vertex with a disproportionally high number of connected edges. Supernodes make it difficult to do a navigational query in real-time due to the amount of effort it may be to pursue paths through it that may be unfruitful. Consider a celebrity in the social graph dataset who has a higher than normal number of followers. Avoiding the celebrity node or limiting the number or types of edges that you traverse when going through celebrity nodes can significantly increase the performance of the navigation.

With InfiniteGraph, we offer two strategies to addressing the supernode problem. First, you can use GraphViews to exclude vertex types to avoid the supernodes in the graph or exclude edge types to limit the number of paths traversed through supernodes. Using the celebrity supernode problem described above, you can exclude the celebrity objects themselves or you exclude vertex objects that have a certain amount of followers. Secondly, we offer a policy called a FanoutLimitPolicy which will limit the number of edges that you traverse for any given vertex during a navigational query. These two strategies, especially the GraphView, will increase the performance of the navigation by orders of magnitude.

### Writing Graph Data

Consider the problem of writing highly interconnected data to a graph database. When an edge is added, both sides of the edge (source and target vertices) must be updated in a single transaction. This can lead to lots of lock collision around vertices with high connectivity (such as V2 below) because edges that share this common vertex (E12 & E23) might be added in different write transactions and need to update the shared vertex (V2) at the same time.

InfiniteGraph offers a high-performance indirect ingest option called Accelerated Ingest in addition to the traditional direct ingest option. Using the indirect method, you can achieve a much higher rate of ingest, but with relaxed consistency, temporarily sacrificing access to the “full picture”. In the case above, all of the graph components will be placed immediately, but the edge updates to the vertex objects (V1, V2, and V3) will be processed asynchronously in batches which are still atomic, isolated, and durable, but are phased. This will allow for a contention free, high performance ingest which is perfect for applications or systems that are not as concerned with getting an immediately consistent state.

```
Listing 1
```

```java
graphview myView = new GraphView();
myView.includeClass(myGraphDB.getTypeId(Person.class.getName()));
myView.excludeClass(myGraphDB.getTypeId(Student.class.getName()));
Navigator myNavigator = myStartVertex.navigate(myView, myGuide,
pathQualifier, resultQualifier, myPolicies, resultHandler);
```

```
Listing 2
```

```java
graphview myView = new GraphView();
long personTypeId = myGraphDB.getTypeId(Person.class.getName());
myView.includeClass(personTypeId, "gender=='F' && age>40 && ownHome==true");
Navigator myNavigator = myStartVertex.navigate(myView, myGuide,
pathQualifier, resultQualifier, myPolicies, resultHandler);
```
Visualization Tools and Toolkits

Graph data sets are more easily understood when viewed and browsed with some kind of visualization framework and visualization tools or toolkits are usually distributed with a graph database. There are also many third party vendors that support visualization with graph data sets including Tom Sawyer, KeyLines, Graphviz and D3js. InfiniteGraph offers a visualization tool called the IG Visualizer which is a standalone tool that can connect to any instance of an IG database and offers a rich development environment to view/traverse the data, use indexes, and customize views and layouts. Many of the features discussed in this article including configuring graph views and executing a distributed navigation, can be done through the IG Visualizer tool.

If you have a high connected data set and use cases that seem to fit the need for a graph database, consider InfiniteGraph. InfiniteGraph is a leading distributed graph database with a simple Java API. It is built on top of Objectivity/DB, a proven object-oriented data storage engine that is known for specializing in connections between data. InfiniteGraph supports direct and high-performance writes with custom placement models, a complex distributed, parallel query engine that allows traversals through a native navigation API, and fully ACID transactions that allow concurrent access to the data. To get more information or resources, visit the Objectivity website [1] or the InfiniteGraph wiki page (wiki.infinitegraph.com).

References

[2] wiki.infinitegraph.com

Nick Quinn is the Principal Engineer for InfiniteGraph, a distributed graph-oriented data management solution from Objectivity Inc. Since joining Objectivity, Nick has been work exclusively on the InfiniteGraph project and has played a key role on the design and architecture of InfiniteGraph 3.0 and 3.1. Prior to joining Objectivity in 2010, Nick worked as a lead Java Developer at Savi Networks, a Lockheed Martin Company. Nick holds a Master’s Degree in the College of Engineering from Santa Clara University in Santa Clara, California.