InfiniteGraph: The Distributed Graph Database

A Performance and Distributed Performance Benchmark of InfiniteGraph and a Leading Open Source Graph Database Using Synthetic Data
### Table of Contents

- High-level Overview – Open Source Graph Database Architecture .................................................. 4
- High-level Overview – InfiniteGraph Architecture ........................................................................... 5
- Performance Benchmarks ...................................................................................................................... 6
  - Graph Construction (Ingesting Vertices and Building Relationships) ............................................. 7
  - Sampling for 64 Search Keys Based on a Unique Indexed Property (Vertex Lookup) .................. 11
  - Reading Graph Objects from the Database ...................................................................................... 12
  - Graph Navigation/Traversal Using Breadth-first-search and Depth-first-search ....................... 12
- Distributed Performance Benchmark of InfiniteGraph ........................................................................ 14
- Graph Construction .............................................................................................................................. 14
  - Ingesting Vertices ............................................................................................................................. 15
  - Ingesting Edges ................................................................................................................................. 16
  - Finding Vertices ............................................................................................................................... 17
  - Graph Traversal ............................................................................................................................... 20
- Conclusion ............................................................................................................................................ 20
Introduction:

In our expanding digital ecosystem the data gathered by companies is not only increasingly complex and interconnected, but there is much more of it. In fact, IBM estimates that 90% of the data in the world today has been created in the last two years alone. Corporate databases have gone from being measured in gigabytes to terabytes and even petabytes, driven by web traffic, social media, financial transactions, email, phone calls, IT logs and more. Buried in this mountain of data is intelligence that can be used to shape strategy, improve business processes and increase profits. Unfortunately, the ability to understand and quickly act on extremely large datasets, or Big Data, has proven to be a big headache for many companies.

*Leading analyst firm, Gartner, reports global enterprise data assets to grow by an additional 650 percent by the end of 2014.*

Technologies such as those in the NOSQL space were developed to deal with large-scale data needs and the storage capacity limitations of traditional relational databases. Using a NOSQL approach, data can be interconnected in ways that, when organized properly, unlock valuable information. These solutions can be divided into four different technology categories: Key-value databases, column family databases, document databases and graph databases.

**Key-value Databases.** A Key-value database, or key value store, is similar to a relational database with rows, but only two columns. The indexing system uses a single string (key) to retrieve the data (value).

- Very fast for direct look-ups
- Schema-less, meaning the value could be anything, such as an object or a pointer to data in another data store

**Column Family Databases.** Column family databases also have rows and columns like a relational database, but storage on disk is organized so that columns of related data are grouped together in the same file. As a result, attributes (columns) can be accessed without having to access all of the other columns in the row.

- Results in very fast actions related to attributes, such as calculating average age
- Performs poorly in regular OLTP applications where the entire row is required

**Document Databases.** Document databases are similar to object databases, but without the need to predefined an object’s attributes (i.e., no schema required).

- Provides flexibility to store new types or unanticipated sizes of data/objects during operation
**Graph Databases.** Graph databases are also similar to object databases, but the objects and relationships between them are all represented as objects with their own respective sets of attributes.

- Enables very fast queries when the value of the data is the relationships between people or items
- Use Graph Databases to identify a relationship between people/items, even when there are many degrees of separation
- Where the relationships represent costs, identify the optimal combination of groups of people/items

The more closely a data model matches the data store structure, the faster queries can be executed and the easier it is to write applications. Of course one size doesn’t fit all needs, and multiple tools may be necessary to fully solve a problem.

**Selecting a Graph Database**

InfiniteGraph, the distributed and scalable graph database from Objectivity, Inc., was designed specifically to traverse complex relationships and provide the framework for a new set of products that deliver real-time business decision support and Big Data connection analytics across copious stores of disparate, unstructured data. Using InfiniteGraph, companies can visualize data relationships and "connect the dots" on a global scale.

This benchmark compares InfiniteGraph and a well-known open source graph database. A high-level view of the two database engines is presented, with a comparison of functionality and performance metrics for some of the most common graph database operations in a distributed environment.

**High-level Overview – Open Source Graph Database Architecture**

The open source graph database is designed to operate in a single-machine/single-disk type of environment. Access to the database from a remote network is achieved using an open source REST web service. Even when accessed locally, concurrent write access to the database is controlled at the database level, preventing two different processes from simultaneously opening the database in write mode. Single server architecture solutions are suitable for environments where access to a distributed or external set of data is not required, but present barriers to scaling for organizations with multiple, distributed data sources and real-world graphs representing complex data relationships.
InfiniteGraph: The Distributed Graph Database

The following figure shows a common use model for the open source graph database.

Several local processes can access the database in read-only mode. In the event multiple applications, or clients, of the database require write operations or when the data is being consumed from a remote machine, the database application has no choice but to use the REST-based web service.

High-level Overview – InfiniteGraph Architecture

InfiniteGraph is built on a highly scalable, distributed database architecture where both data and processing are distributed across the network. A single graph database can be partitioned and distributed across multiple disk volumes and machines, with the ability to query data across machine boundaries. The same database client program can access the graph database locally or across a network in a native manner.

The lock server handles lock requests from database applications, allowing for concurrent read-write access to the graph database. Unlike the open source alternative, database access is not controlled when an instance of the database is created, but rather at the transaction level. The data servers handle remote database application requests for nodes and edges from the distributed graph database. Although the REST interface is useful for interactive access to a database from a browser, it is not suitable for high performance access, ingest (data loading) of very large graphs or navigation across very large graphs. While an InfiniteGraph developer could provide a REST interface if desired, one is not provided with the product as it is designed for high performance and very large graph problems that leverage InfiniteGraph’s unique distributed capabilities.
The following figure shows a common deployment model for the InfiniteGraph database.

The InfiniteGraph deployment includes a dedicated lock server process that manages access to the complete database. There is also a separate data server process on each machine where the database is distributed. These processes provide both local and remote access to the data residing on the disk volumes.

The same database application code can be run locally or remotely. A set of runtime configuration files assists the application code in determining how to interact with the distributed graph database.

**Performance Benchmarks**

The benchmarking tests outlined below are similar to those used by the Graph 500 (http://www.graph500.org/) benchmark for evaluating data intensive applications. The tests involve consuming a set of statistically generated graph data and performing various graph operations on the consumed data. In addition to these tests, the Objectivity team performed common operations like reading elements from the graph.

Following are the benchmark operations:

- Graph construction (Ingesting vertices and building relationships)
- Sampling for 64 search keys based on a unique indexed property (Vertex lookup)
InfiniteGraph: The Distributed Graph Database

- Graph Navigation/Traversal using Breadth-first-search and Depth-first-search
- Reading the graph from the database

The size of the graph is specified in terms of scale and edge-factor (degree). Scale specifies the number of vertices while edge-factor specifies the average number of relationships per vertex.

Given a scale and edge-factor, the number of graph elements (graph size) is determined by:

- Number of vertices = \(2^{\text{scale}}\)
- Number of edges = Number of vertices * edge-factor

Graph Construction (Ingesting Vertices and Building Relationships)

The graph construction operation involves adding a number of automatically indexed vertices with a single property and building the relationships between these vertices. We measure the rate of ingestion for both vertices and edges. We will examine the ingestion rates as a function of the transaction size, number of threads/processes and graph size.

Vertex Ingestion as a Function of Transaction Size

The following figure shows how the database performs when ingesting vertices for a scale of 24 while varying the size of the transaction and the number of concurrent ingest threads. Note that transaction size equals the number of nodes (and edges) per transaction.
There are two key observations that can be made:

1. InfiniteGraph can handle large transactions much more efficiently than the open source graph database.
2. The ingest rate for InfiniteGraph is proportional to the number of threads.

When using 4 concurrent threads and ingesting $10^5$ vertices in a transaction, InfiniteGraph can ingest at a rate greater than 70K vertices per second. Under any condition, the best rate for the open source graph database is less than 20K vertices per second. This is primarily due to InfiniteGraph’s strong support for concurrency and locking management.

The figure below compares the InfiniteGraph ingest rate to the open source graph database when the open source graph database is deployed using the REST web service (note that this is a log-log plot). We can see that InfiniteGraph’s ingest rate is at least two orders of magnitude larger. The transaction size for the REST was controlled using the batch REST API. If the batch REST API is not used, each vertex insert is performed as a single transaction, which would make the open source graph database at least 2 orders of magnitude slower than InfiniteGraph.
Vertex Ingestion as a Function of Graph Size

The following figure shows the relationship between ingestion rate and the graph scale. In this case the maximum transaction size, or number of ingested vertices, was 100,000. When the number of vertices to be added was less than 100,000, the transaction size was simply the total number of vertices to be added. For example for 10,000 vertices, the transaction size was 10,000. When adding 100,000 or more vertices the transaction size was 100,000. The transaction size was kept constant and 4 threads were used to create and index the vertices. For graphs with a very small number of vertices (less than 1000), the open source graph database has a slight advantage. However for graphs with a large number of vertices, InfiniteGraph can be as much as 4x to 5x faster than the open source graph database.

Edge Ingestion (Building Relationships)

InfiniteGraph has two modes of ingesting edges: accelerated and standard ingestion. Accelerated ingestion is a concurrent non-blocking ingest operation where the graph is made to be eventually consistent. The rate of ingestion can be several orders of magnitude faster than standard ingestion due to
to the fact that the operation is multi-tasked and distributed. Accelerated ingestion, in which transactions are ACID compliant, solves a locking issue that is common when updates are performed on the vertex objects. The accelerated ingestion operation resolves these locking issues by creating a set of secondary processes, which efficiently coordinate the relationship building process.

The figure below depicts InfiniteGraph in accelerated ingest mode using the following parameters:

- A single machine
- 4 disk volumes
- 4 agents (not counting the monitoring agents)
- 50MB cache size for the agents

Observations:

- The open source embedded database can only ingest using a single process. When using multiple threads to build the relationship, locking issues are common.
- While the open source graph database performs well when the number of relationships is kept small, the performance degrades significantly when large number of relationships is required.
Using the accelerated ingestion feature, InfiniteGraph allows multiple threads and processes to create relationships.

The ingest rate for InfiniteGraph scales with the number of processing units (threads/processes).

When using the REST API to build a large relationship, the ingest rate is generally 3 orders of magnitude slower than InfiniteGraph.

Sampling for 64 Search Keys Based on a Unique Indexed Property (Vertex Lookup)

Before starting the graph traversal the search tests lookup vertices that were indexed using a unique property key. Automatic indexing is used for both databases; the index provider for the open source graph database is Lucene and built-in Graph indexing is used for InfiniteGraph. The following figure shows the results of this benchmark for various graphs.
Observations:

- InfiniteGraph is slightly faster for lookups compared to the embedded open source graph database.
- The REST-based web service can have a rate that is an order of magnitude slower than InfiniteGraph.

**Reading Graph Objects from the Database**

The reading test iterates the vertex-list in the database and reads the properties of the vertex object and the list of edges for each vertex.

There isn’t an equivalent iterator method for the open source graph database REST service, but fetching the Graph elements using the REST API is in the order of 100s of reads per second.

**Graph Navigation/Traversal Using Breadth-first-search and Depth-first-search**

![Graph Performance Chart](image_url)
The Traversal Test navigates the paths between two given vertices and counts the number of paths that are visited in the process of getting from the first vertex to the second. The search list is the same list as in the query test, which the Graph 500 edge tool generates. It is also worth noting that there is a higher level of control when using the native database interface for navigation versus the REST service API.

The tests involves 64 search paths, which were run when there is a cold read, i.e. with none of the graph elements cached in memory upon initial test start. The following table shows the average rate of traversal for a large graph.

<table>
<thead>
<tr>
<th>InfiniteGraph</th>
<th>Open Source Graph DB (embedded)</th>
<th>Open Source Graph DB (REST service)</th>
</tr>
</thead>
<tbody>
<tr>
<td>68K edges/second</td>
<td>61K edges/second</td>
<td>920 edges/second</td>
</tr>
</tbody>
</table>
Distributed Performance Benchmark of InfiniteGraph

This section of the document presents the benchmark results of the InfiniteGraph database operation in a distributed environment. At the time of this benchmark no other distributed graph database was on the market or available for benchmarking comparisons of distributed performance. The storage of the graph is distributed across several identical machines that are connected over a network. Each machine has several local hard disks connected to it.

In the diagram above, one of the machines contains the system and index databases, while the actual graph objects (vertices/edges) are equally distributed across all the other machines. A single lock server, located on one of the machines, manages access to the complete database. Each machine has an AMS server that serves data stored on a local machine over the network. The ingest agents’ processes are used when accelerated ingesting is performed to create the edges of the graph.

The benchmark operations are:

- Graph construction
- Finding vertices
- Reading vertices
- Graph traversal

Graph Construction

Constructing the graph involves ingesting a number of vertices and then building the relationship (edges) between these vertices. Each vertex has a single property and is automatically indexed using the InfiniteGraph Graph Index.
Ingesting Vertices

An application, which ingests the vertices using multiple threads, runs concurrently on each machine. In total a little over 16 million vertices, equally distributed across all the machines, were ingested. Each vertex has a unique key and is indexed using the Graph Index. The plot below shows the effective rate as a function of the number of machines that were used. The rate is linearly proportional to the number of machines that were used.
Ingesting Edges

Similar to ingesting vertices, an application that ingests the edges using multiple threads runs concurrently on each machine. The edge-ingest application builds the edges given a set of pairs of vertex ids. The graph below shows the rate of ingest as a function of the number of machines. The rate is linearly proportioned to the number of machines used.
Finding Vertices

Searching for vertices, which are uniformly distributed across several machines, involved looking for randomly selected vertices based on unique identifiers and reading the vertex properties. If the properties of the vertex objects were not read, the query rate is a constant regardless of the number of machines used.

Note that the query operation is performed on a single machine while the storage of the graph is distributed.

The following plot shows the query rate as a function of distributing the graph across several machines, when all the properties of the vertices are read. The network bandwidth is the key factor, which determines the query rate. As the number of machines increases the amount of data that has to be transferred over the network also increases. This is the reason for the drop in the query rate.

With InfiniteGraph, clients can run the query operation on each of the machines that are hosting the data, simultaneously and the search rate per machine would be constant, as shown below.
The cumulative effect, using InfiniteGraph in an ideal environment, would be N times faster where N is the number of machines. For example if you have 8 machines and the rate that each machine experiences is 120 query/sec, if the operations is performed on 8 machines simultaneously then the overall rate that you would expect is 8x120 query/sec., an order of magnitude faster than other solutions.

Read Graph Vertex

The reading operation involves iterating the vertex list in the database and reading all the properties of the vertex objects. The plot below shows the rate of this operation as a function of distributing the graph across several machines. Unlike the ingest operations, the rate is not proportional to the number of machines used because similar to the query operations the computation is performed on a single machine while the storage of data is distributed. The network bandwidth is the key factor which determines the rate of this operation.
With InfiniteGraph, the read operation may also be distributed, and depending on the network bandwidth, a more constant read rate is possible, allowing for consistently faster read results.
Graph Traversal

The graph traversal operation traverses the paths between two given vertices and keeps track of the number of paths that were visited in the process of getting from the first vertex to the second. Two sets of tests were performed for this operation. In the first case, the operation was performed when the cache was completely empty (cold traversal) and the rate of traversal degrades as the number of machines used increases. This is primarily due to the increased network traffic. The same traversal is repeated for the second case, however the graph objects are cached in the local memory where the operation was performed, resulting in a traversal rate that is an order of magnitude faster and independent of the number of machines used.

Conclusion

InfiniteGraph and Open Source Graph Database are both NoSQL graph databases that target organizations with large amounts of unstructured data. While similar in some regards, the products have significant differences that have been highlighted in this paper and underscored by a series of benchmark tests.
Open Source Graph Database is a single server solution that works well when dealing with smaller graphs. InfiniteGraph, with a distributed data and process model, has been demonstrated to scale both horizontally and up while maintaining high performance. With support for a true concurrent client connection model, InfiniteGraph is proven to outperform the Open Source Graph Database by orders of magnitude when it comes to ingest and traversal rates.

The InfiniteGraph solution enables organizations to identify and understand complex relationships between distributed data with many degrees of separation. Use cases such as fraud detection, finding the bad guys/surveillance tools, prescription analytics, and network security information and event management (SIEM) which require real-time discovery of connections between “n” degrees of distributed data will require the use of a robust and proven distributed graph database.

The Benchmark findings illustrate some of the reasons why InfiniteGraph is a leading choice in large-scale graph processing and analytics for organizations seeking valuable connections in data and information.

Learn more about InfiniteGraph at www.objectivity.com