InfiniteGraph: Extending Business, Social and Government Intelligence with Graph Analytics

A Technical Whitepaper

Author:
Rick F. van der Lans
Independent Business Intelligence Analyst
R20/Consultancy

September 25, 2010

Sponsored by

Table of Contents

1 Summary

Graph analytics can be applied usefully in a wide range of applications areas, including government, finance, distribution and shipment, commerce, telecommunications, and internet services.

For the business intelligence, social, and government markets a wide range of tools are available with which users can analyze their data stored in data warehouses and production databases. These tools range from straightforward reporting tools via interactive online analytical processing tools to advanced statistical tools. All these tools help users in some way to improve their operations and business decisions. They help by presenting data in a certain graphical way by summarizing data, by grouping data, or by predicting the future.

But there are things these tools can't do and that is analyze data when it's structured as a graph or network and they can't analyze data by traversing graphs. For example, a manager of a social networking website wants to know who the central members of a social network are. Imagine the term central is defined as those members that have the shortest paths to most of the other members. This problem can't be solved by simply summarizing data, nor does it have anything to do with predicting. Instead, the tool must be able to traverse the graph, it has to 'walk' from node to node. And this is not a feature found in most analytical tools available today.

Analyzing network-structures is the domain of *graph analytics*. This is a special form of analytics that has been around for a very long time. In fact, the history of graph analytics and the underlying graph theory goes back to the early 18th century. Today, powerful tools and database servers are available that support graph analytics. What's special about them is that they allow fast online graph traversal even if the graphs consist of millions of nodes.

The challenge of graph analytics on massive graphs is how to store and access the data in such a way that graph traversal is fast. Classic SQL database servers don't offer the right features for graph analytics. Especially, the way in which data is organized as tables and columns, don't make them ideal for graph traversal.

Recently, a new generation of database servers has been introduced, referred to with the intriguing name *NoSQL database servers*. This is not a homogeneous group of products, but a wide range of database servers. The only thing they have in common is that none of them regard SQL as their primary database access language, hence the name. Several don't support SQL at all, a few of them support a form of SQL, and the rest offers SQL as a second language. Some of these NoSQL database servers can be classified as *graph database servers:* products designed and developed specifically to support graph analytics.

InfiniteGraph is a graph database server. It supports many different forms of graph analytics, including single path analysis, shortest path analysis, optimal path analysis, path existence analysis, and vertex centrality analysis. It's developed on top of an object-oriented database server called *Objectivity/DB*. The combination is a robust, proven, and scalable solution

designed to store and analyze massive graphs. It even supports special data partitioning techniques to fully exploit multi-processor hardware technology.

For developers, InfiniteGraph offers a wide range of advantages, including increased productivity, fast processing of massive graphs, efficient data access, optimization techniques for graph traversal. Additionally, it's a database server with low maintenance.

For business users, the advantages of InfiniteGraph are that it allows online analysis of large-scale graphs, quick development of new analytical applications, interactive analysis of graphs, and live analysis of operational data. InfiniteGraph extends the analytical capabilities of the analytical tools used by organizations today.

2 A New Form of Analytics: Graph Analytics

Users of business intelligence architectures can have very different reporting and analytical needs. Therefore, a wide range of tools is available. Here are a few examples of different types of reporting and analytical tools:

- Some users are satisfied with a pre-defined, fixed set of rather straightforward reports in which data is joined, grouped, and summarized. Those reports allow them to view what has happened in the business. Classic *reporting tools* are the preferred tools for these users.

- Large groups of users prefer to do all their reporting and analytics with *spreadsheets*, such as Microsoft's Excel.

- Some users need *OnLine Analytical Processing tools* (OLAP), tools that offer a highly dynamic environment where users can easily perform drill downs and rollups, and where they can view the data in every single way; see Figure 1.

- Another group of users needs to create complex forecasting models, for example, to predict the effect on the sales of houses if the interest rates increase with 0,25%. These are the *statistical tools* for predictions and forecasts.

- And some users prefer to see summarized data being displayed as *key performance indicators* (KPI's) in a dashboard, where data is refreshed periodically, and where the tool should inform them when a specific situation arises; see Figure 2.

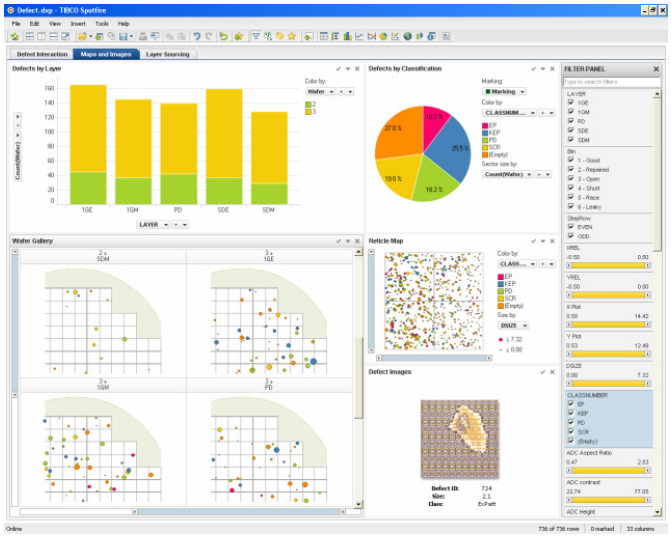

Figure 1 *An example of a report (Spotfire Screenshot Courtesy of Tibco Software Inc.)*

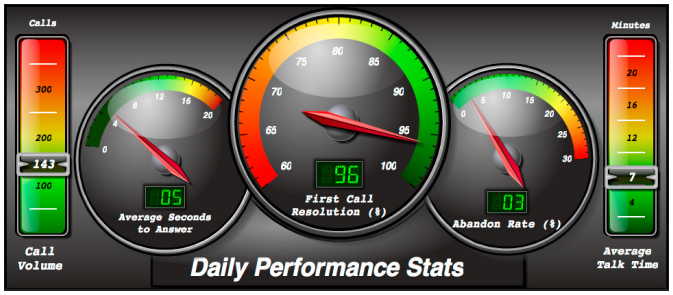

Figure 2 *An example of a dashboard (Courtesy of Corda Technologies, Inc.)*

Graph Analytics – So, for most forms of reporting and analytics tools are available. However, there are forms of analytics that are not supported by these tools. One of these forms is called *graph analytics*. Informally, graph analytics means the study and analysis of data that is formed as a network (graph) of objects and relationships. This form of analytics is characterized not so much by summarizing, grouping, and filtering data, but by traversing or navigating the network. Figure 3 presents a graph of five different people and their relationships. A simple example of a question that can be answered using graph analytics would be: "How many 'hops' are John and Pat removed from each other?" or "Is there a family relationship between John and Pat?"

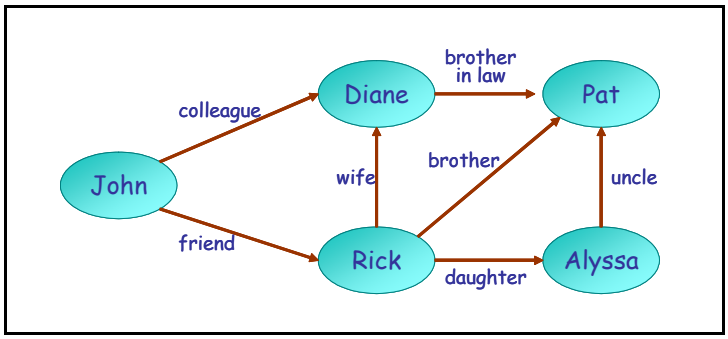

Figure 3 *An example of a simple graph*

Examples of Graphs – Examples of graph-based data structures can be found in almost any industry. They can be found in the world of social networking sites, bio-engineering, drug development, fraud detection, traffic optimization, and many more. A few examples where data can be organized as graphs and where graph analytics might be useful, are:

- All the flights from and to airports can be organized as a graph. In this case, the airports are the objects and the flights the relationships between the objects. Such a graph can be created for all the flights of one airline, or for a set of airlines (such as Expedia's website).
- The network of all the members of a social networking site, such as LinkedIn and Facebook, plus all their relationships can be arranged as a graph.
- The accounts of a bank with all the inter-account money transfers form a graph.
- In the context of a parcel service, all the parcel shipments between addresses world-wide can be organized as a graph.
- A visitor's journey on an organization's website can also be seen as a graph, where webpages form the objects and the visitor's clicks the relationships.
- In the context of a telephone company, all the call detail records between callers can be viewed as relationships between objects, and together they form an incredibly large graph.

Why Graph Analytics? – What's so special about graph analytics? In case of the example with flights between airports, traditional reporting and analytical tools allow us to analyze how many flights depart from a specific airport per day. And if the load factor of flights (percentage of seats sold) is known, we can determine what the average load factor is for flights from Boston to San Francisco. Additionally, if we have the right predictive analytical tools we can even predict what the load factor might be for the coming months. We can also use dashboards that are constantly refreshed and that show the average load factor for flights. But most of the known tools (except for some of the statistical tools) are not able to determine what the two cheapest or four shortest flights with a maximum of two stops in between are from New York to Los Angeles. Nor will they be able to determine for a member of a social networking site, such as LinkedIn and Facebook, which other members someone is likely to know but are not connected to yet, so that they can be encouraged to connect. It will also be hard, in the context of a telephone company, to track down customers that are most likely to influence other customers in their decision to keep or change service providers.

What's special about these latter examples is that they all analyze data that is formed as a graph and that graph is *traversed* (or navigated) by walking from one node to another via their relationships. Sometimes those traversals are guided by the users and sometimes the tools will do the navigation. In short, most existing analytical tools don't provide the mechanisms to navigate and analyze data that is graph-structured. Therefore, a need exists for tools and database servers that do support this form of analytics. These tools should allow for *interactive graph traversal*,

and should be able to traverse massive graphs as found in social network and financial environments.

3 Graph Analytics and Graph Theory in a Nutshell

In this section, a short introduction is given to the foundation and the background of graph theory and graph analytics. In addition, the terminology used in this whitepaper is explained. We begin by explaining the term graph theory.

What is Graph Theory? – Graph theory is the study of graphs in general. It's a set of definitions for concepts such as vertex (node), edge, loop, hyperedge, and anti-triangle. In graph theory a *graph* is a collection of *vertices* (or nodes) and a collection of *edges* that connects pairs of vertices. For example, in Figure 3 the circles represent the vertices and the arrows the edges. Vertices and edges can have *properties*. For example, a vertex in a social network is a member, and its properties might be name, date of birth, and current employer. In that same example, an edge is a relationship between two members. A property of an edge might be the type of relationship between members, such as child-of, friend-of, or employee-of.

A *path* in a graph is a set of vertices connected by edges. It has a starting vertex and an ending vertex (not necessarily different). For example, in a graph that contains family tree data, a path could be formed by a person (John) plus his father (Peter), his father's father (Dyke), and finally his father's father's father (Bob). This path could be presented more formally as follows: John → Peter → Dyke → Bob.

The term *hop* is used when an algorithm traverses a graph and a jump is made from one vertex to another vertex over a specific edge. For example, in the previous example, if an algorithm traverses from John to Bob, three hops are made.

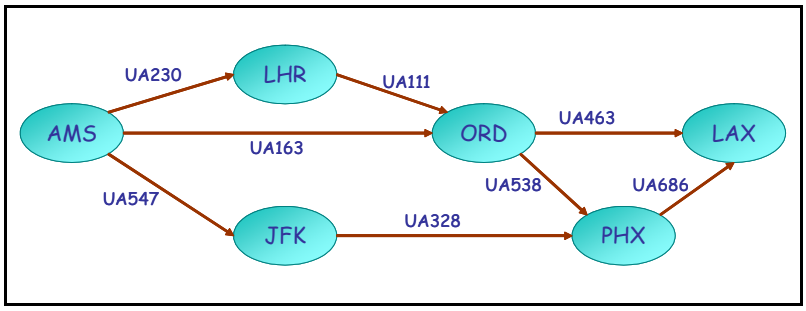

Figure 4 *An example of a directed graph*

A distinction is usually made between *undirected* and *directed graphs*. With the former there is no distinction between the two vertices of an edge. Whereas with the latter there is a distinction: the edge 'points' from one vertex to another. An example of an undirected graph is formed by all the relationships between members of a social networking site. Each member is a vertex and their relationships are the edges. It's an undirected graph because a relationship between John and Mary is the same as a relationship between Mary and John. An example of a directed graph is the route that a user followed to navigate from one wegpage to another. The webpages are the

vertices and the edges are the jumps from one page to another. Another example is formed by all the flights from an airline; see Figure 4. Flights depart from one airport and arrive at another airport. The flights are the edges and the airports the vertices. This is definitely a directed graph, because for a flight a clear distinction exists between the departing and the arrival airport.

What is Graph Analytics? – Graph analytics is the study and analysis of data that is formed as a graph consisting of vertices and edges. An example of graph analytics is to find out in how many ways two members of a social network are linked directly and indirectly. Another example is to determine what the critical path of a project is and to determine how long a project takes minimally. Graph analytics is analogous to web analytics: the analysis of webpage traffic. In this way, it's also comparable to predictive analytics: the analysis of current and historical facts to make predictions about future events.

One of the oldest examples of graph analytics is called seven bridges of Königsberg, which dates back to the first half of the 18th century. This city was set on both sides of the Pregel River, and included two large islands which were connected to each other and the mainland by seven bridges; see Figure 5. The challenge was to find a walk through the city that would cross each bridge once and only once. The islands could not be reached by any route other than the bridges, and every bridge must have been completely crossed every time. Note: It has taken a long time, but in the end it was obvious that there was no solution.

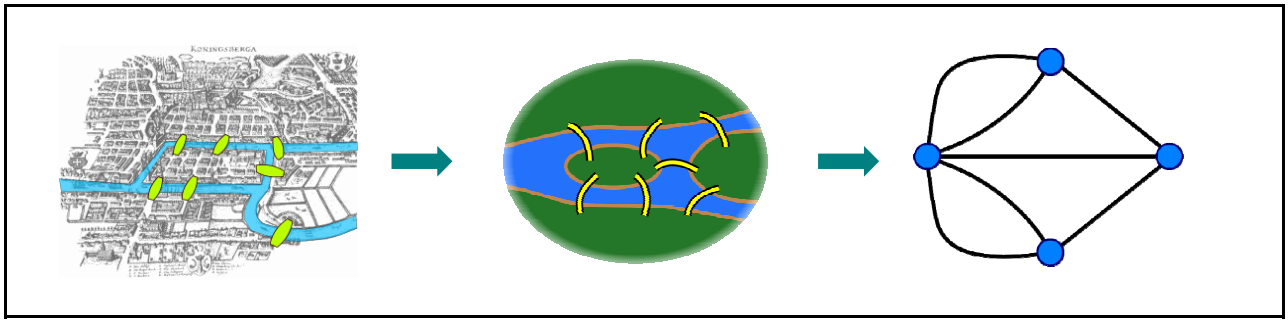

Figure 5 *The seven bridges of Königsberg (Source Wikipedia)*

Nowadays, many examples exist where graph analytics can help organizations to find a solution to their business problems. For example, it can help to find the shortest route for a delivery truck to deliver goods at various stores. Another classic example where it can be applied is when a telephone company wants to analyze call detail records. Besides other details, such a record contains the number of the calling party and the number of the called party. Those parties and all the calls make up a very large graph. Using graph analytics, we might find the answer to the question: "How close together are certain parties?" In other words, determine how many hops are needed to get from one party to another. Another question might be: "How central are certain parties?" The goal of this question might be to find the central parties in the graph.

A more contemporary example of graph analytics relates to social networks. In such an environment, they may have questions such as "Do two members of a social network have more

than five friends in common", "Does a path exists that connects members M_1 and M_2?", and "Of all the members belonging to a social network, can we identify a central member?"

Many other examples can be found. Figure 6 contains a visualization of a graph that represents, among other things, the amount of passes between players during the soccer match between Germany and Sweden at the Worldcup in 2006. Using graph analytics teams can use the results to see how players will behave based on the amount of passes between them.

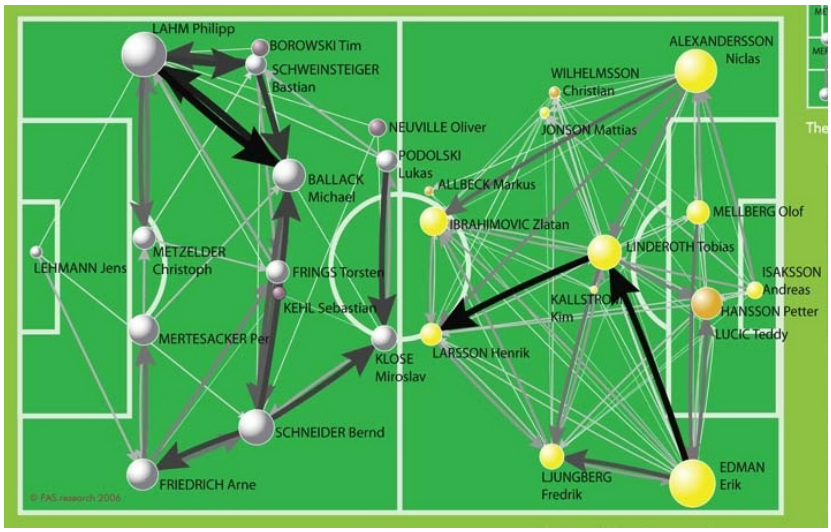

Figure 6 *A graph representing the passes between players during a soccer match*

Most of the problems that can be tackled easily with graph analytics, are quite hard to solve with classic reporting and analysis tools. Most of these tools are good at summarizing data, grouping data, joining data, doing calculations, determining averages, and so on, but they don't really help solve typical graph problems. The main reason is that typical graph problems are solved by *graph traversal;* jumping from one vertex to another via edges. This type of problem is not solved by scanning all the available objects, which is what most other tools are good at.

Note: For more information on graph theory and graph analytics we refer to the works of Chartrand[1], Berge[2], Trudeau[3], and Tutte[4].

4 Different Forms of Graph Analytics

Different forms of graph analytics exist. For example, a graph can be traversed to find an indirect link between two vertices, or the importance of a vertex in a graph can be determined. Here are some popular forms:

[1] Chartrand, G., *'Introductory Graph Theory'*, Dover Publications, 1984.
[2] Berge, C., *'The Theory of Graphs and its Applications'*, John Wiley & Sons, 1964.
[3] Trudeau, R.J., *'Introduction to Graph Theory'*, Dover Publications, second edition, 1994.
[4] Tutte, W.T., *'Graph Theory'*, Cambridge University Press, 2001.

Single Path Analysis – With single path analysis the goal is to find a path through the graph starting with a specific vertex. The path is determined in steps. First, all the edges plus corresponding vertices that can be reached by one hop are evaluated. From the found vertices one is selected, and the first hop is made. Next, the edges and vertices of this found vertex is determined, and this process continues. The result of such an exercise is a path consisting of a number of vertices and edges.

Shortest Path Analysis – When this form of analysis is used, the shortest path is found between two vertices. Shortest means the smallest number of hops. Evidently, the shortest path between two vertices consists of one hop.

Optimal Path Analysis – With this form of analysis the 'best' path is found between two vertices. The best path could be the fastest, the safest, or the cheapest path. The best is based on the properties of the vertices and the edges.

Path Existence Analysis – This form of analysis determines whether paths exist between two vertices. In other words, if we start with two vertices and their edges are followed, will the paths meet somewhere? An example of path existence analysis is the challenge called the *Six Degrees of Kevin Bacon*. If a graph is created in which all the movie stars are the vertices and the edges represent the movies in which they played together, then anyone can be linked to Kevin Bacon within six hops.

Vertex Centrality Analysis – Various measures of the *centrality* of a vertex within a graph have been defined in graph theory. The higher such a measure is, the more 'important' the vertex is in the graph. The following measures have been defined:

- *Degree centrality:* This measure indicates how many edges a vertex has. The more edges, the higher the degree centrality. A vertex with high degree centrality is generally an active vertex or a hub.

- *Closeness centrality:* This measure is the inverse of the sum of all shortest paths to other vertices. In other words, it indicates for a vertex the smallest number of hops to make to reach all other vertices individually. A vertex with high closeness centrality has short paths to many vertices.

- *Betweenness centrality:* This measure indicates the number of shortest paths a vertex is on. It shows a vertex's position within a graph in terms of its ability to make connections to other groups in the graph. Figure 7 shows the degree of betweenness centrality for all the vertices in a graph.

- *Eigenvector centrality:* This measure indicates the importance of a vertex in a graph. Scores are assigned to vertices based on the principle that connections to high-scoring vertices contribute more to the score than equal connections to low-scoring vertices.

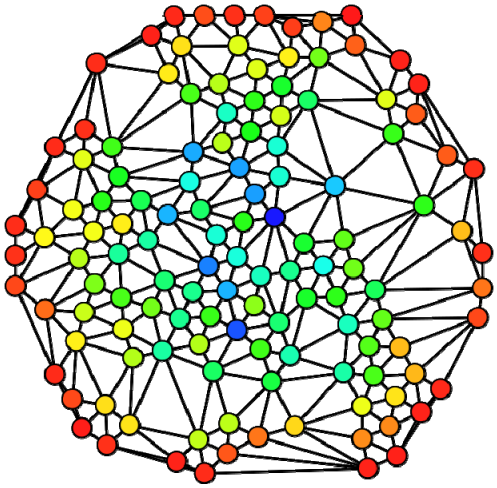

Figure 7 *The colors of the vertices in this graph indicate the measure of betweenness centrality; red is low, blue is high (Source Wikipedia)*

A distinction can be made between *guided* versus *non-guided analysis*. Guided means that during analysis, human analysts make decisions on how to proceed. With non-guided analytics, the tools do all the work, and present a result at the end. For example, single path analysis is probably guided. Every time when a set of vertices is reached by a hop, the analyst determines from which vertex to proceed. Whereas vertex centrality analysis is normally done in a non-guided way. The tool starts its algorithm and presents the results at the end.

5 Application Areas for Graph Analytics

This section describes a few application areas of where graph analytics can be used and where it enriches the existing set of tools we use in business intelligence architectures.

Government – In a government environment a graph can be created linking all private persons, organizations, and their relationships (such as brother, mother, spouse, owner, share holder, member of board of directors, and employee). Vertex centrality analysis can be used to find those vertices that form central roles in the graph: "Which persons are linked to many organizations?" Also, graph analytics could help to find 'hidden' relationships between organizations. For example, path existence analysis could show that some organizations are all linked with two or three hops to a specific person. It could mean nothing, but it could also imply some type of fraud.

Finance – Money transfers between bank accounts could form a very useful graph for all kinds of forms of graph analytics. For example, single path analysis can be used to 'follow' money transfers. Path existence analysis can be applied to determine the different relationships between two account holders. It could be that money flows between two bank accounts via four intermediate bank accounts, but also via two totally different ones. With path existence analysis all those paths are discovered and can be evaluated by specialists. Another example is when the money transfer data is extended with the addresses of the account holders. This might make it possible to see that on a specific address quite a number of accounts is based. It's not that this isn't allowed, but it's an unusual situation.

Distribution and Shipment – Optimal route analysis is an obvious form of graph analytics to be used in distribution and shipment environments. What is the shortest route to deliver goods to various addresses? What is the cheapest flight for a passenger? There are many examples imaginable for these area.

Retail – A graph can be created where products sold form the vertices and the customers buying the products are the edges. By using vertex centrality analysis the dominant products can be identified. Path analysis can also be used to determine the ideal locations for products inside shops.

Telephone Companies – All the call detail records form a huge graph. Applying vertex centrality analysis can give insight into the dominant parties.

Parcel Services – The addresses where parcels must be picked up and delivered can form the vertices of a graph and the parcel deliveries the edges. Graph analytics can give insight in the central address. Especially if this graph is extended with geographical information (where are the addresses located), it can give information about the routes that should be taken. It might also show whether there are popular routes. Additionally, it might show whether warehouses for temporary storage must be moved or whether distribution centers must be relocated.

Websites – Every organization that logs all the traffic on their website can create a graph that shows how individual visitors travel through the website. Different forms of path analysis can be used to simulate this traffic and determine whether visitors are using the correct and the most ideal path. In addition, vertex centrality analysis can be used to determine the central webpages. Graph analytics can also be used for studying the usability of the website. One could answer such questions as: "How many steps does it take to be able to download our product?" or "How can we get the user to an answer in the least amount of steps possible?"

In fact, whether graph analytics can be used, depends greatly on whether analysts can turn their data into a graph. This requires business knowledge and sometimes creativity. Using graph analytics in business intelligence architectures to improve business decisions is a relatively new area, but has an enormous potential. It definitely extends the reporting and analytical capabilities already available.

Graph Analytics and Operational Data – Most of the problems for which graph analytics is suitable, deal with operational management. Normally, strategic and tactical management do not have analytical challenges that can be handled with graph analytics. It's the management layer is closest to the business processes that will use graph analytics. Certain business processes can be optimized or improved using graph analytics. However, this also implies that, in most cases, 100% up-to-date data is needed to perform the analysis. And this could mean that graph analytics is performed on large datasets and, because the analysis is done online, the tools should be fast. Note: In most organizations graph analytics is regarded as a form of *operational analytics*.

6 Storing and Retrieving Graphs

To be able to analyze graphs, at least two components are needed. First of all, a tool that can be used by users to view the analysis results and to guide the analytical process; see for example Figure 8. Secondly, a solution for storing and accessing the graph data is needed. Today, there are three solutions available for storing graphs:

- in files
- in SQL database servers
- in NoSQL database servers

The first solution is discussed in the next section, the second one in Section 8, and the third in Section 9.

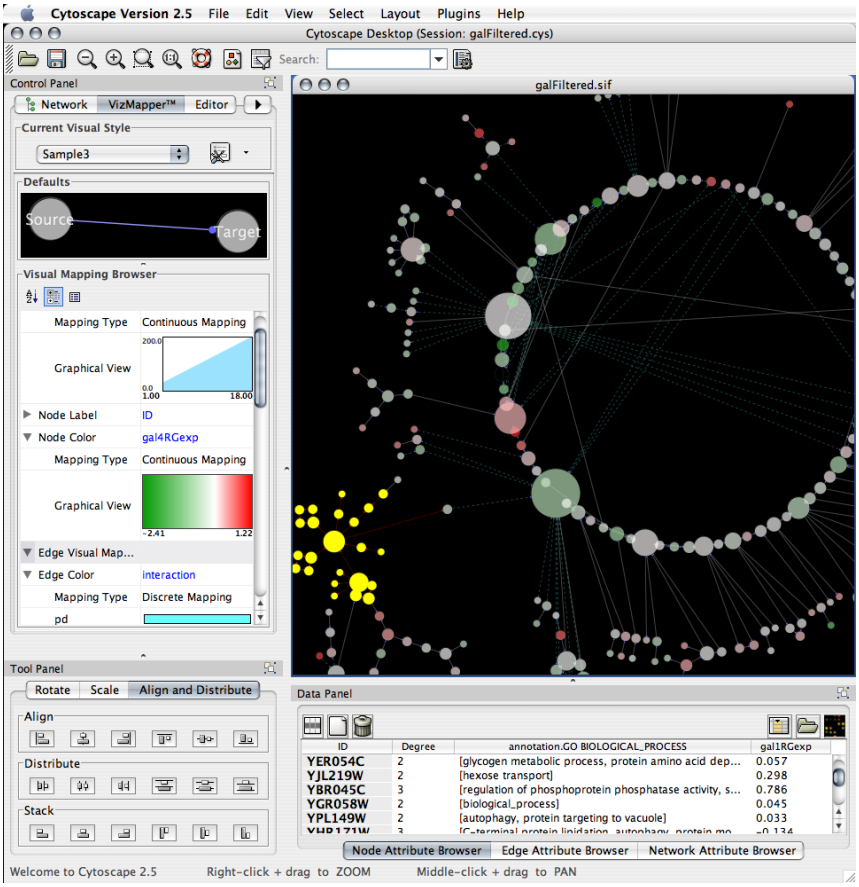

Figure 8 *An example of a tool for viewing the results of graph analytics*

7 Using Files for Storing and Retrieving Graphs

Graph data can be stored simply in files. Many tools on the market for viewing and analyzing graphs work this way. They don't operate on database servers, but instead store the graph data

in a simple file with a GML, GDF, GEXF, GraphML, or DOT format. All the relevant data is loaded in memory before the analysis.

In most cases, these tools run on client machines. The advantage is that response times are very fast. However, a disadvantage is that only a limited amount of data can be loaded and processed. The resources on such a machine are limited, so only graphs of limited size can be analyzed. Some of them are very generic and can deal with any type of data, and some are designed for a specific application area.

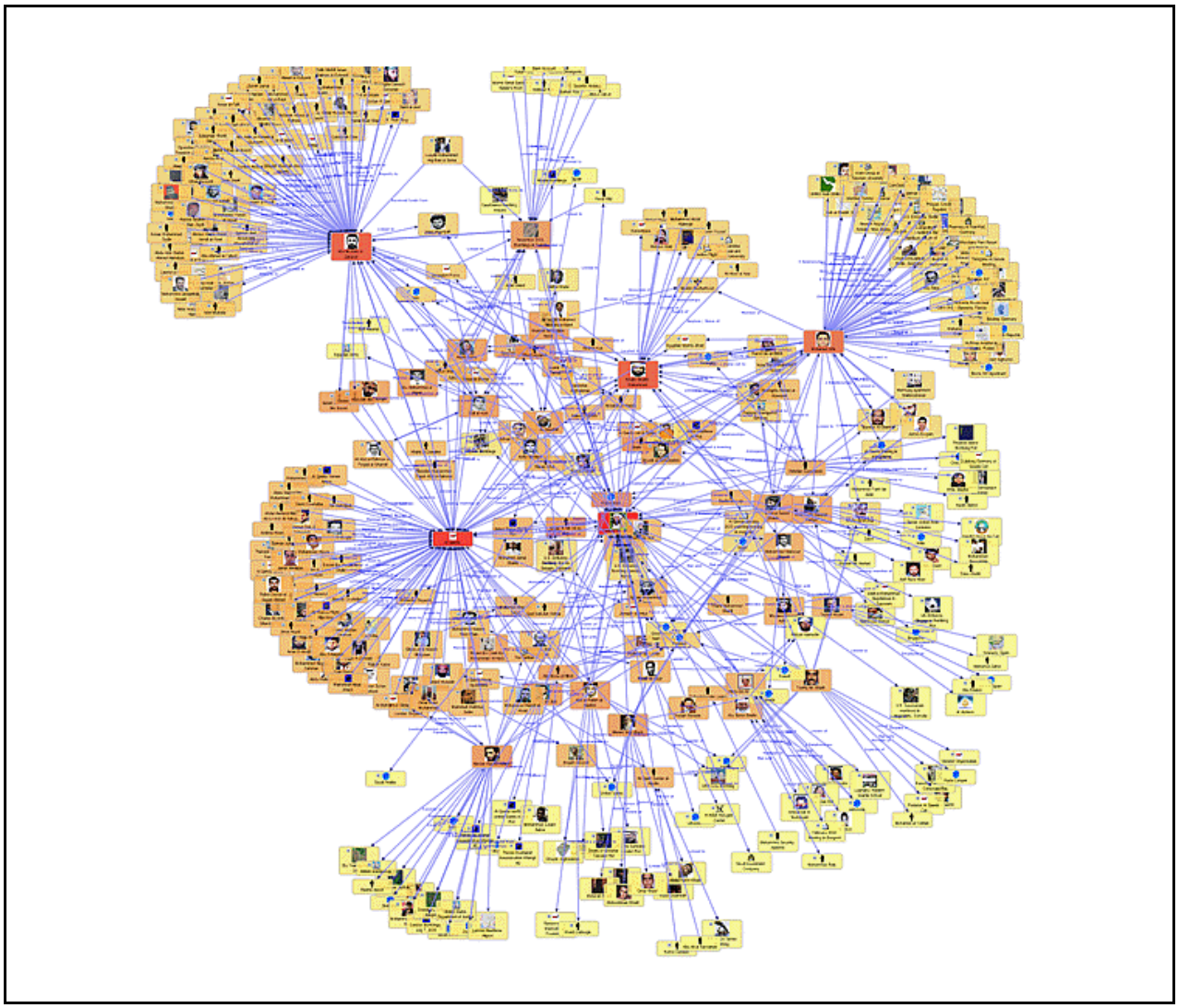

Figure 9 *Example of Sentinel Visualizer*

For example, Cytoscape, used in Figure 8, is a tool for visualizing molecular interaction networks and integrating these interactions with gene expression profiles. Figure 9 contains another example. It shows a result created with Sentinel Visualizer (by FMS Advanced Systems Group); this tool is designed for social network analysis and supports functionality to calculate the degree

centrality, betweenness centrality, and the closeness of members. Figure 10 contains a great example of linking social network data to geographical data.

If large graphs must be manipulated, graph data should be stored in database servers, which is the topic of the next two sections.

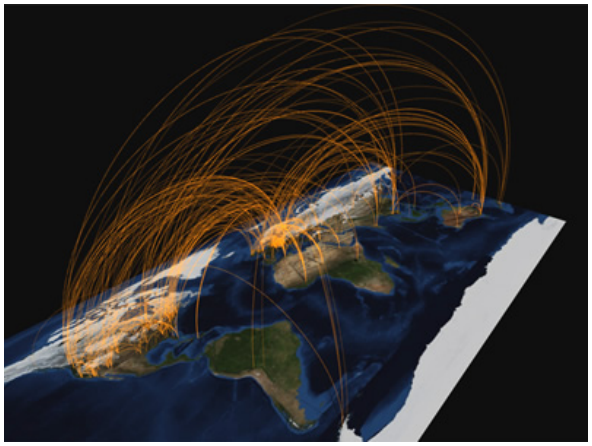

Figure 10 *Graph created by Manuel Lima of VisualComplexity.com*

8 Using SQL Database Servers for Storing and Retrieving Graphs

Querying Graphs with SQL – Graphs can be stored in SQL databases. Only a few tables are needed to store all the data on edges and vertices. Because SQL database servers have proven they can manage databases Terabytes large, they won't have a problem with storing massive graphs.

When data is stored in tables, SQL can be used to query the data. Simple questions such as "How many edges are connected to a specific vertex" are easy to handle and to answer. However, many of the queries that are typical for graph analytics are recursive by nature and that might be a problem with SQL database servers. Let's illustrate this with an example.

Imagine we have data related to flights of one airline leaving from and arriving at airports. In this example the airports form the vertices and the flights form the edges of the graph. This data is stored in the following FLIGHTS table:

```
FNO   DAIR AAIR DEPARTURE_TIME    ARRIVAL_TIME      PRICE
----- ---- ---- ----------------- ----------------- --------
0     AMS  LHR  2007-03-01-11.30  2007-03-01-12.30  160.17
1     LHR  ORD  2007-03-01-13.30  2007-03-01-19.30  964.29
2     ORD  LAX  2007-03-01-20.30  2007-03-02-01.30  583.11
3     LAX  SYD  2007-03-02-02.30  2007-03-02-12.30  1663.04
4     AMS  TYO  2007-03-01-11.00  2007-03-01-22.00  1595.86
5     TYO  SYD  2007-03-02-03.00  2007-03-02-14.00  1487.33
6     AMS  LAX  2007-03-01-18.00  2007-03-02-07.00  1374.15
7     AMS  JFK  2007-03-01-10.00  2007-03-01-16.00  964.61
8     JFK  PHX  2007-03-01-19.00  2007-03-02-01.00  1069.99
9     AMS  LGA  2007-03-01-10.00  2007-03-01-16.00  1081.56
10    LGA  PHX  2007-03-01-20.00  2007-03-02-02.00  911.92
11    AMS  EWR  2007-03-01-10.00  2007-03-01-17.00  911.36
12    EWR  PHX  2007-03-01-19.00  2007-03-02-00.00  937.98
13    AMS  CAI  2007-03-01-09.00  2007-03-01-16.00  1208.67
14    CAI  TYO  2007-03-01-19.00  2007-03-02-00.00  977.95
15    AMS  JFK  2007-03-01-15.00  2007-03-01-21.00  1155.43
16    AMS  LGA  2007-03-01-12.00  2007-03-01-18.00  923.61
17    AMS  LHR  2007-03-01-15.00  2007-03-01-16.00  114.23
```

Next, we have the following query to solve: “Get the cheapest flights from Amsterdam to Phoenix leaving on March 1, 2007, with a maximum of two stops, and each stop should be less than 4 hours”. Using SQL, the query would look as follows:

```
WITH     FLIGHTPLAN(FLIGHTNO, PLAN_AIRPORTS, PLAN_FLIGHTS,
                    START_AIRPORT, END_AIRPORT, START_TIME, END_TIME,
                    DEPARTURE_AIRPORT, ARRIVAL_AIRPORT,
                    DEPARTURE_TIME, ARRIVAL_TIME, PRICE, STOPS) AS
        (SELECT   FLIGHTNO, CAST(DEPARTURE_AIRPORT || '->' ||
                  ARRIVAL_AIRPORT AS VARCHAR(100)),
                  CAST(RTRIM(CHAR(FLIGHTNO)) AS VARCHAR(100)),
                  DEPARTURE_AIRPORT, ARRIVAL_AIRPORT,
                  DEPARTURE_TIME, ARRIVAL_TIME,
                  DEPARTURE_AIRPORT, ARRIVAL_AIRPORT,
                  DEPARTURE_TIME, ARRIVAL_TIME, PRICE, 0
         FROM     FLIGHTS
         WHERE    DEPARTURE_AIRPORT='AMS'
         AND      CAST(DEPARTURE_TIME AS DATE) = '2007-03-01'
         UNION ALL
         SELECT   P.FLIGHTNO, P.PLAN_AIRPORTS || '->' || F.ARRIVAL_AIRPORT,
                  P.PLAN_FLIGHTS || '->' || RTRIM(CHAR(F.FLIGHTNO)),
                  P.START_AIRPORT, F.ARRIVAL_AIRPORT,
                  P.START_TIME, F.ARRIVAL_TIME,
                  P.DEPARTURE_AIRPORT, P.ARRIVAL_AIRPORT,
                  P.DEPARTURE_TIME, P.ARRIVAL_TIME,
                  P.PRICE + F.PRICE, STOPS+1
         FROM     FLIGHTPLAN AS P, FLIGHTS AS F
         WHERE    P.ARRIVAL_AIRPORT = F.DEPARTURE_AIRPORT
         AND      P.ARRIVAL_TIME < F.DEPARTURE_TIME
         AND      F.DEPARTURE_AIRPORT <> 'PHX'
         AND      LOCATE(F.ARRIVAL_AIRPORT, P.PLAN_AIRPORTS) = 0
         AND      STOPS < 1
         AND      P.ARRIVAL_TIME + 4 HOURS > F.DEPARTURE_TIME)
SELECT   PLAN_AIRPORTS, PLAN_FLIGHTS, START_AIRPORT, END_AIRPORT,
         START_TIME, END_TIME, PRICE
FROM     FLIGHTPLAN
WHERE    END_AIRPORT = 'PHX'
ORDER BY PRICE ASC
FETCH FIRST 1 ROW ONLY
```

Although this query returns the right result, it’s not an easy query to write. In addition, it will be hard for the database server to process this query quickly. So especially when the graphs become

large, these queries might take a long time and probably consume a lot of resources. It's also a query that's hard to optimize and to parallelize.

Note: In the previous paragraphs the assumption was made that SQL database servers support recursive queries, but that's not the case. In fact, only a few do. For the other database servers, code must be added to the applications to traverse the graphs. In other words, the database server won't be doing the analysis, but the applications.

Storing Graphs in SQL Databases – Another problem is related to how data is stored in SQL database servers. In most of them, all data is stored as records belonging to tables. This means that the vertices making up a graph are also stored as records. So far, so good. Records can be retrieved by scanning tables or by using indices.

If tables contain a large set of rows, and all these rows must be scanned frequently, tables can be partitioned. These partitions are then distributed across multiple processors. Each processor controls one or more partitions. The advantage is that partitions can be scanned in parallel (instead of scanning all the rows serially) by the processors. This definitely improves the overall query performance.

In order to distribute query processing across multiple processors, the architecture of a database server is distributed. Figure 11 shows the typical architecture for such a parallel database server. The database server has processing modules, frequently called *nodes*. One of those nodes is called the *Master* and the other nodes are *Workers*. Each Worker manages a number of tables or table partitions. Usually, the Master knows where all the data is stored. The Master and the Workers can run on different processors in one single machine, or they can be distributed across a network or cluster of machines.

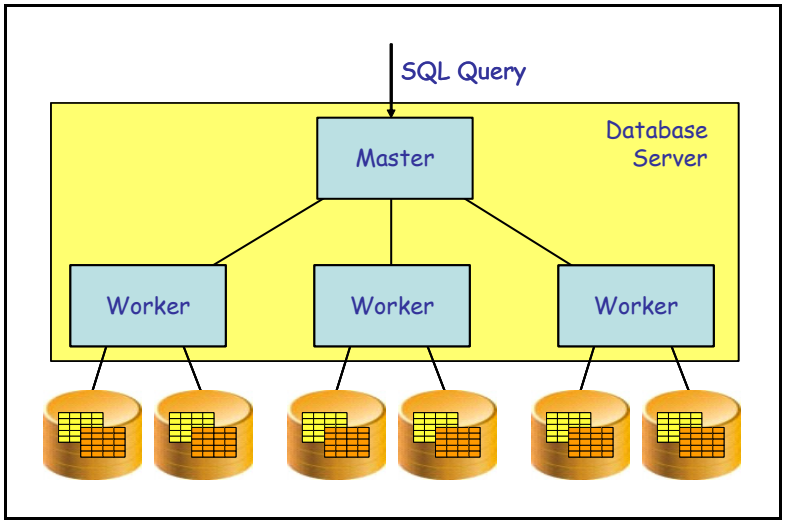

Figure 11 *Typical architecture for parallel database servers*

This architecture is very useful if it makes sense to scan each partition completely. However, this is not the case when graphs are traversed. For example, take the graph in Figure 12, and let's say we want to find out whether there is a link between members M_1 and M_9. If we would store all the relationships between members in a table, and if we would partition that table, the

members that make up the graph would be spread out over all the partitions. For every hop to the next set of vertices, the table (read the partitions) must probably be scanned. So we might end up with scanning the partitions four times. In addition, we're probably not accessing large set of rows, but we are scanning most of them. In fact, traversing graphs in this situation could lead to bad query performance, because of the somewhat brute-force scanning solution.

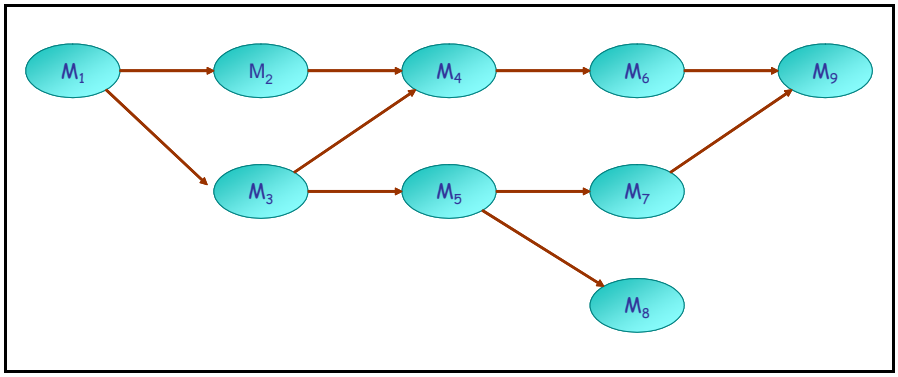

Figure 12 *A graph containing members of a social network*

The last few years, a new generation of SQL database servers has been introduced. These are database servers that have been designed and optimized for reporting and analytics. It includes *data warehouse appliances*, *columnar databases*, and *analytical database servers*. Their internal architectures make them very efficient for running straightforward reporting and analytical queries up to running complex statistical forecasting models. However, the queries generated by graph analytics are not typical queries. They have a very different pattern. The consequence is that even this new generation has a hard time in improving the performance of graph-oriented queries.

Maybe the products will still have a decent performance if the graphs contain a limited number of vertices and edges. But nowadays graphs might consist of millions, sometimes even billions of vertices, such as the massive graph created by all the LinkedIn members. In May 2010, LinkedIn had 70 million members, and every second a new member is added.

Traversing Graphs in SQL Databases – Doing graph analytics always implies traversing the graphs. Let's first explain what graph traversal could mean before we indicate whether a SQL database server is ideal for this.

In most cases, if a graph has to be traversed there is a starting point: the vertex where the traversal starts. It might be a member of a social network, or the departure airport of a multi-stop flight. Depending on what the original query is, we talk about directed and undirected traversals.

With *directed traversal*, just a small portion of a graph is accessed. It's almost as if the query knows where to go. For example, imagine a graph containing money transfers where the vertices represent the transfers and the edges the bank accounts. Imagine the objective is to find where a certain money amount went to. You must start with a specific vertex and follow the edge that represents the relevant money transfer. By making this hop, another account is found. Next, all the money transfers originating from this second account are studied and you follow

the edge that has the same money amount (which is probably only one hop). You must continue this process, and you will create a path from the account where the money transfer originated to the one where it stops. If this algorithm is programmed, the number of edges to use and the number of members to access is very limited. Not that many resources are needed for a directed traversal.

A more complex and more resource intensive query is an *undirected traversal*. With such a traversal, many (maybe all) edges of a graph must be traversed and all the edges evaluated. Imagine the following query: In a social network, in what ways is member M_1 linked (indirectly or directly) to M_2? In principle, these two members could be linked via many different paths, and some of those paths may be quite long. The only way to find an answer to this query is to traverse a large portion of the graph. It starts with finding all the members to whom M_1 is linked directly. Let's assume that each member is linked to on average 25 other members. In that case, approximately 25 members will be found. If M_2 is on that list, one path found is $M_1 \rightarrow M_2$. Next, for all the other 25 members an extra hop must be executed. Note that this could return 25^2 (625) members. Again, if M_2 is found, new paths are added to the result. For example, it might be that the path $M_1 \rightarrow M_{15} \rightarrow M_2$ is added. This recursive process will continue. Everyone can do the math: with hop number six over 244 million (25^6) members might be found. And with ten hops we're in the trillions of members. Because the graph is traversed by fanning out through the graph, there will be an enormous amount of internal administration to keep track of.

And where does the algorithm stop? We can't say stop after ten hops, because there might a relationship over fifteen members. The algorithm should stop when after a certain hop no members are found that haven't already been evaluated.

To optimize this process, you must keep track of which members have already been evaluated. The reason is that it doesn't make sense to evaluate a member twice. Plus, it doesn't make sense to traverse away from the same member again. In other words, if M_{12} has already been evaluated, it shouldn't be evaluated again.

If a SQL database server supports recursive queries, undirected traversal of graphs will be computationally expensive, and will require large amounts of memory and CPU processing. Most developers will opt for moving the traversal process to the applications. On the other hand, if a SQL database server doesn't support recursive queries, the applications must do the graph traversal themselves anyway. So in both cases it will be the task of the applications to do all the graph traversal and to remember all the vertices and edges it has covered already.

Conclusion, SQL database servers have no problem with storing graph data, not even when the graphs consist of millions of vertices, but using them for graph traversal is not an ideal solution.

9 Using NoSQL Database Servers for Storing and Retrieving Graphs

Besides the SQL database servers, there are numerous other database servers that don't support SQL or a limited version of SQL. Nowadays, these products together are called *NoSQL database*

servers. The question we would like to answer in this section is whether NoSQL products exist that can easily support graph analytics, storage-wise and query-wise. But we begin by explaining what NoSQL means and by describing the background of why these products were introduced.

The Origin of NoSQL Database Servers – As the name suggests, all NoSQL products have in common that they don't regard SQL as their primary database access language. Several don't support SQL at all, a few support a form of SQL, and the rest offer SQL as a second language. In addition, there are NoSQL database servers that don't present data as tables, columns, and records. The name NoSQL might bring on that suggestion, but it's not a homogeneous group of products and there are no standards for NoSQL database servers.

They all have their own language, their own API, and their own storage structures; in a certain respect they are proprietary products. Whereas SQL database servers are suitable for various kinds of application areas, most NoSQL products are designed for just one or two application areas, such as document management and analytical processing. The term NOSQL (Not Only SQL) was first used in 1998 for a database server that had no support for SQL. Early 2009 the term was re-introduced by Eric Evans for an event that was organized for database servers that had no SQL support. And the term stuck. So, it's now being used to refer to all the non-SQL-based database servers.

Looking back at history, there have always been database servers not supporting SQL. In fact, there were database servers on the market even before SQL existed. For example, one of IBM's first database servers called IMS (Information Management Services) did not support SQL, but a language called DL/I and data was not stored in tables but hierarchies. IMS was released in 1968 (and it is still being supported), whereas the first research articles on SQL were not published before the end of the 1970s. And we had to wait for the first products until the end of the 1970s. The first standard of SQL was published mid 1980s. Other examples of database servers that didn't support SQL are IDMS, which was released in the 1960s by Cullinane Database Systems, and Cincom Systems started to sell Total since 1968.

There are other examples of NoSQL database servers. For example, non-SQL-based database servers for doing reporting and analytics have existed for quite some time, such as Express (introduced in 1970), Essbase (introduced in 1992), and Microsoft Analysis Services (1997); the first two are now owned by Oracle. These so-called *multidimensional* or *OLAP database servers* did not support SQL either, but had their own languages. Microsoft's product supported a language called MDX and is now seen as a de-facto standard. For more information on these products, we refer to this document written by Nigel Pendse.

Besides the pre-SQL and the multidimensional databases, there are other groups of NoSQL database servers, such as XML and object-oriented database servers.

More recently, various new NoSQL database servers were introduced. There are products specifically designed for storing and searching documents, sometimes called *document stores* or *document-oriented database servers*. Examples are Apache's CouchDB and Jackrabbit; MongoDB; and RavenDB. Another popular group of NoSQL database servers is called the *key/value stores*, such as Google's BigTable, Amazon's SimpleDB, and CDB. In most SQL

database servers data is stored in rows or in columns. In key/value stores the data is stored as individual values.

Graph Database Servers – Finally, one of those groups of NoSQL database servers are the *graph database servers*, with their specific storage formats and specific API's, all aimed at doing graph analytics fast on large amounts of data. More on these products at the end of this section.

To summarize, there have always been and there will always be database servers that are not based on the SQL language and that do not organize data in tables and columns, because there will always be requirements for which SQL database servers are not sufficient. But it was only recently that they have been classified as NoSQL. The main reason was the introduction of a number of those products by popular vendors, such as Apache, Amazon, and Google.

The Performance Gap – Besides the products introduced before SQL existed, the dominant reason these NoSQL products have been developed and have received quite some attention is that there are organizations for which SQL database servers didn't offer the right performance, functionality, or both. For example, if the amount of data to be queried is so massive, it could be that even the fastest SQL product can't guarantee the right performance. Other reasons could be that the applications contain very special types of queries that are hard to formulate with SQL, hard to optimize and have, therefore, a poor performance. It could also be that the type of data to be stored doesn't work well with SQL database servers.

To summarize, NoSQL database servers were introduced because the performance offered by SQL database servers was not what organizations required for certain application areas. This difference is shown in Figure 13. The blue-ish area indicates the performance offered by SQL database servers. Through time, the offered performance increases, in other words, with every new version or edition the products become more powerful. This might be due to new query optimization techniques, new index types, improved storage formats, and even new hardware technologies. The three red arrows indicate what different organizations need with respect to performance. Due to an increase of users, queries, or data, their requirements for performance will increase. Organizations with minimal performance needs will experience that, over time, there will be an excess of performance. For organizations with average performance needs, the vendors are probably improving their products with the same speed as the performance needs of organization's increase. For both groups of organizations, SQL database servers offer sufficient performance. However, there is always a group of organizations for which the current SQL database servers do not offer enough performance. The difference between what they need and what's offered, is called the *performance gap*. This gap could be due to the enormous amount of data they must store and query, it could also be due to the query workload, or due to the type of queries they have, they might be so complex, so hard to optimize, or not at all suitable for SQL database servers. All this could lead to performance problems. Those are the organizations that experience a performance gap and that investigate what NoSQL database servers have to offer.

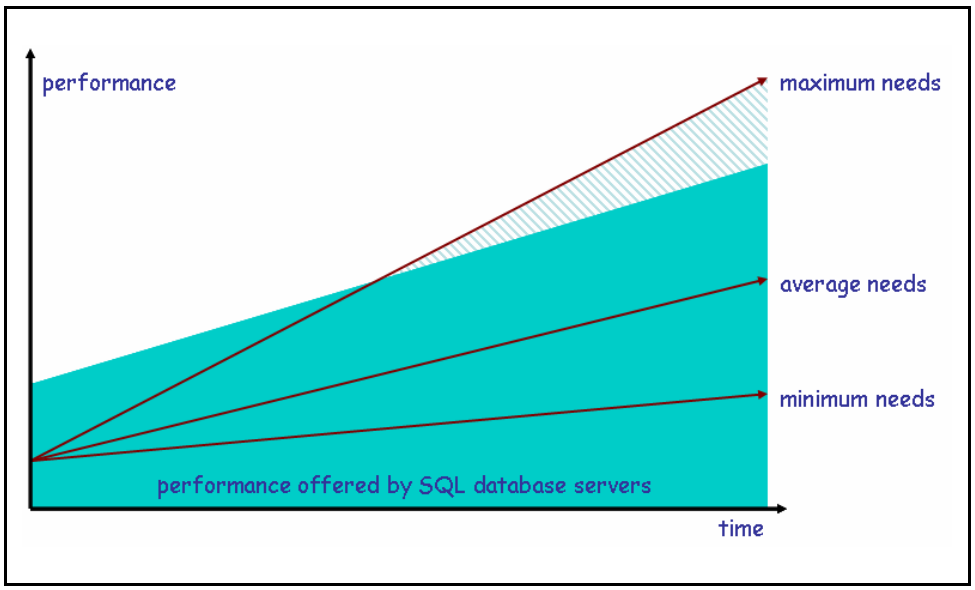

Figure 13 *Performance offered by database servers versus performance required by organizations*

As indicated in Section 6, graph analytics is also a clear example of an application area that is hard to manage for SQL database servers. Graph database servers, such as InfiniteGraph, are designed specifically for this application area. Internally, everything in a graph database server is designed for storing, searching, analyzing, and managing graphs. Its storage format fits navigation required for graph analytics, it has an API specifically designed for this purpose, and it can traverse a graph consisting of many vertices and edges very quickly.

To answer the question raised at the start of this section: not all groups of NoSQL database servers are ideal for graph analytics, but the graph database servers certainly are. They can be used in those areas where graph analytics is needed, but where other database servers can't offer the right performance level; the light-blue area in Figure 13. Graph database servers close the performance gap for graph-type applications.

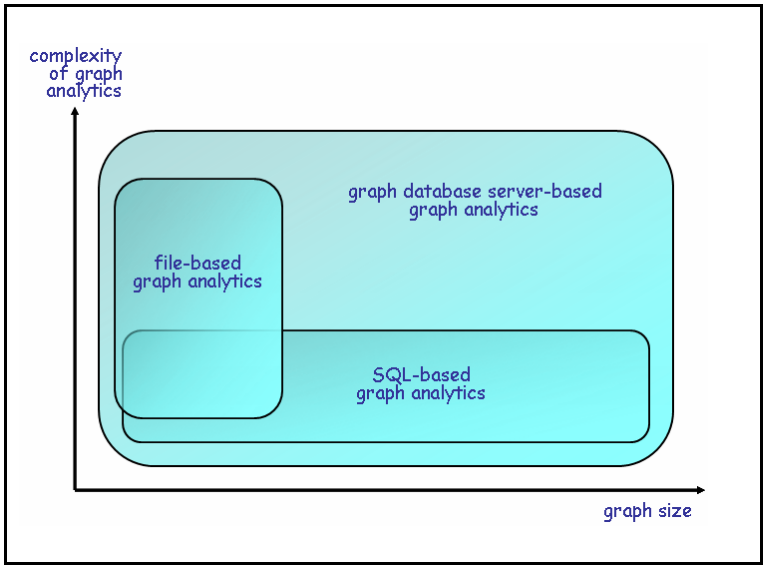

Figure 14 *Positioning of graph analytics using graph database server*

To summarize, Figure 14 compares graph analytics supported by files, by SQL database servers, and by graph database servers. When the complexity of graph analytics is high and when the graphs are very large, graph database servers offer the right solution.

10 What is a Graph Database Server?

A graph database server is designed specifically to support graph analytics through fast and efficient graph traversal. Its storage model is aimed at storing database objects, such as vertices, edges, and their respective properties, in such a way that it makes graph traversal very efficient. In addition it should offer a language or API with which applications can insert, modify, and delete these database objects and with which the graphs can be traversed.

Inserting the Graph – Most developers are familiar with inserting, updating, deleting, and querying data with a SQL database server. In this section we use a few simple examples to show what comparable operations look like in a graph database server. The examples show how vertices are introduced and how those vertices are linked with an edge. The context of these examples is a social network environment. The language used here is the one supported by InfiniteGraph.

First we create four vertices. On the first line, one member is created called Peter Johnson. On the second line, this member is added to the database called myGraphDB. On the next six lines three more members are created.

```
Member member1 = new Member("Peter Johnson");
myGraphDB.addVertex(member1);
Member member2 = new Member("Mary Metheny");
myGraphDB.addVertex(member2);
Member member3 = new Member("Mark Johnson");
myGraphDB.addVertex(member3);
Member member4 = new Member("Diane Cools");
myGraphDB.addVertex(member4);
```

Next, we create three types of edges. They represent three different types of relationships (colleague, friend, and family) between vertices.

```
RelatedTo relation1 = new RelatedTo("colleague");
RelatedTo relation2 = new RelatedTo("friend");
RelatedTo relation3 = new RelatedTo("family");
```

Finally, we link the members using edges. On the first line member2 (Mary Metheny) is linked to member1 (Peter Johnson), and the type of relationship is relation1 (colleague). On the second line, Peter Johnson and Mark Johnson are linked as family, and on the last Mary Metheny and Diane Cools are linked as friends.

```
member1.addEdge(relation1, member2, EdgeKind.BIDIRECTIONAL);
member1.addEdge(relation3, member3, EdgeKind.BIDIRECTIONAL);
member2.addEdge(relation2, member4, EdgeKind.BIDIRECTIONAL);
```

The specification `EdgeKind.BIDIRECTIONAL` indicates that the edge is bidirectional, or in other words, an undirected graph is created.

Querying the Graph – The API of InfiniteGraph offers very extensive query capabilities. Classic questions can be formulated, such as "Get the name and birth data of a member" and "What's the sum of all the sales in the Boston region". And of course there are constructs to traverse graphs. With a few examples we will show what queries look like.

Here is an example of how to retrieve one specific member from the database (one vertex):

```
myGraphDB.nameVertex("member1", member1);
Person retrievedmember1 = (Member)myGraphDB.getNamedVertex("member1");
```

The result is that member Peter Johnson is assigned to the object called retrievedmember1.

But more importantly, how can we traverse a graph? Here is an example of how to find all the colleagues of a specific member:

```
private static void findAllColleaguesOfMember(Member m)
  {
     Navigator query = m.navigate(Guide.SIMPLE_BREADTH_FIRST, Qualifier.ANY,
           new Qualifier() {
              @Override
              public boolean qualify(Path currentPath)
              {
                 Hop h = currentPath.getFinalHop();
                 RelatedTo relatedTo = (RelatedTo)h.getEdge();
                 if (currentPath.size() == 2 && relatedTo.getType().equals("colleague"))
                    return true;
                 return false;
              }
           },
           new NavigationResultHandler()
           {
              public void handleResultPath(Path result, Navigator navigator) {
                 System.out.println("Found the following colleagues : ");
                 for (Hop h : result)
                 {
                    if (h.hasEdge())
                    {
                       System.out.println(h.getVertex().toString());
                    }
                 }
              }
           });
     query.start();
  }
```

With the next example, we locate whether there is a direct or indirect relation between Peter Johnson and Diane Cools:

```
private static class RelationTypeResultsHandler implements NavigationResultHandler  {
    public enum RelationType   {
       NONE,  DIRECT,  INDIRECT;
    }

    private RelationType rt = RelationType.NONE;
    private long targetMemberId;

    public RelationTypeResultsHandler(long memberId)    {
       this.targetMemberId = memberId;
    }

    public RelationType getRelationType()   {
       return rt;
    }

    @Override
    public void handleResultPath(Path result, Navigator navigator)   {
       Hop h = result.getFinalHop();
       if ((h.getVertexHandle().getId() == targetMemberId))    {
          // Found the target member.  Now check the relationship to the start member
          if (result.size() == 2)   {
             // 2 hops = Vertex <- Edge -> Vertex or direct connection
             rt = RelationType.DIRECT;
          }  else   {
             rt = RelationType.INDIRECT;
          }
          navigator.stop();
       }
    }
 };
}
```

In Section 6 we showed what a recursive query looks like in SQL. The query was: "Get the cheapest flights from Amsterdam to Phoenix leaving on March 1, 2007, with a maximum of stops, and each stop should be less than 4 hours". A comparable query in the InfiniteGraph API would look like this:

```
Vertex departAirport = graphDB.getNamedVertex(departAirportCode);
Vertex arrivAirport = graphDB.getNamedVertex(arrivAirportCode);

PathResultsHandler resultsHandler = new PathResultsHandler();

Navigator query = departAirport.navigate(new CustomGuide(departDate),
   Qualifier.ANY,
   new ResultQualifier(arrivAirport.getId(), maxStops, maxStopTime),
   resultsHandler);

query.start();
```

This solution is much more simple and relatively straightforward. So an additional advantage of this API is productivity and maintenance, because the code is shorter and simpler.

Besides these graph-specific features, a graph database server should support all the classical features of any database server, such as data recovery, data concurrency, and data integrity, and ACID-based transactions. These features should lead to high levels of concurrency.

11 InfiniteGraph

InfiniteGraph is a graph database server. In June 2010 the first public beta version was released. A month later, the first 1.0 version was released. InfiniteGraph was developed by Objectivity Inc. based in Sunnyvale, California. Currently, Windows and Linux are supported. For both platforms, 32-bits and 64-bit versions exist. In the near future, support for the Mac is expected. The database server is accessible from Java.

There were two dominant reasons for Objectivity to develop InfiniteGraph. The first reason was that, over the years, Objectivity/DB had been successfully used by various customers for supporting several large, distributed, custom-made, graph-based applications. The ability of Objectivity/DB to handle complex many-to-many data issues made it a natural solution for these customers. Then the market for graph analytics emerged with the need for a dedicated graph database server. The second reason was the emergence of the NoSQL movement and cloud computing. The emergence of NoSQL to support large, distributed cloud computing environments began to gain significant traction in the database market. Objectivity saw an opportunity to build a commercial graph API on top of a highly scalable and distributed persistence layer that would support cloud-scale graph applications.

Objectivity/DB – Objectivity/DB is a distributed *object-oriented database server* with a long history. In fact, Objectivity was one of the first vendors supplying an object-oriented database server. The first version of this database server was released twenty years ago in 1990, two years after founding the company. The current version is 10. Objectivity/DB has always been a highly scalable object-oriented database server with a very robust transaction mechanism. The distributed storage structure allows for storing and manipulating massive amounts of data.

Objectivity/DB supports different query languages, including EJBQL, LINQ, and SQL++. Especially the latter is important for analytical applications. The database server is used by many organizations for a wide range of applications, including military command and control systems; manufacturing, automation, and process control systems; medical equipment; financial services; and telecommunications systems. The product is also embedded in various software tools where it operates in the background as a built-in database server.

It's a native object-oriented database server, which means that data is not arranged as tables, columns, and rows. Data is stored as objects (with properties) and as relationships between objects. Maybe this data model is not so fast for scanning all the objects from a class, but for picking one object, or, more importantly, for navigating from one object to another it's very fast. In a SQL database server navigating from one row in a table to a row in another table always involves a join. Executing a join is a more complex process than what Objectivity/DB has to do: follow the pointer from one object to another.

This object-oriented data model is ideal for InfiniteGraph. Mapping vertices and edges to objects and relationships is straightforward, it's almost a one-to-one transformation. This also means that when applications ask InfiniteGraph to traverse a graph, technically it's just a matter

of navigating from one object to another. And as indicated, this is something for which Objectivity/DB has been specifically designed.

As indicated, Objectivity/DB is a distributed database server. It can distribute its data and processing. The data can be distributed over 2^{32} database containers. Each container may reside physically on any host within the distributed system. So, it's possible to distribute data among a very large number of hosts. Objectivity/DB will always present all these containers as one logical databases to the clients. The big advantage of being able to handle massive amounts of data is that InfiniteGraph can manage and analyze graphs consisting of billions of vertices and edges.

Objectivity/DB also supports distributed processing. If task can be split between two or more clients, then each of those will share the work of the database server in some respects. All the distribution of data and processing is done through a peer-to-peer architecture.

InfiniteGraph – As said, InfiniteGraph is developed as a layer on top of Objectivity/DB; see Figure 15. The top layer in this figure is purely an API for inserting, updating, and deleting vertices and edges, and for traversing the graphs. The API is designed for graph analytics and hides the complexities of Objectivity/DB, such as how to translate the vertices and edges to objects and relationships. The InfiniteGraph API is more specific than the very generic Objectivity/DB API, therefore, it's a lot easier to learn and use. The examples in Section 10 show how easy the API is, and how well it has been integrated into the programming language.

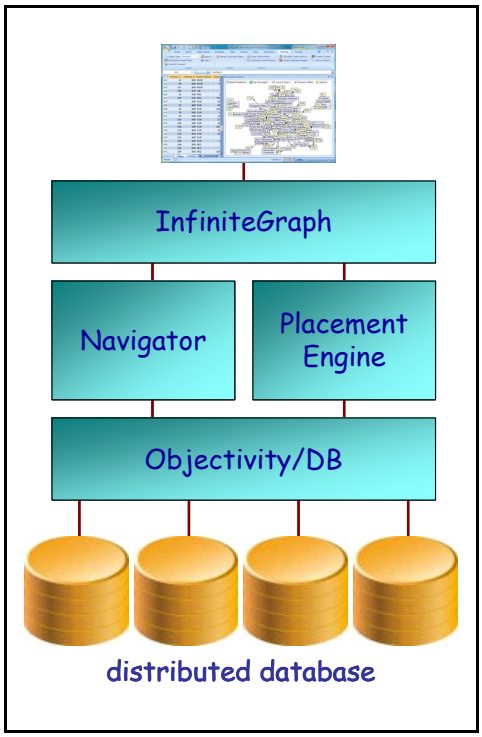

Figure 15 *The architecture of InfiniteGraph with underneath the distributed object-oriented database server Objectivity/DB*

The modules in the middle of Figure 15 do all the hard work. The module on the right, called the *placement engine*, is responsible for storing the vertices and edges in the database. A placement engine is somewhat comparable to the storage engine of a SQL database server.

Currently, two placement engines are supported: the *single database placement engine* and the *multi-database placement engine*. The former is a more simple engine, and the latter is more scalable and can handle bigger amounts of data. In the future, Objectivity might develop other placement engines.

The left-hand module in the middle is the *navigator*. This is the module that handles the graph traversals. It's the module that keeps track of all the vertices and edges already accessed when doing traversal. The smartness of this engine determines how fast a graph can be traversed. A lot of research has already gone into this module and will without a doubt continue the coming years.

Note: Technically, InfiniteGraph is not a database server, but an API (on top of Objectivity/DB). Nevertheless, because the two products are sold as one product that presents itself as a database server, we call it a database server in this whitepaper.

Partitioning Data in InfiniteGraph – InfiniteGraph has been developed to support analytics of large-scale graphs. One of the dominant features that makes the product suitable for this is support of multi-processor machines and the capability of partitioning data.

As indicated in Section 6, almost every SQL database server uses partitioning to speed up queries that need to access most of the rows of a table. Most products will have different criteria to determine which row will be assigned to which partition. A popular approach is to partition the rows based on a property, such as location.

InfiniteGraph supports partitioning as well. However, for typical graph traversals, a SQL-like form of partitioning might not be a very efficient solution. The reason can be found in the way data is accessed by InfiniteGraph. Take for example the graph in Figure 12. Assume that an application starts with member M_1 and traverses via M_2, M_4, M_6, to M_9. Also assume that the members have been partitioned based on the property location. The effect could be that every member is stored in a different partition; see Figure 16. The result is that there will be a lot of inter-process communication that will seriously slow down the traversal process.

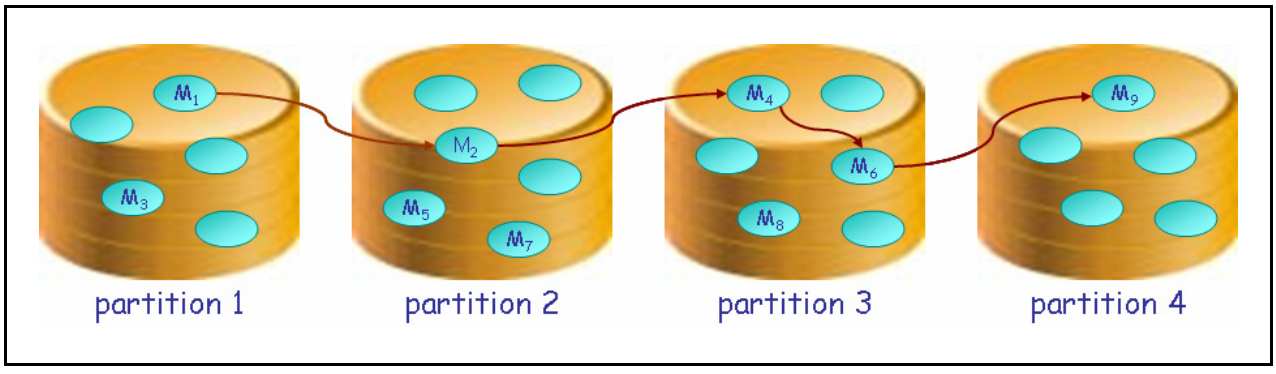

Figure 16 *Partitioning vertices based on their properties*

For this reason, InfiniteGraph supports other partitioning schemes. One is called *best effort partitioning*. Here objects are partitioned based on their inter-relationships. The more direct

and indirect relationships the vertices have, the bigger the chance they are stored in the same partition. In a way, it's like clustering data. For example, place all the members of one family in a partition; see Figure 17. This style of partitioning minimizes the amount of cross-processor traversal, and therefore improves the performance considerably.

Objectivity is in the process of developing other forms of partitioning, such as *self-healing partitioning*. This means that if the relationships of a vertex change over time, it might be that the vertex is dynamically moved to another partition.

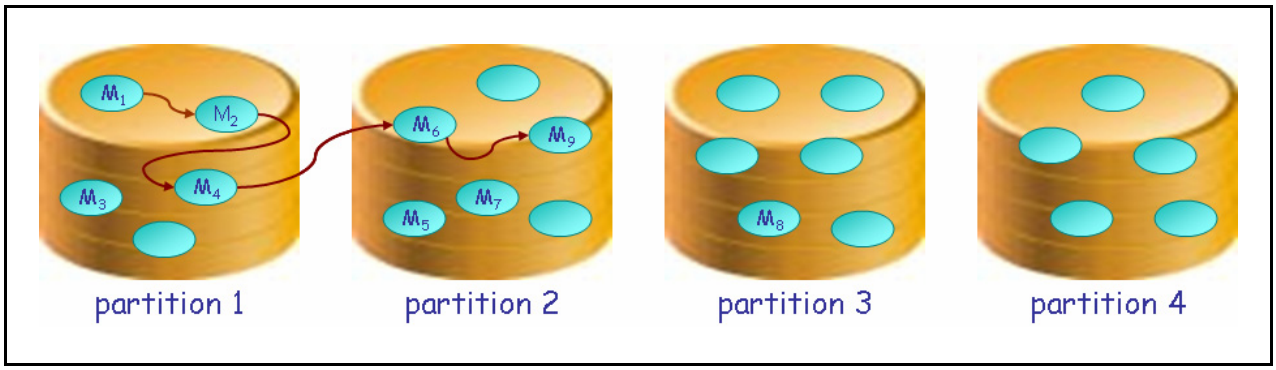

Figure 17 *Partitioning vertices based on their relationships*

InfiniteGraph and SQL-based Reporting – Classic reporting and analytical tools are not able to access the InfiniteGraph API. These tools have been developed to access database servers that support SQL or MDX. If applications do need to access data stored in the database in a SQL-style, they can use the SQL/ODBC interface supplied with Objectivity/DB called Objectivity/DB+. Use of SQL might be necessary for writing more traditional reports such as "Get the names and birth years of all the members and group them on country" or "What's the average age of all the members located in the USA and who are older than 50 years". These are not typical graph-type queries, but typical reporting queries.

The SQL interface supports standard SQL. That means that most of the queries executed by reporting tools will work. SQL3 object extensions are also supported. Note that Objectivity/DB+ is not supplied with InfiniteGraph, but is a separate product.

A challenge though is that the developers writing the SQL queries have to understand how the vertices and edges are mapped to objects in the Objectivity/DB database, and how those are presented as tables and columns. This will require some studying, but because it's documented, it can be done.

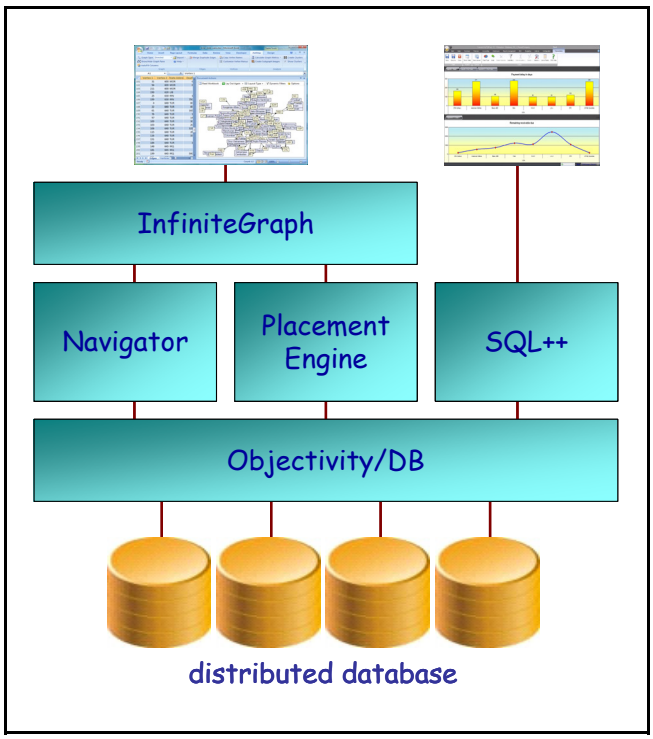

Figure 18 *The SQL interface for reporting purposes*

Transaction Support in InfiniteGraph – InfiniteGraph has been developed on top of Objectivity/DB, and therefore it inherits all its transactional features. This makes InfiniteGraph a highly safe, scalable, and robust product.

To summarize, InfiniteGraph is a scalable graph database server designed to traverse large-scale graphs and therefore to be suitable for high-end graph analytics as demanded by organizations today.

12 The Position of InfiniteGraph in Business Intelligence Architectures

There are many different ways to develop business intelligence systems. One organization might base their solution on a data warehouse extended with an operational data store plus a number of cubes; while another might decide to use a staging area and a number of data marts with conformed dimensions; a third one could choose to develop an architecture based on the Data Delivery Platform. But whatever they select, their solution is based on an architecture, a so-called *business intelligence architecture*:

> *A business intelligence architecture is a set of design guidelines, descriptions, and prescriptions for integrating various modules, such as data warehouses, data marts, operational data stores, staging areas, ETL tools, analytical and reporting tools, into an effective and efficient business intelligence system.*

In most business intelligence architectures various databases are deployed, although it depends on the architecture which ones are really implemented. There could be a database that acts as an operational data store, another one as a staging area, again another as a central data warehouse, and others as data marts. The question to be answered is: "Where can InfiniteGraph be used in business intelligence architectures?", or "As what type of data store could InfiniteGraph be deployed?"

First of all, InfiniteGraph has not been designed to act as central data warehouse. As described in this whitepaper, it's a database server designed for graph analytics and therefore doesn't have the traditional SQL interfaces. Most of the operations executed on a central data warehouse are not typical graph traversals, but more classic reporting queries. Nor would it make sense to use InfiniteGraph as operational data store or staging area. Those databases will have to process a constant stream of inserts and updates coming from the production environment. Objectivity/DB can handle this, but InfiniteGraph would not add anything beneficial.

InfiniteGraph can be deployed as a *data mart*, one that is accessed by users who need to analyze graph-structures interactively; see Figure 19. In many business intelligence architectures, data marts are created for users with homogenous information needs and who have comparable analytical wishes. Data marts are normally loaded with relevant data from the central data warehouse. InfiniteGraph could be such a data mart where graph-structured data is stored and where the users require graph analytics. Applications can be written that periodically read new data from the central data warehouse and insert it into InfiniteGraph.

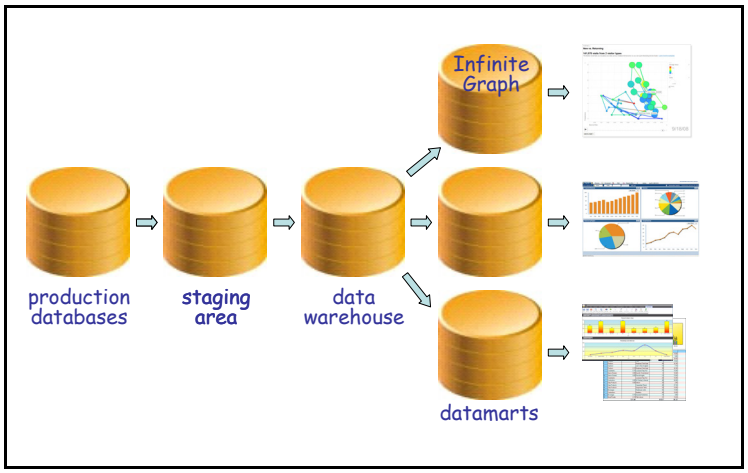

Figure 19 *InfiniteGraph as data mart in a business intelligence architecture*

In this way, the role of InfiniteGraph is not much different from what is normally done with multi-dimensional database servers that support MDX. These multi-dimensional databases are also used for building data marts and for specific analytical needs and are refreshed with new data periodically.

13 Technical Advantages of InfiniteGraph

This section lists the technical advantages of InfiniteGraph for graph analytics. The next section describes its business advantages.

Robustness and Scalability – Because InfiniteGraph is built on top of the peer-to-peer distributed architecture of Objectivity/DB, it inherits all its technical features. This means that InfiniteGraph has a highly scalable architecture capable of distributing data and processing, and has a robust foundation that can be relied upon by the InfiniteGraph developer.

Fast Processing of Massive Graphs – Because all the graph traversal is done on the server and not on a client, InfiniteGraph is capable of traversing massive graphs fast, even if they consist of millions of edges and vertices. Additionally, because InfiniteGraph keeps the traversed graphs in memory, even interactive graph traversal can be done fast.

Increased Productivity – Especially compared to SQL database servers, the dedicated and simple API for inserting and traversing graphs, makes it much easier to write applications for graph analytics. This increases the productivity of developers, particularly those familiar with Java.

Optimization of Graph Traversal – For queries that require undirected graph traversal, InfiniteGraph has added optimization techniques to improve performance and minimize resource usage.

Understanding of Graphs – InfiniteGraph understands graph-based structures. Therefore, it has built-in features to make traversal easy. For example, in most cases when graphs are traversed and the same vertex is accessed, InfiniteGraph will know that it should stop there. No additional programming is required.

Smart Partitioning of Graphs – InfiniteGraph supports partitioning of data, but not based on the properties of rows, but on the number of edges between vertices. This makes parallel traversing of graphs more efficient and minimizes cross-processor traffic.

Full Control over Performance – Developers have full control over how graphs are accessed, and therefore, they have full control over the performance. The code to traverse a graph is completely integrated with the programming language.

Efficient Data Access – Because data is stored in an object-oriented structure, it's stored in a form aimed at graph traversal. The effect is that the required I/O is minimal, and only relevant data elements belonging to the graphs are accessed. Therefore, traversing graphs is highly efficient and fast.

Carefree Database Server – InfiniteGraph (together with Objectivity/DB) requires minimal database administration effort. After the product has been installed and after data has been loaded, there is minimal need for tuning and optimizing the database server.

Open API – The InfiniteGraph API is well-documented. This makes it possible for every tool designed for analyzing graphs to act as a backend storage and query engine.

Comments on InfiniteGraph – InfiniteGraph offers some very valuable advantages for graph analytics. But there are a few considerations:

- There are no standards for graph databases, which means that the InfiniteGraph API can't be based on a standard that would make the applications more portable.
- Currently, no SQL interface exists directly on InfiniteGraph itself. This means that developers must learn how InfiniteGraph maps the graph objects to the Objectivity/DB objects before they can write the correct queries.
- Popular ETL products can't be used to load data into an InfiniteGraph database. A special program must be written to copy, for example, data from a SQL-based central data warehouse to an InfiniteGraph-based data mart.
- Currently, InfiniteGraph doesn't come with a simple interactive tool with which developers and users can easily manipulate data in the database. All manipulations must be done through an API and a programming language.

14 Business Advantages of InfiniteGraph

InfiniteGraph offers a platform for graph analytics that offers the following business advantages:

Many Forms of Graph Analytics Supported – A wide range of graph analytics forms can be used, including single path analysis, shortest path analysis, optimal path analysis, path existence analysis, and vertex centrality analysis. This means that most business problems for which graph analytics is the right solution can be solved.

Analysis of Large-Scale Graphs – With InfiniteGraph, organizations are able to analyze large-scale graph-structures fast, even if they consist of millions of vertices and edges.

Quick Development of Applications for Graph Analytics – Due to the simple API, it's a platform on which organizations can develop new analytical applications quickly.

Interactive Graph Analytics – If InfiniteGraph is extended with one of the many graphical tools on the market, the combination can make it a very interactive experience for the users, even if the graphs are large.

Analysis of Operational Data – Because of its robust and proven transactional features, it can be used in an operational analytical environment where large amounts of up-to-date data must be analyzed.

15 Case Study – A Government Agency

Objectivity's technology was first used to solve high-speed graph queries in a government application. Years before graph analysis exploded on the market, the government needed to search through large amounts of information that was highly interconnected to understand how entities were related. To do this programmatically, it required that data be persisted from a graph data model. Graph data was the only way to search for entities that might have dozens of connection points that could discovered through potentially millions of links. Early on, it was determined that relational databases were far too expensive in terms of the joins and self-joins required to query entities that were separated by four or five degrees.

The specifications for this project required a system that could handle these queries in a system that contained billions of objects that were distributed across multiple agencies. Additionally, the application was going to be used by hundreds of concurrent users that would be analyzing this data in a 24x7 environment. Given the critical nature of this application the government performed an exhaustive evaluation of the technologies available and chose Objectivity as part of the core infrastructure to support their system.

The project has been in operation for several years, performing analysis around the clock with virtually no unplanned down time or interruption. While the system has grown larger than expected, the government found their selection of Objectivity to be validated over years of operational use through its scalability, performance, robustness, and flexibility.

As the commercial market has evolved to explore similar types of analysis, Objectivity leveraged the core technology, experience and knowledge to commercial the capabilities used for graph analysis. That effort has resulted in the product known as InfiniteGraph, which is a single purpose database dedicated to help customer exploit the rich information contained in the relationships of their massive datasets. InfiniteGraph also leverages the distributed, peer-to-peer architecture the government used to support scalability that exceeded expectations, and this scalability is what helps InfiniteGraph meet the ever increasing demands of cloud and web generated data for analytics and information.

About the Author Rick F. van der Lans

Rick F. van der Lans is an independent analyst, consultant, author and lecturer specializing in data warehousing, business intelligence, service oriented architectures, and database technology. He works for R20/Consultancy, a consultancy company he founded in 1987.

Rick is chairman of the annual European Data Warehouse and Business Intelligence Conference (organized in London), chairman of the BI event in The Netherlands, and he writes for the B-eye-Network. He introduced the Data Delivery Platform in 2009 in a number of articles that were published on BeyeNetwork.com.

He has written several books on SQL. His popular *Introduction to SQL* was the first English book on the market in 1987 devoted entirely to SQL. After more than twenty years, this book is still being sold, and has been translated in several languages, including Chinese, German, and Italian.

For more information please visit www.r20.nl, or email to rick@r20.nl. You can also get in touch with him via Twitter, LinkedIn, and Facebook.

About Objectivity, Inc.

Objectivity, Inc. is the leader in distributed, scalable data management technology. Our patented distributed database engine and persistent object store and flagship product, Objectivity/DB, is the enabling technology within many markets and sectors, powering some of the most complex applications and mission critical systems used in government, business and science organizations today.

Objectivity formed the InfiniteGraph business unit in 2010 to create a product to meet the new demands of highly connected, and socially aware data. From social networking to government intelligence, the holy grail of data analysis will be finding the hidden answers by leveraging the relationships in data. InfiniteGraph's distributed architecture is designed to handle the largest and most complex graph requirements, while meeting the needs for performance and reliability.

We are committed to your success. Objectivity, Inc. has offices and representatives worldwide, and works directly with organizations, integrators and technical teams to recommend solutions and support options specifically tailored to clients' project and technical requirements.

Objectivity, Inc. is headquartered in Sunnyvale, California, USA. Please contact us online or call (408) 992-7100. Objectivity, Inc. provides direct assistance from our technical representatives and team of systems engineers, in addition to free Web-based training, and the Objectivity Developer Network, a resource for developers and evaluators that offers free open source applications, tutorials, code snippets, and other technical materials.