Graph Database Proof of Concept Report
Managing The Internet of Things
# Table of Contents

Executive Summary .................................................. 3  
Background ................................................................. 3  
Proof of Concept .......................................................... 4  
  Dataset ........................................................................ 4  
  Process ....................................................................... 4  
  Query Catalog ............................................................. 4  
  Environment .................................................................. 5  
    Internal data center ..................................................... 5  
    EC2 environment ......................................................... 5  
  Data model .................................................................... 6  
  Performance Test Results ............................................... 6  
    Data Import .............................................................. 6  
    Query Performance ..................................................... 9  
    Scalability ................................................................ 13  
    Results by Query Type ................................................ 13  
Conclusions ................................................................. 16  
Appendix ........................................................................ 17  
  Dataset ........................................................................ 17  
  InfiniteGraph Features .................................................. 19  
    GraphView ................................................................ 19  
    Qualifiers .................................................................. 19  
    Predicates .................................................................. 19  
    Navigation Policies ..................................................... 19  
    Distributed Navigation ............................................... 19  
References ....................................................................... 20
Executive Summary

The amount of data is exploding. In order to maintain competitive advantage, businesses are looking to discover new ways to add value to data, beyond traditional business intelligence and analytics.

The next level of data management must take into account the “Internet of Things” (IoT), in which everyday objects have network connectivity, allowing them to send and receive data. The implications of the IoT is that every data point, every object, has value and influence within any given network that it may be connected to.

In this PoC, we look at using InfiniteGraph as the graph database that enables analytics and connectivity across machines, data, and people in order to optimize asset management and operations, or the IoT. An ACID compliant, scalable, and high performance graph database was a requirement for this PoC. Ultimately, in this use case, InfiniteGraph will enable the management of complex industrial assets and their connections across multiple industries.

Understanding the implied associations between data points (nodes) and their relationships with each other (edges) by utilizing scalable object and/or graph database technologies will enable innovative in-time business insights for companies to stay ahead of the technology curve, develop valuable new services to their customers and multiply growth opportunities.

Background

This graph database Proof of Concept (PoC) was designed with the objective of testing query performance and scalability as well as additional database features for an asset management use case. Through a thorough analysis of the generated dataset, careful design of graph queries, and implementation on multiple hardware configurations, Objectivity demonstrated high ingest rates during graph build, sub-second graph query response times with no memory bound, and high scalability across multiple machines in large clusters for its InfiniteGraph graph database solution.

The dataset simulates asset management data and how assets (nodes) relate to each other, either as a hierarchy (bill of materials) or adjacency relationships (edges).

During the PoC process, we performed several tests ingesting data into InfiniteGraph from MySQL for building graphs of various sizes. We achieved rates of up to 390K vertices per second and 500K edges per second over a 16 machine cluster from a generated MySQL raw data source of one billion vertices and over one billion edges.

After performing the stress tests using limited memory, we also demonstrated that there is no obvious correlation between query performance and the ratio of the system memory to the graph size.

The scalability tests included an 8 machine cluster and a 16 machine cluster. We found that the only difference was our distributed navigation server which processes part of the graph locally on each machine and only sends the query results to the machine issuing the query resulting in faster response times.

Overall, almost every query type tested under every scenario and configuration had sub-second average response times even after removing query results that returned no data.
Proof of Concept

Dataset
A dataset generator based on Facebook’s LinkBench benchmark\(^1\) was provided for this PoC. The generator creates a dataset of vertices and edges in a MySQL RDBMS that follow a power-law distribution similar to that found in Facebook’s own social graph. The generator can be configured to generate different volumes of data. The generated MySQL database is used as data source for Objectivity during the PoC. Detailed analysis of the data is in Appendix 1.

We included one type of data redundancy in our PoC implementation. In the data, the same type of edge could connect the same two vertices more than once, with a difference in the timestamp. Instead of pruning those edges, we included all of them to enrich the graph with edge attributes that could represent time-based connectivity manipulation of the graph.

Process
The following steps were required in the PoC process:

1. Data Preparation
2. Data Import
3. Uniqueness Index
4. Uniqueness violation
5. ACID Transactions and Read Concurrency
6. Stream-Based Queries
7. Hierarchical Materialization
8. Federated Queries
9. Performance Tests (query performance and scalability tests)

In addition, a list of success criteria was provided, for example

a) **Query Performance**: using provided query catalog and using different levels of limited memory, different indexing scenarios, cold vs warm scenarios

b) **Scalability**: test query performance for graph queries that visit different number of machines (1, 2, 4, 8, 16) in a cluster

We focused the PoC effort on the above two criteria since they are considered critical when working with a large graph system.

Query Catalog
For the query performance and scalability tests, the following types of queries were used:

1. Transitive Closure
2. Shortest Path
3. Path Filtering
4. All Paths
5. Subgraph Similarity

\(^1\) except for the “All paths” query type which is an exhaustive graph query which returned averages between 2.7 and 3.1 seconds
<table>
<thead>
<tr>
<th>Transitive Closure</th>
<th>Find the total amount of material used in a certain product including all its sub parts.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortest Path</td>
<td>What is the path to a device from the nearest service node.</td>
</tr>
<tr>
<td>Path Filtering</td>
<td>Find path between two nodes. The path could have constrains such as type of connectivity, degree of separation... etc.</td>
</tr>
<tr>
<td>All Paths</td>
<td>Find all possible paths between two nodes.</td>
</tr>
<tr>
<td>Subgraph Similarity</td>
<td>Compute degree of similarity between two sub graphs based on the node attributes, for example, type of materials used.</td>
</tr>
</tbody>
</table>

**Environment**

Given the requirements for the PoC, Objectivity ran the tests in a mixture of environments including Objectivity’s own internal data center and Amazon’s EC2 machines in the cloud.

**Internal data center**

It consists of a cluster of blades that is configured to implement the memory stress tests where we generated enough data to cover the 10% to 100% memory to graph-size ratio.

**EC2 environment**

We utilized the Amazon Cloud to test the scalability and performance of both data ingest and data query using 8 and 16 machine clusters. This configuration was used to load the 1 billion vertices raw graph into InfiniteGraph (IG) and also test the various query requirements based on the query catalog provided.

All machines were placed in the same subnet group and in the same zone to minimize any possible network and disk access latencies.

We used the Amazon EC2 RDS service to create the MySQL instance using the following configuration:

- Instance Class: db.m3.2xlarge
- Storage: 1000GB
- IOPS: 8000

It took about 10 hours to generate the raw data in MySQL with the above configuration.

Because of the time constraints and the various configurations we needed to test, we utilized a key/value store (Redis) to allow us to store the raw IDs to InfiniteGraph vertex IDs as we ingested the vertices. This facilitated a very fast link ingest. To be able to accommodate the large key value store we used two large memory instances (r3.8xlarge) on EC2. These instances were only used during the initial load and they are not part of the query tests.

For the clusters, we used the same configuration for all nodes, with the only difference in the EBS attached disk used for the data, where on nodes 1-8 we used 100GB, and on nodes 9-16 we used 50GB EBS. With the exception of machine number one that used another EBS disk to store the system database. The instance class used is the c3.2xlarge, which has 15GB of memory and 20GB of local disk.

We also used a separate machine to drive the processing on the cluster and to collect that data generated during ingest and query. The driver machine was a general instance with enough local disk memory to store the results.
**Data model**

The following is the graph model used during the PoC. The model is very simple. In more sophisticated systems, the model could be enhanced with other types that could be used more intelligently during the navigational query. During the ingest we parsed the payload and set the correct attribute values on each created vertex and edge.

**Performance Test Results**

In this section, we describe the configuration of the environments and datasets we set up to execute the different performance tests required by the PoC and the corresponding results.

**Data Import**

The data ingest process was split into two stages. The first, read all rows from the nodes table and created vertices in InfiniteGraph, during which we stored the row ID to InfiniteGraph ID mapping in Redis. We also stored the Asset ID to eliminate duplicate assets from being created. This helps in speeding the processing of data. The second, stage reads the linktable rows, lookup the InfiniteGraph IDs based on the row ID, and then create edges as needed with the timestamp attribute.

We used the same configuration to load into the 8 machine cluster and the 16 machine cluster.

During data ingest into InfiniteGraph, we used the following configurations:
Four processes on each machine for vertex ingest and 2 processes for edge ingest.

Our edge ingest process used the edge pipelining method of ingesting which allows agents to run in the background and insert the connectivity between vertices and edges in a more efficient way. The agents run in parallel with the edge ingest processing and they finish processing their task within 2-10 minutes of the completion of the edge processing.

Number of vertices generated: 779 million
Number of edges generated: 720 million

Data Ingest Performance per type of Cluster:

<table>
<thead>
<tr>
<th>Type of Cluster</th>
<th>Data Ingest Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 machine cluster</td>
<td>~3 hours</td>
</tr>
<tr>
<td>16 machine cluster</td>
<td>~2 hours</td>
</tr>
</tbody>
</table>

Overall rate

The following graph shows the aggregation of the rates from each machine.

Average Rates [8 machines] ~ (220K/sec)     Average Rates [16 machines] ~ (390K/sec)
Average Rates [8 machines] ~ (290K/sec)  Average Rates [16 machines] ~ (500K/sec)

**Machine rate**

This is the aggregation of the rates from each process running on the machine.
Process rate
This rate is measured when importing/processing 10K SQL IDs for vertices. The maximum number of vertices ingested is 10K.

For edges, we started with a range of 10K SQL IDs. This varies drastically for edges depending on local density of the graph. An approximate range of 3K to 50K edges per transaction was seen during the ingest.

Query Performance
Using the exact descriptions of each query in our initial query tests did not reveal any results due to lack of matching cases in the dataset. The performance was blazingly fast, since some of the restrictions were pruning the graph early on.

To be able to satisfy the required query and get some tangible results, we relaxed some of the restrictions (which made the search space larger) to get results that could show the scalability and the performance of InfiniteGraph.
We also did a study on the initial dataset to select areas with connectivity. We used a set of the Asset IDs that have potential connectivity, then navigated from such set to generate enough path samples that helped us pick the right Asset IDs for the various types of queries.

**For example:**
For transitive closure, there is a restriction on “metal” and the query is related to a navigation from “a” for incoming edges “b” (which is part-of “a”). When we did the analysis, we found that the most you get is 5 hops for incoming edges. We also relaxed the directionality by using both in and out so we can get tangible results.

Various advanced graph query features were used. See Appendix 2 for detailed description.

1. GraphView
2. Pre-defined qualifiers
3. Predicate qualifications on properties

To be able to measure various aspects of the system, we ran each query normally, i.e. by the client application, and distributed, i.e. utilizing the distributed query server. Each run was done twice to measure the warm and hot cache effect on performance.

**Stress Test (Limited Main-Memory)**
The purpose of this test is to determine the performance of various queries when the graph database does not fit in memory. To determine the graph size (G) we define the following relationships:

\[
V = \text{Number of Vertices} \\
E = \text{Number of Edges} \\
Ef = \text{Ratio of Edges to Vertices} = \frac{E}{V} \\
\text{sizeof(}V\text{ertex}) = 4 \times \text{sizeof(integer)} + \text{sizeof(long)} + \text{sizeof(pointer)} + Ef \times \text{sizeof(pointer)} \\
\text{sizeof(}Edge\text{)} = \text{sizeof(long)} + \text{sizeof(pointer)} + 2 \times \text{sizeof(pointer)} \\
G = V \times (\text{sizeof(}V\text{ertex}) + Ef \times \text{sizeof(}Edge\text{)})
\]

And, finally, we define the memory ratio (R) as follows:

\[
R = \frac{M \text{ (available memory on system)}}{G \text{ (graph size)}}
\]

The query server was shut down for this test. In other words, all the navigation was performed by the PoC application. The dataset resided on multiple disks on a single blade machine.

The following constant values were used for the tests:

\[
Ef = 2 \\
M = 8 \text{ GigaBytes}
\]
Various values were tried for the Java heap size ranging from 2GB to 4GB. Similarly, InfiniteGraph’s cache size was varied from 500MB to 2GB. The results shown below are when the heap size is limited to 2GB and InfiniteGraph cache size to 500MB.

**Note on Memory Usage for the Various Queries**

For a lot of the cases, InfiniteGraph navigation requires minimum memory to perform the basic operation. The vast amount of the heap memory usage is utilized by the query application to hold some transient representation of the result and other application level data structure for bookkeeping purposes. The following table lists the memory usage for various queries.

<table>
<thead>
<tr>
<th>Query Type</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transitive Closure</td>
<td>An application level hash map is used during the navigation to keep track of the target vertices that have been found to be matches. This is required because you can have multiple paths to the same vertex, but you are interested in accounting for the same vertex only once.</td>
</tr>
<tr>
<td>Shortest Path</td>
<td>There is no application level data-structure that is required. The key point for this query is that breadth first search is used. This sort of guide requires as much memory as the most connected node in the graph (your super nodes).</td>
</tr>
<tr>
<td>Path Filtering</td>
<td>A data structure is required to maintain a transient version of the paths that are found.</td>
</tr>
<tr>
<td>All Paths</td>
<td>Same as path filtering.</td>
</tr>
<tr>
<td>Sub-graph Similarity</td>
<td>Similar to Transitive Closure, a hash map is required to maintain unique target vertices. In addition, data structures are used to maintain the sets of part numbers for the various assets.</td>
</tr>
</tbody>
</table>

**Using Policies to Circumvent Memory Constraint Issues**

The following policies are key to working on a system that has memory constraints:

- **Maximum Breadth Policy**: Limits the maximum size of a breadth first navigation. It can be useful for shortest path queries.
- **Maximum Memory Use Policy**: Limits the memory used for navigation processing. It can be useful for shortest path queries.
- **Maximum Path Depth Policy**: Limits the depth from the starting vertex. It will be useful for all queries.
- **Maximum Result Count Policy**: Stops the navigation when a certain number of qualifying results/paths are found. It will be useful for all queries.
Query Results
All times include finding the originating vertex, time to perform the navigation and the time to gather all the results. The results are based on running several queries and taking the results of the queries that have at least 2 unique targets. For the shortest path only, a single result is required.

<table>
<thead>
<tr>
<th>Query</th>
<th>25%</th>
<th>50%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transitive Closure</td>
<td>Range (0.4 - 3.2)</td>
<td>Range (0.3 - 4.7)</td>
<td>Range (0.4 - 3.8)</td>
</tr>
<tr>
<td></td>
<td>Average 0.69</td>
<td>Average 0.67</td>
<td>Average 0.70</td>
</tr>
<tr>
<td>Shortest Path</td>
<td>Range (0.3 - 1.2)</td>
<td>Range (0.4 - 1.5)</td>
<td>Range (0.4 - 0.9)</td>
</tr>
<tr>
<td></td>
<td>Average 0.47</td>
<td>Average 0.59</td>
<td>Average 0.41</td>
</tr>
<tr>
<td>Path Filtering</td>
<td>Range (0.3 - 4.1)</td>
<td>Range (0.4 - 4.4)</td>
<td>Range (0.4 - 3.9)</td>
</tr>
<tr>
<td></td>
<td>Average 0.72</td>
<td>Average 0.77</td>
<td>Average 0.67</td>
</tr>
<tr>
<td>All Paths</td>
<td>Range (0.6 - 12.1)</td>
<td>Range (0.6 - 11.3)</td>
<td>Range (0.4 - 11.7)</td>
</tr>
<tr>
<td></td>
<td>Average 2.7</td>
<td>Average 3.1</td>
<td>Average 2.9</td>
</tr>
<tr>
<td>Sub-graph Similarity</td>
<td>Range (0.6 - 3.6)</td>
<td>Range (0.7 - 3.2)</td>
<td>Range (0.6 - 3.2)</td>
</tr>
<tr>
<td></td>
<td>Average 0.9</td>
<td>Average 0.9</td>
<td>Average 0.8</td>
</tr>
</tbody>
</table>

Based on the above table, we can safely state that for the given dataset there is no obvious correlation between the query performance and the ratio of the system memory to the graph size.

Starting Vertex Selection
The starting vertices for the navigation were selected based on the graph density distribution within a given SQL ID range. A sample distribution for a range of 100K nodes results in the distribution shown below. Nodes with at least two edges were sampled from various sections of the distribution.
The selection based on connectivity is sufficient for queries like transitive closure and similarities. For queries that require at least one valid path to exist between them, we performed an explorative navigation without any restriction to capture the target nodes before starting any of the tests queries.

Indexed vs. Non-Indexed

With InfiniteGraph, indexes are used only to find the initial vertices (nodes) to initiate the navigation. There is no need to index on the edges or edge types since they are processed as we navigate the graph. Objectivity/DB, which is the backend store for InfiniteGraph, persists the connection with each vertex (node).

The memory constraint queries above include looking up the initial vertices in the index. In the following section we only show the navigation performance on the cluster and we eliminated the overhead of looking up the initial vertex.

Scalability

For the scalability test, we built two graphs, one over a cluster of 8 machines and another over a cluster of 16 machines.

We ran the same query tests described above with various concurrency models i.e. we ran 4 sets of queries on one of the machines in the cluster, then we repeated the process by using 2 machines, then 4 machines.

Each set was picked from a different SQL row ID ranges. Based on our understanding of the raw data, this would allow us to perform queries that touch different parts of the graph.

For the Transitive Closure and Similarity tests, we measured the average time performing the navigation, and for the Path tests we showed each navigation time, with tests showing hot vs. warm numbers.

(Note: We used the term warm instead of cold, since the system was not restarted prior to processing; only the Infinite-Graph cache was cleared, so we do start from a cold cache, but a warm system.)

We eliminated results that did not yield any value or did not produce tangible results.

We found that the results are very similar when running the query on one machine or on 4 machines while the graph is distributed on an 8 or 16 machine cluster.

The only obvious difference was the advantage provided by our distributed navigation server which processes part of the graph locally on each machine and only sends the results to the machine issuing the query.

We believe that when running the test on one machine, we are satisfying the requirement of having 10% or less of the graph in memory ratio. The results we have seen are comparable to the table above for the total time and average time for query.

Results by Query Type

The following section shows select results from our query performance tests by each of the five query types from the catalog. We collected results for all cluster and graph configurations, but most of them will be showing similar patterns, so we picked the 16 machine cluster results for reference since it was the most demanding from the scalability perspective.

Each query type was pushed to 4 nodes to run in parallel to simulate an operational usage of the system.

Each node performed the query four times, two with local processing on the node and two using the parallel query server (processing local to the data and not the initiating process). This will simulate the warm machine usage (no Infinite-Graph caching) and the hot machine usage (the data is in the cache) scenarios, which is typical in operational systems.

As shown below, utilizing the parallel query server does benefit the query operation since the data is processed locally on each machine and only the results are pushed to the query processing server. We also notice the difference in warm
vs. hot operation, where disk I/O is eliminated.

The time measure in the graphs does not include the lookup of the initial vertex node to perform the query, it only measure the navigation and processing time of the graph.

**Transitive Closure**

As mentioned before, we relaxed constraints to allow for more data to be found. The vertices participated in the results ranged from 2 to 3126 with an average of 96.5.

![Transitive Closure on 16 node cluster](image)

**Shortest Path**

![Shortest Path on 16 node cluster](image)
Path Filtering

All Paths
Conclusions

In this Proof of Concept, Objectivity demonstrated that InfiniteGraph can achieve high ingest rates during graph building, sub-second graph query response times with no memory bound, and high linear scalability across multiple machines in large clusters where a large graph is distributed.

We derived the following conclusions from the PoC:

- The simplicity of the data model (two edge types, one node type, very few attributes) and limited connectivity among vertices in the dataset did not allow Objectivity to exploit important features in InfiniteGraph such as its placement model to demonstrate even better performance and scalability.

- The data import tests demonstrated linear scalability as we achieved proportionally higher ingest rates as we increased the number of machines in a cluster.

- The stress tests, using limited memory, demonstrated that there is no obvious correlation between query performance and the ratio of the system memory to the graph size.

- The scalability tests found that results are very similar when running the queries on one machine or on 4 machines while the graph is distributed on an 8 or 16 machine cluster. The only significant difference in configuration was our distributed navigation server which processes part of the graph locally on each machine and only sends the query results to the machine issuing the query resulting in faster response times.

- Also, the scalability tests found that results are very similar when running the query on one node or on 4 nodes while the graph is distributed on an 8 or 16 machine cluster.

- Overall, almost every query type tested under every scenario and configuration had sub-second average response times even after removing query results that returned no data.

\[2\] except for the “All paths” query type which is an exhaustive graph query which returned averages between 2.7 and 3.1 seconds
Appendix

Dataset

The density of the graph varies for various sections of the dataset. For instance, if we take a sample of outgoing connections for the first 1 million nodes (SQL ID), we would get a plot shown below.

![Number of Outgoing Edges](image1)

The number of nodes, which contribute to these output-going edges, is a constant for the complete graph; it is about 14% to 15%. For the first 1 million nodes (SQL ID), the plot is shown below.

![Number of Nodes with Outgoing Edges](image2)
The number of nodes with incoming edges is uniformly distributed across the sections of the graph.

The dataset consisted of a constant number of nodes, which are self-referencing (linking to themselves). These relationships were ignored when importing the dataset to the InfiniteGraph database. The plot below shows the number of self-referencing relationships.

Another important consideration is the proximity of a node to connect to another. It was found that a node connects to another within a constant range (approximately a 2K SQL ID range). The plot below shows a sample for SQL ID between 10K and 30K ID range.
InfiniteGraph Features

**GraphView**
The Graph View is an important feature that lets you define a subset of graph vertices and/or edges on which to run a navigation query, thus, improving the overall performance. We used the Graph View during navigation to filter on edge types, for example, PartOf instead of AdjacentTo. For a more sophisticated model with various vertex types, the Graph View can be used to focus the navigation on the types of interest to the query and filter out all other types. This would prune graph branches earlier on during the navigation.

**Qualifiers**
InfiniteGraph comes with a set of advanced internal qualifiers that could be utilized to accomplish various types of navigation more efficiently, see [http://wiki.infinitegraph.com/3.2/w/index.php?title=Using_Built-In_Qualifiers](http://wiki.infinitegraph.com/3.2/w/index.php?title=Using_Built-In_Qualifiers) for details. Aside the pre-defined qualifiers, InfiniteGraph allows users to define their own custom qualifiers to implement any kind of graph algorithm needed.

During the PoC, we utilized the VertexPredicate and the VertexIdentifier qualifiers, the latter was used for path between points, i.e. to a specific target and the former used during the subgraph navigation.

**Predicates**
InfiniteGraph has a very strong predicate language for query objects (vertexes). The predicate language can also be used during the navigation to apply filters to the vertices and edges as we navigate. For example, we used predicate to filter on the metal types during navigation. In the latest release of InfiniteGraph, the predicate language was extended to include graph operators that allow users to define navigational path matching to the qualifiers.

**Navigation Policies**
The navigation policies are used to influence the behavior of a navigation query. For example, a navigation policy can be used to set the maximum path depth for navigation or specify a fanout limit to prohibit vertices with too many outgoing edges from being traversed.

**Distributed Navigation**
Distributed navigation provides parallel execution of different branches in the navigation query. Such processing is done locally on the data node and only the results are provided to the initiating client. The execution is controlled by a navigation policy and requires running the parallel query server on each data node.
References


Objectivity, Inc.
3099 North First Street
Suite 200
San Jose, CA 95134
Office: 408-992-7100
Fax: 408-992-7171
Info@Objectivity.com