Università degli Studi di Padova
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Graph databases and their application to the Italian Business Register for efficient search of relationships among companies

Basi di dati a grafo e loro applicazione al Registro Imprese Italiano per la ricerca efficiente di relazioni tra le imprese

4 aprile 2017

Laureando:

Sinico Luca
1108637

Relatore:

Prof. **Ferro Nicola**

Anno Accademico 2016/2017

Università degli Studi di Padova
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Graph databases and their application to the Italian Business Register for efficient search of relationships among companies

Basi di dati a grafo e loro applicazione al Registro Imprese Italiano per la ricerca efficiente di relazioni tra le imprese

4 aprile 2017

Laureando:

Sinico Luca
1108637

Relatore:

Prof. **Ferro Nicola**

Anno Accademico 2016/2017

Abstract

"What are the main characteristics of graph databases? In what do they differ from relational ones, and in what situations should we use them?" These are the main questions of our work. We studied and tested three of the major graph databases of the moment (ArangoDB, Neo4j, OrientDB), and we compared them with a relational database, implemented with PostgreSQL. We worked on a dataset representing equity participations among companies, and we found out that the strong points of graph databases are: the purposely designed storage techniques, which let them have good performance on graph datasets; and the purposely designed query languages, which go beyond the standard SQL and manage the typical problems that arise when graphs are explored. However, we have seen that the main performance increments have been obtained when heavy graph situations are queried; for simpler situations and queries, a relational database performs equally well.

Dedicato ad Anna.

Dedicato alla mia famiglia.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Goals

In this work we will inspect the potentialities of an emerging technology in databases' field: the Graph Database. We will see how this new database model can be useful in certain data domains, and why it may be preferred to the winning and well consolidated relational model.

This work has been developed as part of an internship in InfoCamere S.C.p.A., an Italian company that covers the IT side of the Italian Chambers of Commerce ("Camere di Commercio"), which are the Italian Public Authorities responsible for dealing with the Italian production system and for providing links between the business system and other Public Authorities.

The goals of the internship were to perform an investigation on the topic of graph databases; to compare some of the available technologies on the market; to compare them with a relational solution in terms of performance; and to inspect the adoption possibilities of a graph database for some of InfoCamere's applications.

All the work will be based on data extracted by the Italian Business Register, which is maintained by InfoCamere. In particular, we will focus on "equity participation" relationships among Italian companies.

Something about benchmarking

The goal of this work is to inspect the performance of graph databases on the dataset given. It is not then a systematic work of benchmarking.

There already exist some benchmarking works available on Internet; however very few compare together the three graph DBMSs (Database Management Systems) we will analyze; and probably only one compares them with also a relational database [147]. However, this last comparison work has been realized by the developers of one of the analyzed products, so it may be somehow biased.

Some other benchmarking works are reachable by the following links in bibliography: [119], [132], [56], [116], [150], [142] and [138].

Most of the benchmark works typically focus their attention on handling concurrency or measuring performance for bulk insertions; which is not what we are mainly interest in. In addition, for what concerns query testing, they usually perform some “basic” queries, like “get or count the nodes directly connected to the node X by a single edge of type Y”; or “get all nodes that are *k* hops distant from node Z”; or “find the shortest path between nodes V and W”; but without going to high depth values (for example, see [116]).

In our case, instead, we will perform queries that are a bit more complex; mainly for what concerns results ordering and for the number of levels traversed. Indeed, with an exploration with a low depth level, a relational database could still work well; the real advantages of graph DBMSs should arise with high depth levels (that is, something like more than 3 or 4 levels, obviously depending on the graph).

It is common sense that benchmarking works that are found around should not be taken as the only parameter for making a choice for a database that would be used in an application product. Furthermore, performance are in general sensible to the statistics of the dataset given; to the hardware configuration; to the DBMS settings chosen; and to the queries performed. What should be done is to try replicate some of these tests by using a sample of the data that would be used by an application; and this is what we will do here.

Outline

In Chapter 2 we will give a background for what in general concerns graph databases, with notions about graphs, graph databases, graph computing engines, and the differences from relational databases and document stores. In Chapter 3 we will present a comparison between three graph DBMSs and a relational DBMS for what concerns aspects regarding how they store data, what are their query languages, etc. In Chapter 4 we will present the dataset that will be used for our performance tests. In Chapter 5 we will present some information about the test environment, the queries implemented and the results obtained by the execution of such queries on the four DBMSs. In Chapter 6 we will talk about how to write queries so that it is possible to display, in a graph form, the obtained results of a query. Finally in Chapter 7 we present the conclusions of the work, we give an assessment table for the compared DBMSs, and we talk about future developments.

Chapter 2

Graph Databases

In this chapter we will cover some general notions about graph databases, both of "conceptual" and of "practical" nature. We will start by giving, in Section 2.1, an introduction to the topic. In Section 2.2 we will give some general definitions about graphs, and 2.3 we will give the definition of graph database. In Section 2.4 we will present the three graph data models that are used to represent real world contexts, which are: the *Property graph*, the *RDF graph*, and the *Hypergraph*. In Section 2.5 we will present the main characteristics of graph databases. We will also present some general motivations for their adoption; the differences among them and both graph computing engines and relational databases; and also the differences from document stores. At the end of the chapter, in Section 2.6, we will explain what is the philosophy that stands at the basis of the queries that can be defined over data with a graph structure.

2.1 Introduction to Graph Databases

It has been long recognized that graphs are a natural way to represent information. Their architecture, in fact, allows to model those situations where relationships among entities hold great relevance.

Graph databases inherit and are built on top of the by now solid *Graph theory*. Graph theory was pioneered by Euler in the 18th century, and has been actively researched and improved by mathematicians, sociologists, anthropologists, and other practitioners ever since [48].

The desire to bring the expressive power of the graph to databases does not have so recent origins: graph database models have been studied even before 1990s. However it is only recently that a combination of needs and technological developments have made it possible to make graph databases a reality [7].

Gartner and Forrester, two of the most known research and strategic advisory companies in the field of Information Technology, observed the phenomenon and gave some opinions on how

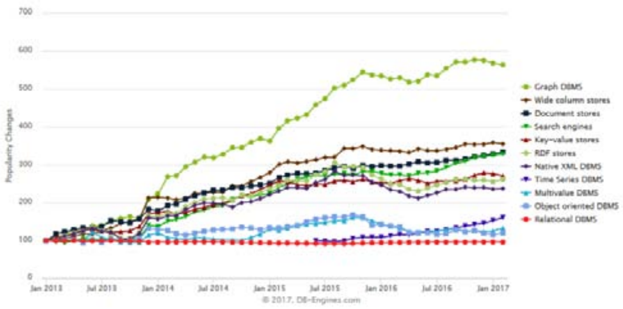


Figure 2.1: Popularity changes of database technologies during the last four years [1].

they thought that such technology would have evolved in the next years. We quote some of their statements:

"By 2017, over 25% of enterprises will be using a graph database." - Forrester [81, 123].

"By the end of 2018, 70% of leading organizations pursuing data-driven will have one or more pilot or proof-of-concept efforts underway utilizing graph databases." - Gartner [57, 100].

In addition, *"Graph analysis is possibly the single most effective competitive differentiator for organizations pursuing data-driven operations and decisions after the design of data capture."* - Gartner [39, 123].

So it emerges that the topic is experiencing good interest from businesses. In fact, as it can be seen in Figure 2.1, the popularity of graph databases was subjected to a great increment in the last four years.

Let's then start our analysis with some theoretical notions about graph fundamentals.

2.2 General definitions

Before looking at the definition of "*graph database*", we have to give some definitions about the topic that lies at its basis: the *graph*.

A **graph** $G = (V, E)$ (directed, unlabelled, without multiple edges) consists of a finite set of nodes (or vertices) V and a set of edges $E \subseteq V \times V$. There is an edge from v to w, if and only if the pair $(v, w) \in E$ (v,w). A graph can be directed or not, based on the presence of a direction information (or ordered pairs) for the edges. An undirected graph can however just be seen as a symmetric directed one. Note that self-loops - edges from a vertex to itself - are possible.

Graphs are generally represented by points or circles (vertices) connected by optionally directed arcs (edges). One example is given in Figure 2.2

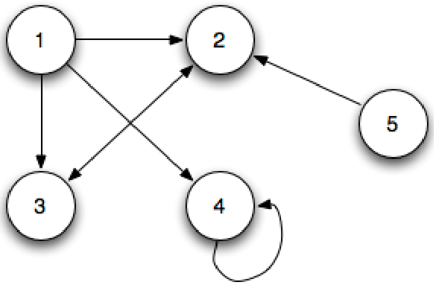


Figure 2.2: A directed graph [31].

A generalization of the graph, in which an edge can join any number of vertices, is the **hypergraph**.
Formally, *a hypergraph H is a pair $H = (X, E)$ where X is a set of elements called nodes or vertices, and E is a set of non-empty subsets of X called hyperedges or edges.* Therefore, E is a subset of $P(X) \setminus \emptyset$ where $P(X)$ is the power set of X; or E is a set of edges such that $E \subseteq (V \times ...?... \times V)$; or again, an edge is nothing more than a set of vertices [156] [101].

While graph edges are pairs of nodes, hyperedges are arbitrary sets of nodes, and can therefore contain an arbitrary number of nodes. However, it is often desirable to study hypergraphs where all hyperedges have the same cardinality: a *k-uniform hypergraph* is a hypergraph such that all its hyperedges have size k. (In other words, one such hypergraph is a collection of sets, each such set a hyperedge connecting k nodes.) So a 2-uniform hypergraph is a graph, a 3-uniform hypergraph is a collection of unordered triples, and so on [156].

There exist two standard ways to represent a graph: as a collection of *adjacency lists* or as an *adjacency matrix*; both valid for either directed and undirected graphs. The adjacency-list representation provides a compact way to represent *sparse* graphs, that is those for which $|E|$ is much less than $|V|^2$; when instead a graph is *dense* (i.e. $|E|$ is close to $|V|^2$), the adjacency matrix could be used for optimization purposes.

The *adjacency-list representation* of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $e = (u, v) \in E$.

The *adjacency-matrix representation* of a graph $G = (V, E)$ - assumed that vertices are numbered $1, 2, ..., |V|$ in some arbitrary manner - consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that $a_{ij} = 1$ if $(i, j) \in E$; 0 otherwise.

It is obviously possible to extend these two forms of representation in order to encode weighted arcs information [122].

In this work we will borrow some words belonging either to the graph and to the tree terminology for the description of the various elements that can be recognized within a graph. Here is a list of the terms we will use:

- *Nodes* or *vertices* or *vertexes* are synonyms and represent the discrete elements composing the graph. They are the constructs that can model real-world entities participating in eventual relationships.
- *Edges* or *arcs* or *links* or *hops* are synonyms and represent connections and relationships among nodes. They can be directed or undirected, and may own some values such as a weight or a name.
- In this work we also consider synonyms the words *depth*, *level*, and *distance*, which refer to the depth level - or the distance in terms of hops number - of a node with respect to a given reference node. The term level thus stands for "depth level", but will also be used for referring the set of nodes who stand at that same depth value with respect to the given vertex (e.g. level 2 is the set of nodes reachable by the initial node by following two consecutive edges).
- A *path* or a *route* is a path that starts from one source node and that, traversing consecutive edges (either with the same direction or not), reaches a destination node.
- We say that two nodes are *directly connected* if there exists between them a path made of a single edge. We say instead that two nodes are *indirectly connected* if the paths between them are made of more than one edge; that is, between them there is at least another vertex on the path.
- An edge has *outbound* or *outgoing* direction if its direction, with reference to the node considered, points out from it. An edge has *inbound* or *incoming* direction if its direction points exactly to the reference node.
- We call *directed path* a path made of only edges with the same concordant direction; an *undirected path* instead is a path made of edges without restrictions on their relative directions. In particular an *outgoing path* is a path that starts from the initial node and that is made of only outgoing edges with respect to the vertices of the path; an *ingoing path* is a path made only of ingoing edges.
- We will call for convenience *neighbors* (even if we are imposing a restriction to the standard notation) only those nodes that are directly connected to the reference node, regardless of the direction of the edges involved.
- We call *children* the nodes that are directly connected to the given node and that can be reached by means of an outbound edge. In a dual way, we call *fathers* the nodes that are directly connected to the given node and that can be reached by means of an inbound edge.
- We call *descendants* the nodes that are reachable by the initial node by means of outgoing paths. The children set of an initial node thus belongs to this set, and also their children, and so on.
- We call *ancestors* the nodes that are reachable from the initial node by means of ingoing paths. Ancestors could also have been defined as the set of those nodes that can reach the given vertex by means of outgoing paths. The same similarly holds also for *descendants*.
- We call *cycle* a directed path that starts and ends with the same vertex. It follows that a path contains a cycle if it contains the same vertex more than a time. Note that the

definition is based on the avoidance of multiple visits of the same node, and it is not based on multiple visits on the same edge. Furthermore, an undirected path that passes across the same vertex more than a time not necessarily contains a cycle, because it is only an aggregation of different directed paths (though one of them could be a cycle; in that case that path effectively contains a cycle).

- The operation of graph exploration is also called *traversal*.

One interesting characteristic of graphs, which can be useful while working with them, is that complex graphs can be seen as different webs placed on top of one another and connected by some "joints". For example, suppose we are dealing with a graph that represents both social relationships and geospatial information in a social network website, so that it is possible to suggest or search for new friends according to the place of birth, visited places, or the area where a user lives. The complex graph, made of both *user* and *place* nodes, could be split in two subgraphs: one for the relationships among users; the other for relationships among places (distance, etc). The two graphs would be brought back to be a single graph by placing those "joints" which are the relationships that connect users with places. This graph characteristic allows to query data on the two distinct domains as they were independent and separated, but also allows to perform complex multidimensional queries across both domains.

The problems that arise in a graph context - treated by the graph theory - are principally linked to routing, network flowing, covering, coloring and partitioning. The most notable problems are linked to the individuation of the shortest path between two nodes; the computation of the connected components of the graph; the computation of the minimum spanning tree and the maximum flow network; the individuation of a least-cost cycle on the graph; and others similar. Some of these problems belongs to the NP-complete class, and during the years some heuristic algorithms have been proposed to solve them in feasible time.

As for as graph traversal is concerned, there exist two main graph exploration algorithms which reflect the order by which nodes are visited starting from an initial node, which are *Breadth-First Search* (BFS) and *Depth-First Search* (DFS).

These algorithms are similar to the Pre-Order and Post-Order Traversal algorithms for tree data structures; however, graphs add an additional complexity element, given by the fact that while trees are acyclic, graphs in general are not. These kind of situations must be managed if the main focus of the exploration is to navigate the graph without "visiting" nodes more than once.

The two algorithms are presented in Figures 2.3 and 2.5. It can be seen that they use some node fields to manage the graph exploration; which are: *color*, d and π. The *color* field is used to store the visit information, in simple words: it is set to white at the beginning, to grey if it belongs to frontier between discovered and undiscovered vertices, and to black if discovered. The d field is used to compute the depth value for the node, or better the distance from the starting node. The π field is used to save the predecessor of the node.

```
BFS($G, s$)
for each vertex $u \in G.V - \{s\}$
    $u.color$ = WHITE
    $u.d = \infty$
    $u.\pi$ = NIL
$s.color$ = GRAY
$s.d = 0$
$s.\pi$ = NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u$ = DEQUEUE($Q$)
    for each $v \in G.Adj[u]$
        if $v.color$ == WHITE
            $v.color$ = GRAY
            $v.d = u.d + 1$
            $v.\pi = u$
            ENQUEUE($Q, v$)
    $u.color$ = BLACK
```

Figure 2.3: Breadth-First Search pseudocode [122].

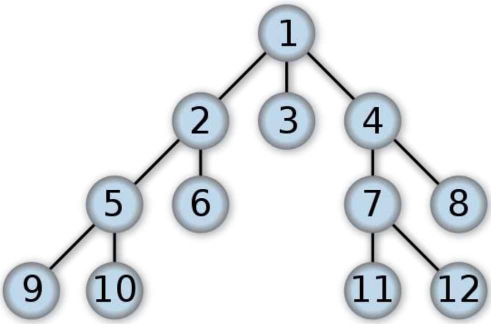


Figure 2.4: Breadth-First Search exploration order [153].

```
DFS($G$)
for each vertex $u \in G.V$
    $u.color$ = WHITE
    $u.\pi$ = NIL
$time = 0$
for each vertex $u \in G.V$
    if $u.color$ == WHITE
        DFS-VISIT($G, u$)

DFS-VISIT($G, u$)
$time = time + 1$              // white vertex $u$ has just been discovered
$u.d = time$
$u.color$ = GRAY
for each $v \in G.Adj[u]$      // explore edge $(u, v)$
    if $v.color$ == WHITE
        $v.\pi = u$
        DFS-VISIT($G, v$)
$u.color$ = BLACK              // blacken $u$; it is finished
$time = time + 1$
$u.f = time$
```

Figure 2.5: Depth-First Search pseudocode [122].

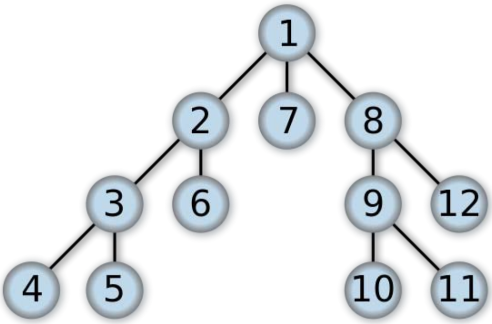


Figure 2.6: Depth-First Search exploration order [154].

The BFS algorithm constructs a breadth-first tree, that is the generated tree obtained by an exploration in breath-first order. Since we extract a tree, and by its definition of acyclic graph, those edges which start from a given node and point to an already visited node will not be included, thus the algorithm will suppress the different possible descendant paths from the initial node to the inspected node and will just keep the first path encountered with such traversal order.

The BFS traversal order is shown in Figure 2.4. In simple terms, the graph traversal starting from a given point proceeds level by level, and it will reach the more distant node only at the last step.

Let's analyze the BFS computational complexity. Supposed that enqueing and dequeuing single operations take $O(1)$ time, we have that the total time devoted to queue operations is $O(V)$. The adjacency list elements are scanned only when the vertex is dequeued, so this phase also takes $O(V)$. Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(E)$. The overhead for initialization is $O(V)$, and thus the total running time of the BFS procedure is $O(V + E)$. Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G [122].

The DFS traversal order is shown in Figure 2.6. In simple terms, the graph traversal starting from a given point proceeds path per path, where every path starts from the initial node and goes down in depth until there are not further edges to follow.

Similarly to Breadth-First Search, also the DFS algorithm works in $O(V+E)$ time. In fact, the two loops in DFS(G) take $\Theta(V)$, exclusive of the time to execute the calls to DFS-VISIT. This procedure is called exactly once for each vertex in V, because of the coloring policy. During an execution of DFS-VISIT, the loop on lines 4–7 executes $|Adj[v]|$ times; and since the sum on all $v \in V$ of $|Adj[v]|$ is $\Theta(E)$, the total cost of the loop in these lines is $\Theta(E)$ and thus the total cost for the entire algorithm is $O(V + E)$ [122].

2.3 Graph database definition

Now that we gave some definitions about graphs, we can proceed giving the definition of graph database. However, there seem not to be an official and universally accepted definition for it.

One definition could be this one:

*A **graph database** is a database that organizes data by modelling it by means of the concepts of nodes and edges:*

- *nodes represent the entities to be stored;*
- *edges represent the relationships that exist among nodes.*

However this surely is a loose and simplistic definition. A more specific definition can be the one that follows:

A ***graph database*** *is any storage system that:*

- *is designed to store graph-like data;*
- *organizes data by means of node and edge elements;*
- *provides built-in graph traversal functions;*
- *(hopefully) provides graph data integrity;*

which is very similar to the definition provided by Angles et al. in "An introduction to Graph Data Management": *A* ***graph database*** *is a database where the data structures for the schema and/or instances are modeled as a (labeled)(directed) graph, where data manipulation is expressed by graph-oriented operations, and appropriate integrity constraints can be defined over the graph structure* [7].

Another definition focused on database requirements is the one that follows:

A ***graph database*** *is an online, operational database management system with Create, Read, Update, and Delete (CRUD) methods that expose a graph data model. Graph databases are generally built for use with transactional (OLTP) systems. Accordingly, they are normally optimized for transactional performance, and engineered with transactional integrity and operational availability in mind* [48].

However, some others define the graph database in a completely different manner:

A ***graph database*** *is a database that uses graph structures with vertices, edges, and properties to represent and store data. A graph database is any storage system that provides index-free adjacency* [42, 114].

The last definition is mainly based on a single property, which depends on the way by which the database management system stores the graph data. The *index-free adjacency* property will be better explained later on; however the fundamental idea is that the adjacency information among nodes is not derived by the use of indexes, but is directly obtainable by the node itself by means of physical pointers or similar kinds of link.

It is quite delicate to state that only one should be the right definition and that the other definitions are completely wrong; so we will assume them as equally valid.

As far as retrieving data is concerned, some new concepts and generally a new query language are required with respect to the standard query language of relational databases. There is not a standard query language for this domain: each graph database product comes with its own query language and, up to now, none of these query languages has risen to prominence in the same fashion as SQL did for relational databases. Some standardization efforts however took place, leading to systems like Gremlin (which works with a variety of graph engines that realize the property graph model), and SPARQL (which is used by those that realize RDF graphs) [158]. Further details on this aspects will be given later.

Note that it is not assured that commercial graph database products provide, with their query languages and APIs, traversal functions that realize both DFS and BFS algorithms. In addition, it is also not assured that the provided functions implement the idea of avoiding multiple visits on nodes, which instead is at the base of the two algorithms provided. Further details will be given in next chapters.

2.4 Graph data models

One way to distinct graph databases is by looking at the data model they implement. There are three dominant graph data models: *property graph*, *RDF graph*, and *hypergraph* [48]. Talking about them, we could state that while the *RDF* model constitutes a W3C standard, the *property graph* is more an industry standard [6].

2.4.1 Property graph

The definition of property graph is simple and much intuitive.

A **property graph** is a graph that has the following characteristics:

- It is made up of nodes, edges, properties, and labels (or similar categorization forms).
- Nodes contain an arbitrary number of properties, which typically are key-value pairs.
- Edges connect nodes and structure the graph. Each edge has a direction, a single name, and a start node and an end node - there are no dangling edges.
- Nodes and edges can be categorized in one or more categories (by means of classes or labels). Categories group nodes together, and indicate the roles they play within the dataset. Edges can also be categorized. Together, edge direction and type (category) add semantic meaning to the network structure.
- Also relationships can have properties. The ability to add properties to them is particularly useful for enhancing the semantics of the graph, and for limiting the results at query-time [48].

So a property graph simply is an enrichment of the mathematical definition of the graph given in Section 2.2. In Figure 2.7 we show a quick visual representation of a property graph.

This kind of model obtained a good success and most of the popular graph DBMSs on the market realize this data model. Furthermore, with the Apache TinkerPop project, there has been an attempt to lay down a standard for this form of data organization, which is gaining more and more adoption within the field. Apache TinkerPop defines itself as an open source, vendor-agnostic, graph computing framework for both graph databases (OLTP) and graph analytics systems (OLAP); and it is distributed under the commercial friendly Apache v2 license [127].

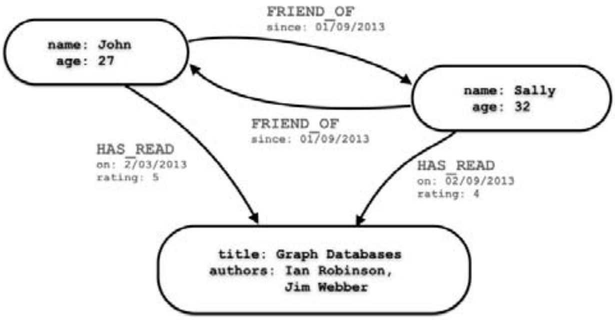


Figure 2.7: Visualization of a property graph [62].

In Figure 2.8 we show TinkerPop's architecture.

For graph analytics systems, TinkerPop allows to express graph analytics algorithms; while for graph databases it allows to perform graph data modifications and to express traversal queries.

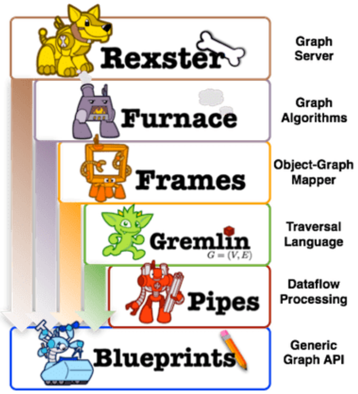


Figure 2.8: TinkerPop system architecture [125].

Gremlin is the traversal query language that comes with TinkerPop, and it is implemented in Groovy. It allows to express a traversal in either an imperative manner and a declarative manner. An imperative traversal tells the traversers how to proceed at each step; while the declarative traversal allows each traverser to select a pattern to execute from a collection of (potentially nested) patterns. In both cases, the user's traversal is later rewritten by a set of traversal strategies which do their best to determine the most optimal execution plan, based on an understanding of graph data access costs as well as the underlying data systems's capabilities [126].

However, not all graph databases are based on Gremlin for their query language; rather, most of them developed their own declarative query language. The reasons for this is that with a declarative language there typically is more room for letting the server perform query planning analysis and optimizations; and it is easier to create languages designed for remote invocations or query-writing simplifications [80].

2.4.2 RDF graph

The formal definition of **RDF**, given by W3C, is:
"Resource Description Framework (RDF) is a standard model for data interchange on the Web. RDF has features that facilitate data merging even if the underlying schemas differ, and it specifically supports the evolution of schemas over time without requiring all the data consumers to be changed."

In other words, RDF extends the linking structure of the Web to use URIs to name the relationships between things as well as the two ends of the link. The core structure of this model is a set of **triples**, each consisting of a *subject*, a *predicate* and an *object*. A set of such triples is called an **RDF graph**; or a **triples store**.

There can be three kinds of nodes in an RDF graph: *URI* nodes, *literal* nodes, and *blank nodes* [145]. URI stands for *Universal Resource Identifier* and is a string of characters used to identify a resource. The most common type of URI is URL (Uniform Resource Locator), which is used to identify Web resources. Literal nodes are used for holding values such as strings, numbers, and dates. Blank nodes instead represent anonymous resources, i.e. those for which a URI or literal value is not given. According to the standard, a blank node can only be used as subject or object of an RDF triple.

The *subject* is typically identified by a URI, while the *object* can be a URI or a literal node. *Predicates* are also typically identified by URIs and can be interpreted as either a relationship between the two nodes or as defining an attribute value (object node) for some subject node [145]. The fact that both verbs and resources can be identified by URIs enables anyone to define a new concept (both verb and data), just by defining a URI for it somewhere on the Web. This new concept would acquire world-wide meaning and (possibly) visibility, and allows anyone to use it for declaring its graph and maybe connect it to some others [124]. Blank nodes may be given a document-local identifier called a blank node identifier [146].

As a consequence of its architecture, the RDF graph, from a low-level point of view, is made of arcs that connect entities with both their attributes and other entities. However, from a high-level point of view, the RDF graph realizes a directed labeled graph made of edges among entities [145]. In Figure 2.9 we show a visual example of this.

Furthermore, by using this simple model, RDF allows structured and semi-structured data to be mixed, exposed, and shared across different applications [144].

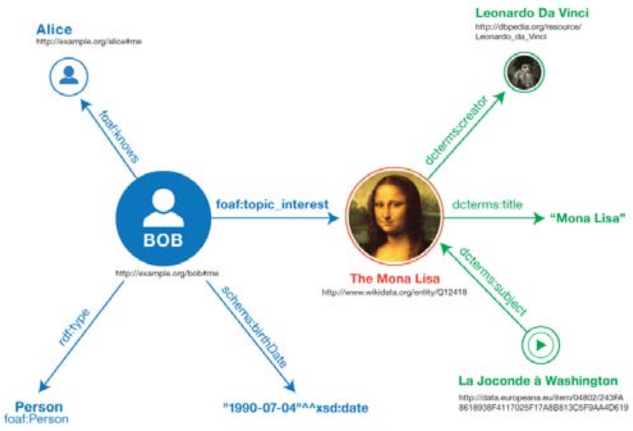

Alice
http://example.org/alice#me
Leonardo Da Vinci
http://dbpedia.org/resource/Leonardo_da_Vinci
foaf:knows
dcterms:creator
BOB
foaf:topic_interest
dcterms:title
"Mona Lisa"
http://example.org/bob#me
The Mona Lisa
http://www.wikidata.org/entity/Q12418
dcterms:subject
rdf:type
schema:birthDate
La Joconde à Washington
http://data.europeana.eu/item/04802/243FA8618938F4117025F17A8B813C5F9AA4D619
Person
foaf:Person
"1990-07-04"^^xsd:date

Figure 2.9: Visualization of a RDF graph [141].

RDF storage techniques

The way by which an RDF can be implemented - i.e. the underlying storage model - is not unique. RDF schemas (metadata) and instances could be efficiently accessed and manipulated in main memory; or they could be serialized to files, for persistence purposes; or even (for large amounts of data) a more reasonable approach is by using a database management system. Many of the existing RDF stores use relational and object-relational database management systems (RDBMS and ORDBMS) for this purpose. Storing RDF data in a relational database however requires an appropriate table design; and within them there are two main approaches, which could be classified as generic schema (or schema-carefree), i.e. schemas that do not depend on the ontology; and ontology specific schema (or schema-aware) [3].

The paper "*A survey of RDF storage approaches*" by Faye, Curé and Blin [38] suggests a classification of the RDF storage techniques as shown in Figure 2.10. In the figure is highlighted the difference between native and non-native storage techniques, also dividing them in different sub-types based on the underlying storage adopted.

There exist many serialization formats that support RDF: RDF/XML, Turtle, N-Triples, N-Quads, JSON-LD, N3. They are all human-readable and focused on being easy-to-parse; in addition, some of them also allow to encode inference rules. Let's then see a quick example on how RDF triples may be encoded with the RDF/XML format. This example makes use of one of the existing ontologies (see the next subsection for ontologies), which is FOAF (Friend Of A Friend): realized for describing persons, their activities and their relations to other people and objects.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
```

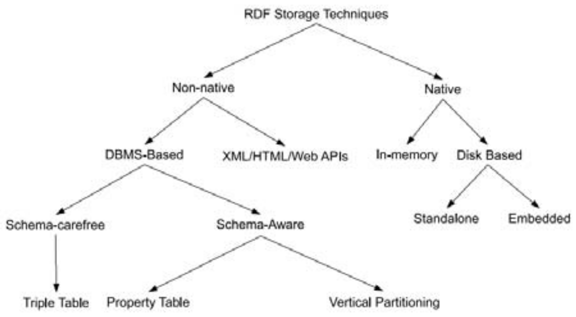


Figure 2.10: RDF classification based on storage technique [38].

```
         xmlns:foaf="http://xmlns.com/foaf/0.1/">

 <foaf:Person rdf:nodeID="mark">
   <foaf:name>Mark</foaf:name>
   <foaf:mbox rdf:resource="mailto:mark@email.com"/>
   <foaf:knows>
       <foaf:Person>
         <foaf:name>John</foaf:name>
       </foaf:Person>
   </foaf:knows>
 </foaf:Person>

 <foaf:Person rdf:nodeID="peter">
   <foaf:name>Peter</foaf:name>
   <rdfs:seeAlso rdf:resource="http://www.peter.com/peter.rdf"/>
   <foaf:knows rdf:nodeID="mark"/>
 </foaf:Person>

</rdf:RDF>
```

The example describes then a situation in which we have three nodes of type *Person*; one has name Mark, one John and the other Peter. Mark owns an email adress, while Peter has an RDF description for itself. In addition, Mark knows John, and Peter knows Mark.

Talking about triples stores, some have been built as database engines from scratch; others instead have been built on top of existing commercial relational database engines or NoSQL document-oriented database engines. This approach of exploiting existing databases allowed large and powerful database engines to be constructed for little programming effort, and also allowed a better management of big amounts of data. However, the implementation of triples stores over relational databases can be tricky because, although triples may be stored in them, the implementation of efficient querying mechanisms (e.g., mapping from SPARQL) onto SQL queries for data domains made of triples is hard. For this reason, a good design of a *native triples store* may bring better performance [157] with respect to a non-native solution.

Ontologies

Because of the fact that URIs are used by RDF to encode the information, concepts become not just words in a document but are tied to a unique definition that everyone can find on the Web. This form of shared definition is called **ontology** or *vocabulary*. Let's consider this example, provided by Tim Berners-Lee et al. in [124]: "imagine that we have access to a variety of databases with information about people, including their addresses. If we want to find people living in a specific zip code, we need to know which fields in each database represent names and which represent zip codes. RDF can specify that '(field 5 in database A) (is a field of type) (zip code)', using URIs rather than phrases for each term. However, two databases may use different identifiers for what is in fact the same concept, such as zip code. A program that wants to compare or combine information across the two databases has to know that these two terms are being used to mean the same thing. Ideally, the program must have a way to discover such common meanings for whatever databases it encounters. A solution to this problem is provided by the third basic component of the Semantic Web, collections of information called ontologies." These ontologies (or vocabularies) represent then one of the most distinctive components of RDF.

The first role for ontologies is then to help data integration when, for example, ambiguities may exist on the terms used in the different datasets; or when a bit of extra knowledge may lead to the discovery of new relationships. The other role of ontologies is to organize knowledge, maybe coming from datasets own by different organisations, in collections in order to diffuse standard and shared terminology and to allow linked data [139].

Inferences

Another important component of RDF is the possibility to perform inferences (or reasoning) among data.

Inference means that automatic procedures performed by inference engines (or "reasoners") can generate new relationships based on data and some additional information in the form of an ontology. The new resulting relationships could be explicitly added to the dataset; or simply returned at query time, according to the implementation of the engine [137]. In this way, in addition to retrieve information, it is also possible to use the database to deduce new information by examining facts (assertions) in the data.

Inferences can be used to improve the quality of data integration on the Web, by discovering new relationships, automatically while analyzing the content of the data; or by managing the knowledge on the Web in general. Inference based techniques are also important in discovering possible inconsistencies in the (integrated) data [137].

Let's take the classical syllogism as example, and see what inferences mean: suppose that in the RDF graph is encoded the information "all men are mortal" and "Socrates is a man". It is possible then to infer the following conclusion: "Socrates is mortal". This comes useful, for example, when working with entities that may belong to classes and subclasses; in fact inferences would put in relevance the fact that a record belonging to a subclass also belongs to its super-class. The inference ability is provided by means of rules defined on meta-data, which may be stored together with data itself. They are typically encoded by using *OWL* (Web Ontology Language), which adds the ability to express those concepts like relationships between classes (e.g. *disjointWith*); equality (e.g. *sameAs*); richer properties (e.g. *symmetrical*); and class property restrictions (e.g. *allValuesFrom*) [54]. It is then typically used by the applications that explore the graph and that have to manage some of the problems that may arise when more distributed sources of data are used [140].

Let's consider now another famous example: "Epimenides says that all Cretans are liars" and "Epimenides is a Cretan"; the database might (if programmed to avoid getting into endless loops) point out that here there is a contradiction in the data. Other graph databases typically do not support by default such things; however some libraries and frameworks are going through this direction (see Apache TinkerPop [127]) [34].

Of course, for semantic web to "function", computers must then have access to structured collections of information and sets of inference rules that they can use to conduct automated reasoning. One of the challenges of the Semantic Web, therefore, has been to provide a language that expresses both data and reasoning rules; and that rules, from any existing knowledge-representation system, could be exported onto the Web [124].

SPARQL

SPARQL, pronounced "sparkle" [32], is a recursive acronym for "SPARQL Protocol And RDF Query Language". It is an RDF query language, i.e. a semantic query language for databases, able to retrieve and manipulate data stored in Resource Description Framework (RDF) format [159]. It is not the only one existing RDF query language; however it is the W3C Recommendation for this purpose.

SPARQL can express queries both on data stored natively in RDF format and on data viewed as in RDF format via middleware. It contains capabilities for querying required and optional graph patterns, it supports aggregation, subqueries, negation, and limitation on the query by RDF source graph. The results of SPARQL queries can be result sets or RDF graphs [143].

SPARQL queries are based on (triple) patterns. These triple patterns are similar to RDF triples, except that one or more of the constituent resource references are variables. A SPARQL engine would returns the resources for all triples that match these patterns. One query example is the following, where we ask for John's friends:

```
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?N
WHERE { ?X sn:type sn:Person . ?X sn:firstName ?N .
        ?X sn:knows ?Y . ?Y sn:firstName "John" }
```

The combination of the RDF standard format for data, and of the SPARQL standard query language, permits the existence of an extended version of the actual Web, which is the Semantic Web.

Semantic Web and Linked Data

The key concept of Semantic Web can be summarized in these words by Tim Berners-Lee, one of the founders of the World Wide Web: "A new form of Web content that is meaningful to computers" [124].

Semantic Web is thus an effort to bring structure to the meaningful content of web pages; to let be possible a guided exploration of their contents by the introduction of semantic information; to give a better processing support for other kind of data (different than from the web pages documents) which are media and structured data; and more in general to provide a means of integration over disparate sources of information [143].

The term *Linked Data* mainly refers to a set of best practices for publishing and connecting structured data on the Web [44]. To make the Web of Data a reality, it is important that both nodes and relationships are published on Internet; so that inbound and outbound connections are possible and the local graph becomes attached to other graphs. In Figure 2.11 we show how data coming from different data sources and technologies can be brought to be published.

Semantic Web technologies can be used in a variety of application areas, for example: in data integration, whereby data in various locations and various formats can be integrated in one; in resource discovery and classification, to provide better domain specific search engine capabilities; in cataloging for describing the content and content relationships available at a particular Web site, page, or digital library; by intelligent software agents to facilitate knowledge sharing and exchange; for content rating; for describing collections of pages that represent a single logical "document"; for describing intellectual property rights of Web pages (see, eg, the Creative Commons), and in many others [45]. A typical case of a large Linked Dataset is DBPedia, which, essentially, makes the content of Wikipedia available in RDF. The importance of DBPedia is not only that it includes Wikipedia data, but also that it incorporates links to

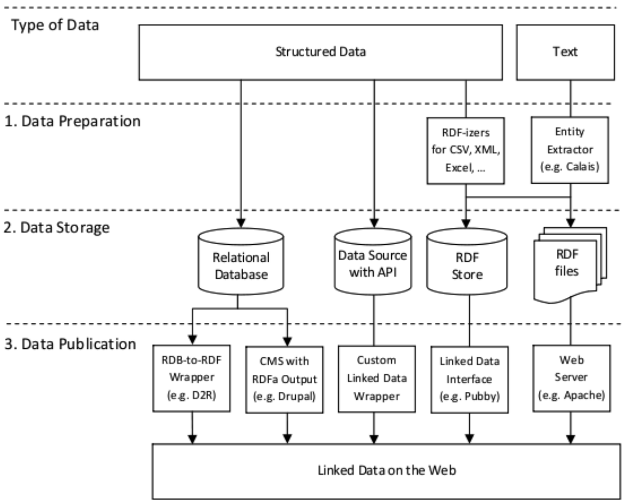

Type of Data
Structured Data
Text
1. Data Preparation
RDF-izers for CSV, XML, Excel, ...
Entity Extractor (e.g. Calais)
2. Data Storage
Relational Database
Data Source with API
RDF Store
RDF files
3. Data Publication
RDB-to-RDF Wrapper (e.g. D2R)
CMS with RDFa Output (e.g. Drupal)
Custom Linked Data Wrapper
Linked Data Interface (e.g. Pubby)
Web Server (e.g. Apache)
Linked Data on the Web

Figure 2.11: Linked Data publishing options and workflows [131].

other datasets on the Web, e.g., to Geonames. By providing those extra links (in terms of RDF triples) applications may exploit the extra (and possibly more precise) knowledge from other datasets when developing an application; by virtue of integrating facts from several datasets, the application may provide a much better user experience [136].

Figure 2.12 shows the diagram of Linked Open Data in May 2007. Figure 2.13 shows instead the situation in August 2014: the amount of data and the number of involved domains has undergone a great enhancement; RDF and Linked Data seem to gain popularity and adoption during years.

One note to be given is the following: the use of RDF graphs does not implies the fact that these connected data must also be Open, in the sense of being accessible by anyone. Indeed it is possible to access the data (by means of Semantic Web constructs) only if the data and relationships URIs have been shared and access grants are given; otherwise data will not be reachable from not allowed people or software agents. Thus, this technology could be also useful in cases of distributed, but not open, data sources that have to be integrated [44, 55].

Differences from property graphs

With RDF graphs the information relative to nodes and their relationships is organized differently from the databases that implement the property graph model; in fact, with RDF there are not two separate constructs that represent the entire node (and its properties), and the

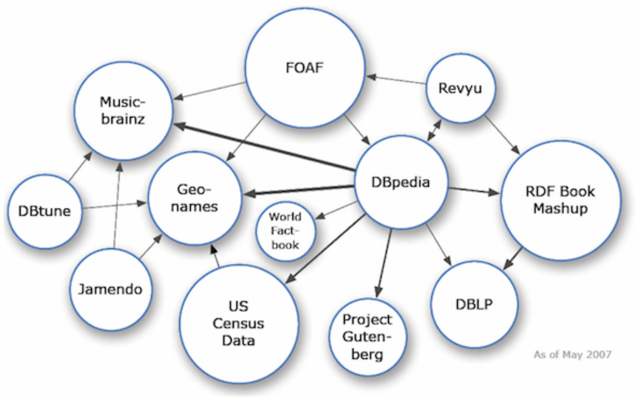


Figure 2.12: Linking Open Data cloud diagram in May 2007 [58].

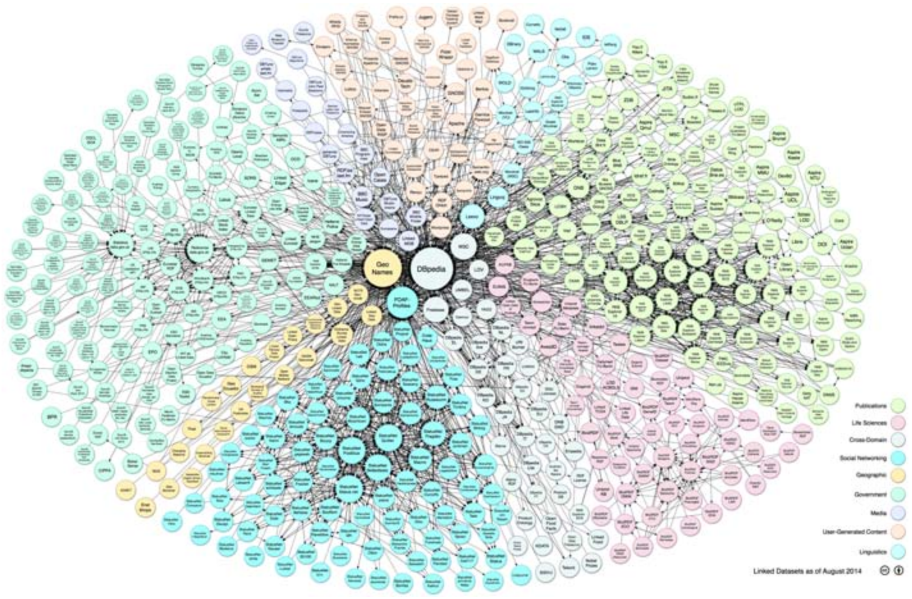


Figure 2.13: Linking Open Data cloud diagram in August 2014 [58].

relationships among nodes. In fact, nodes (*subjects*) have their proper representation, but their properties and relationships are represented by the same mediums, that is by using an arc (*predicate*) and a connected node (*object*); where the object can represent a value for the property suggested by the *predicate*, or a relationship to an another node.

Since properties are realized by means of links to other nodes which contain the property value, and since only nodes can be used as subject for a triple, it follows that there is not a direct way to express properties on those arcs that represent relationships among node entities. This is precisely the reason why RDF does not fully realize the property graph model: the only property that relationships can hold is their "name", or better, their type.

One way to encode relationship' properties is by performing a sort of "reification", i.e. by transforming the relationship in a node. In particular, suppose we want to define a distance value (15 kilometers) for a relationship of type 'highway' that connects the two nodes X and Y. The ways by which one could encode such information with RDF are:

- by define an ontology which only represents highway distances in kilometers (<http://www.highway_distances_in_kilometers.org/>), where the types for the relationships are the valid numeric values. In this way, it would be possible to define highway distances like:

  Code snippet 2.1: SPARQL query with custom ontology for highway distances.

  ```
  PREFIX highway_distance_km:
     <http://www.highway_distances_in_kilomeeters.org/>
  PREFIX hghw: <http://www.highway.org/>
  SELECT ?N
  WHERE { ?X hghw:type hghw:Tollbooth .
          ?X hghw:tollbooth_name ?N .
          ?X highway_distance_km:15 ?Y .
          ?Y hghw:tollbooth_name "Padua Sud" }
  ```

  Alternatively, by define multiple types for relationships which encode the distance information, like "15_km".

  Code snippet 2.2: SPARQL query with highway ontology and distance types.

  ```
  PREFIX hghw: <http://www.highway.org/>
  SELECT ?N
  WHERE { ?X hghw:type hghw:Tollbooth .
          ?X hghw:tollbooth_name ?N .
          ?X hghw:15_km ?Y .
          ?Y hghw:tollbooth_name "Padua Sud" }.
  ```

  However, the two solutions involve the definition of multiple relationship types, one for each valid distance measure. Furthermore, just one property is added to the relationship with this approach. More properties may be added by extending the string which contains the property value; however this clearly is a stretch.

- by creating a node that represents the relationship type (highway_sector), by using it as object and by connecting it with the subject X by a generic relationship type (like "connected_to"), by connecting it with the object node Y by a relationship "connected_to", and by assigning a value to this node named "highway_sector" by connecting it as a subject to an object representing the 15km value by a predicate called "length". This second approach is more feasible, but clearly dismantle the way in which RDF has been designed.

Another point of divergence is the fact that even the standard query language for the RDF graph shows some limitations in its expressive power. For example, with SPARQL it is possible to write queries that use a variable-length path formulation, which is used for searching the descendants of a given node; however it is not provided a way for returning the paths that reached them. In addition, it is not possible to impose a maximum length value for the variable-length path formulation; so the only way for searching the descendants with depth value less than 10 is by writing ten chained patterns within the query body, and by returning also the intermediate nodes. However it is clear that this approach is not feasible if there is the desire to express the maximum depth value as a parameter. Another example is that in SPARQL there is not a built-in function that computes the shortest path between two nodes. Such operation would be left in charge of the developer, which would implement it by means of high-level languages that iteratively perform a bunch of "small" queries for obtaining the nodes reachable step-by-step. Finally, since RDF does not support properties on relationships, it is not obviously possible to filter traversed edges by some characteristics that could regard them. Further examples are given by Renzo Angles and Claudio Gutierrez in "*An introduction to Graph Data Management*" [7].

These kinds of query are instead typically possible for property graph databases by using Gremlin or the query languages provided by the DBMSs themselves.

2.4.3 Hypergraph

Another way to describe graph data is with a *hypergraph*. Hypergraphs can be useful when the dataset includes a large number of many-to-many relationships. However, with such hyperedges, it is simple to lose the possibility of specifying some fine-grained edge details for the relationship represented by the edge. Let's see an example: in the (directed) hypergraph shown in Figure 2.14 we see that Alice and Bob are the owners of three vehicles; we can express this fact by only using a single hyperedge. In a property graph, instead, we would have to use six relationships to express the same fact, as shown in Figure 2.15.

By using six relationships instead of one, however, there are two advantages:

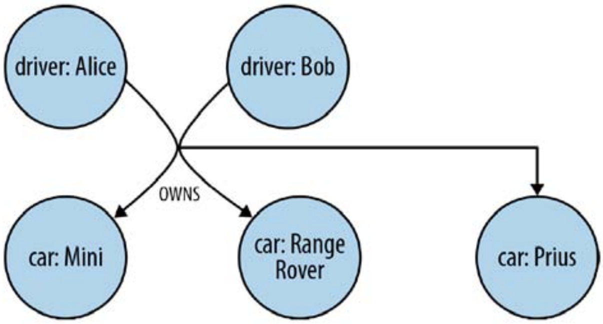


Figure 2.14: Directed hypergraph example [48].

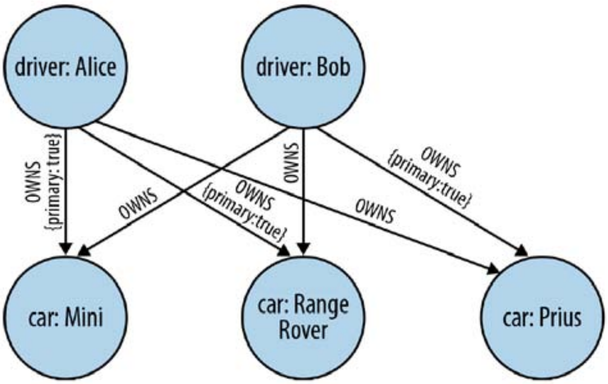


Figure 2.15: Hypergraph translated in a property graph [48].

- First, we're using a more familiar and explicit data modeling technique (resulting in less confusion for a development team).
- Second, we can also fine-tune the model with properties on relationships, such as, for example, a "primary driver" property useful for insurance purposes. The same thing cannot be done if a single hyperedge is used [48].

Because of the multi-dimensionality of hyperedges, hypergraph models are more general than property graphs. Yet, the two are isomorphic, so you can always represent a hypergraph as a property graph (albeit with more relationships and nodes). However, the opposite conversion is not so immediate and it depends on the information stored [117].

While property graphs are widely considered to have the best balance of pragmatism and modeling efficiency, hypergraphs show their particular strength in capturing meta-intent. For example, if you need to qualify one relationship with another (e.g. "I like the fact that you liked that car"), then hypergraphs typically require fewer primitives than property graphs [48]. The hypergraph data model did not raise the same adoption as the other two data models shown before, and very few graph DBMSs manage data according to this model; one example for them is HypergraphDB [47].

2.5 Graph database characteristics

Graph databases can be distinct according to two aspects: the *underlying storage* and the *processing engine.*

Based on the storage mechanism, a graph database is said to be a **native graph storage** if organizes and stores graph data with an expressly built graph model architecture. On the other hand, it is said to be a *non-native graph storage* if it stores graph data with some already existing technologies (e.g. relations for a relational database, files, documents, etc).

Based on the processing engine, graph databases can be distinct by whether they realize or not the index-free adjacency. In particular, a graph database is said to be a **native graph processing engine** if it realizes the index-free adjacency; otherwise it is said to be a *non-native graph processing engine.*

It is straightforward that the relational model is neither a native graph storage, nor a native graph processing engine.

A database engine that realizes ***index-free adjacency*** is one in which each node maintains references to its adjacent nodes; or to the connected edges, through which the adjacent nodes are reached. Each node, therefore, acts as a "micro-index" of nearby nodes. This approach is thus in contrast with all those solutions that make use of indexes for finding out connections among data.

The index-free adjacency is somehow bound to what is called the *locality principle of graph queries.* This principle states that, in order to answer to a standard query (i.e. not a query for global graph analytics), only the portion of the graph that is reachable by the specified node will be taken in exam. This principle is at the foundation of some types of query balancing techniques, as will be shown later in Chapter 3.

Let's now examine the potential benefits of an *index-free adjacency* approach. Suppose we want to start a query from a specified node and to search for its descendants that hold a particular property value. Suppose also that a classic binary tree index is used by the database which stores such data. If the database does not support the index-free adjacency, it means that to traverse a network of m steps, it will take $O(m \log n)$ time; where n is clearly the number of total nodes indexed in the graph. However, for an implementation that uses index-free adjacency, the cost would be only $O(m)$ [48]. This is due to the fact that, because the information about connections is already collected by each vertex, the cost to find and traverse each connected edge is *O(1)*.

This is the typical argument provided by those graph databases that realize such property within their engine. However, this argument is only valid with the assumption that the databases that do not support the index-free adjacency make use of indexes that have a *O(log n)* algorithmic complexity for lookups, like the ones with binary tree structure. However, if a Hash index is used, its lookup will have a complexity of *O(1)*. In contrast, the limitation of hash indexes is that they are only valid for direct text matching, and it is not possible to use them for partial text matching or for performing range searches.

Figure 2.16 gives a quick view on how some of the DBMSs on this field are placed within the "graph database space".

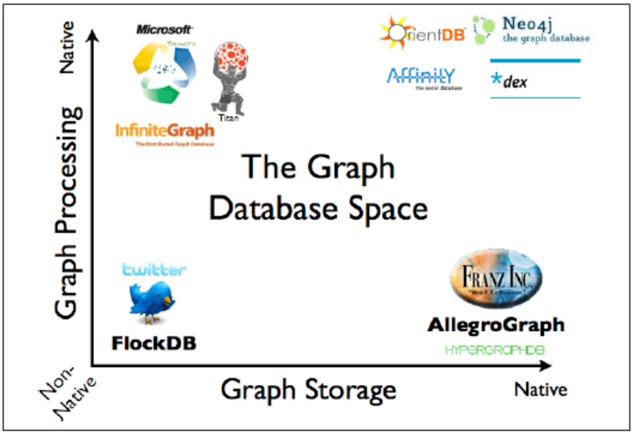


Figure 2.16: The graph database space [48].

There is then a sort of conflict among different graph database products on what should be the characteristics of a database in order to effectively be considered as a graph database. This is also the reason why there is not a uniquely accepted definition of what a graph database is. As seen, some state that a graph database must have the index-free adjacency property; others state that it is wrong to consider a database as a graph database only if it shows such characteristic [148].

From a theoretical point of view, in contrast to a relational database (where join-intensive query performance deteriorates as the dataset gets bigger because of the increased size of indexes); with a graph database (better if with index-free adjacency) performance should remain relatively constant, even if the dataset grows, because the execution time for each query is proportional only to the size of the part of the graph that is traversed to satisfy such query [48].

It is not true however that query times, in a database with index-free adjacency, are totally independent of the size of the graph. In fact, given the particular initial node, it is right that the traversal will be quite independent of the graph size; however the query have to find the node before the traversal can start. Such preliminary phase of the query depends on global indexes, which in turn are dependant of the graph size; so the execution time of the query will be affected by it.

2.5.1 Motivations for the adoption of graph databases

It is now clear that graph databases are a useful instrument when we want to use the general features provided by databases (persistence and reliability, data integration and consistency, isolation and atomicity, etc.), and when the real world data we want to model exposes a graph structure.

During last decades the necessity of being able to model such data domains has been subjected to an interesting increment; for example think about all the web sites that realize social networks. Social networks and recommendation systems indeed are clear examples of how finding out connections, and gathering nodes according to some graph topological characteristics, represent the true business activity.

"*We live in a connected world. There are no isolated pieces of information, but rich, connected domains all around us. Only a database that embraces relationships as a core aspect of its data model is able to store, process, and query connections efficiently.*" - This is the introduction made by one of the most known graph databases (Neo4j) on the reasons why the world should use graph databases.

The more the data domain is connected, the more the real value of it comes from relationships. Facebook, for example, was founded on the idea that while there's value in discrete information about people - their names, what they do, etc. - there's even more value in the relationships between them. Similarly, Larry Page and Sergey Brin figured out how to store and process not just discrete web documents, but how those web documents are connected: Google captured the web graph [48].

Common use cases for graph databases are on fields about social graphs, recommendation systems, business relationships, dependencies (network impact analysis), geospatial applications (road maps and route planning for also rail network or logistical networks), telecommunication or energy distribution networks (network management and analysis), master data management (management of distributed data sources), access control, fraud detection, etc [4, 48].

One usage example in the field of recommendation systems is: users and products are modeled like two different kind of vertexes; between users there are 'friend_of' relationships and between users and products there are 'purchased' and 'likes' relationships. By storing the purchase history of each user and its favourites list it is possible to leverage this information to suggest additional interesting purchases to his friends or other people that shown similar purchase characteristics. Taken together, social networks and recommendation engines provide key differentiating capabilities in the areas of retail, recruitment, sentiment analysis, search, and knowledge management [48].

The success of graph databases on such fields is not only due to their natural graph data models, but also because of the query functions or APIs provided. These query interfaces allow to express more easily the operations to be done on such a graph structure, with respect to the SQL language. Querying a graph stored in a relational database, on multiple depth levels and by following only some types of edges, with the provided SQL language is indeed quite complex.

Another field where graph databases are collecting success is fraud detection. This is done by discovering on graph some relationships patterns that are unusual or properly fraudulent. Patterns outside of the norm for a given user can be detected on the graph and then flagged for further attention.

Let's see a short explanation of the reasons why businesses look at graph databases:

Performance
One reason for inspecting and maybe choosing a graph database is the hypothesized performance increase when dealing with connected data. This is due to its ability in retrieving those nodes that have, for example, only certain incoming types of edges and not other types; or for the built-in ability of working on weighted graphs and to compute shortest or least-cost paths. Performance are also helped by the *locality principle of graph queries* seen before.

Scalability
Some graph databases allow horizontal scaling directly at data level, which allows for large graphs to be stored as one but in different devices. Horizontal scalability may help in saving on hardware costs; however for graphs it does not come so easily.

Developer agility and schema flexibility
Graphs are naturally additive, i.e. new kinds of relationships, nodes, labels, and in general subgraphs can be added to an existing structure without disturbing existing queries and application functionality. These aspects may bring positive implications for developer productivity and project risk. Because of the graph model's flexibility, we don't have to model our domain in exhaustive detail ahead of time in order to minimize further changes, both in terms of structure (relationship types, labels) and in terms of nodes and relationships properties. In addition, the schema-free nature of the graph data model avoids those schema-oriented data governance operations we're familiar with in the relational world [48].

However, this does not clearly implies only a benefit, because schema flexibility can bring some troubles on later moments of the database life cycle.

Intuitive data model
Graphs are familiar and intuitive to both users and developers, and a well designed query

language or API can bring great helps in both data model definition and development and maintenance time.

2.5.2 Differences from graph computing engines

There are two categories of graph data systems: *graph databases* and *graph computing engines*.

- **Graph databases** are the ones presented until now, i.e. technologies used primarily for transactional online graph persistence and typically accessed directly in real time from an application. They are the equivalent of "normal" online transactional processing (OLTP) databases in the relational world.
- **Graph computing engines** (or *graph analytics engines*) are technologies used primarily for offline graph analytics, typically performed as a series of batch steps. They can be thought of as being in the same category as other technologies for analysis of data in bulk, such as data mining and online analytical processing (OLAP). They thus typically works on a global graph level and take advantage of multiple machines for the batch works to be executed [7, 48].

Graph computing engines are designed to do things like the identification of clusters on data; or to answer questions such as: "how many relationships, on average, does anyone in a social network have?".

Whereas some graph analytics engines include a graph storage layer, others (and arguably most) concern themselves strictly with processing data that is fed in from an external source, and then returning the results for storage elsewhere [48].

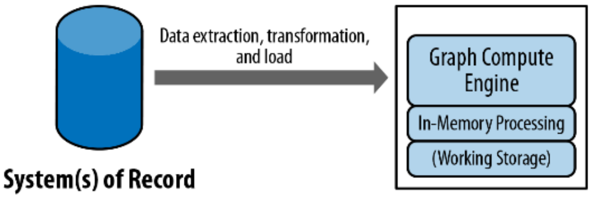


Figure 2.17: A high-level view of a typical graph analytics engine deployment [48].

Figure 2.17 shows a common architecture for deploying a graph analytics engine. The architecture includes a system of record (SOR) database with OLTP properties (such as some relational or document or graph DBMSs), which services requests and responds to queries from the application (and ultimately the users) at runtime. Periodically, an ETL (Extract Transform and Load) job moves data from the system of record database into the graph analytics engine for offline querying and analysis [48].

We now give the main reasons why graph databases are not the right instrument for graph analytics operations. The processing of graphs generally entails mostly random data access. For large graphs that cannot be stored in memory, random disk access becomes a performance bottleneck. However, even in the case of smaller graphs that could be entirely kept in main memory, since graph databases are centralized systems that have to assure ACID properties, they automatically lack the computational power of a distributed and parallel system. So when some computations have to be executed on the entire graph in a single-thread approach, it immediately emerges that computation times become prohibitive.

One option for realizing graph analytics with graph databases would be the realization of batch processes that perform multiple queries whit graph-local scope on the database; and later perform some computations on the intermediate results in order to obtain the graph global result. However, when distributed computation is required, there already exist some products that manage such operations.

The first and most known framework that allows to do this is Apache **Hadoop**, an open source distributed-processing framework for large data sets that includes a MapReduce implementation. With Hadoop and MapReduce, commodity computer clusters can be programmed to perform large-scale data processing in a single pass [115].

In the **MapReduce** programming model - introduced by Google in 2004 - the Map function takes key/value pairs as input and produces a set of intermediate key/value pairs. The framework groups all intermediate values that are associated with the same intermediate key and passes them to the Reduce function. The Reduce function receives an intermediate key with its set of values and merges them together. Unlike graph databases, MapReduce is not designed to support online query processing. MapReduce is optimized for analytics on large data volumes partitioned over hundreds of machines [115, 52].

However, Hadoop was thought for working with a big amount of files organized on the Hadoop Distributed File System (**HDFS**); and it was not designed mainly to support scalable processing above graph-structured data.

The main limitation in the MapReduce cluster computing paradigm is that it forces a particular linear dataflow structure on distributed programs: it reads input data from disk; maps a function across the data; reduces the results of the map; and stores reduction results on disk. This massive use of disks for storing the elaboration results becomes a bottleneck when iterative algorithms are deployed [152].

In 2010, Google introduced the **Pregel** system as a scalable platform for implementing graph algorithms. In particular, graphs are inherently recursive data structures as properties of vertices depend on properties of their neighbors which in turn depend on properties of their neighbors. As a consequence many important graph algorithms iteratively recompute the properties of each vertex until a fixed-point condition is reached. The Pregel system targeted such problematics [120].

GraphX is a distributed graph processing framework on top of Apache Spark. It provides two separate APIs for implementation of massively parallel algorithms (such as PageRank): a Pregel abstraction, and a more general MapReduce style API. GraphX has full support for property graphs and it is part of Apache Spark [152].

Apache **Spark** provides programmers with an application programming interface centered on a data structure called the resilient distributed dataset (RDD), a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way. In particular, Spark's RDDs function as a working set for distributed programs that offers a (deliberately) restricted form of distributed shared memory [152].

The availability of RDDs facilitates the implementation of both iterative algorithms, that visit their dataset multiple times in a loop, and interactive/exploratory data analysis, i.e., the repeated database-style querying of data. The latency of such applications (compared to Apache Hadoop, a popular MapReduce implementation) may be reduced by several orders of magnitude [152].

In this work we will only analyze some of the products belonging to the former category: graph databases.

2.5.3 Differences from the relational database

Relational databases were initially designed to encode tabular structures, and in this they work exceedingly well. However they could struggle when attempting to model the web of relationships that arise in different domains of the real world.

One of the stronger arguments of graph databases is that "*Relationships are first-class citizens of the graph data model*". Graph databases indeed treat both vertexes and relationships with full importance, storing them with dedicated constructs. The reason for it is that relationships are the fundamental descriptors for the web of connections among the elements stored, and they are the main information carriers for all applications that need to work with graph data.

In other database management systems - and in particular in the relational ones - this does not hold: there are not dedicated constructs for treating relationship information differently from other forms of information, and the connections among entities must be inferred by performing join operations on values of the fields with referential integrity constraints (like foreign keys); or by out-of-band processing such as MapReduce [48].

The relational model is by definition "based on values": relationships among two different relations are expressed by means of the values that appear within the *tuples* of both relations [41]. Graph databases instead do not need to check and link entities by comparing the values of a given property residing in both of them; they have the linking information already stored, and all the operations that are normally executed at query time by relational databases or document databases are completely avoided.

The way by which relationships are stored and linked to the nodes involved depends on the database implementation. However, if thinking about those that realize index-free adjacency, the fundamental idea is that the node "record" directly stores a "pointer" (either logical or physical) to the linked nodes; or it stores a pointer to some relationship objects that store the information of the other node involved, together with any other possible property.

By directly storing relationships, the database lightens the system by those operations that can become heavy - in particular with large amount of data - like JOINS, multiple searches by indexes, temporary table, etc. Suppose we are searching for the friends information of those social network users that have a specific value for one of their information fields (e.g. zip code). In a relational database, modeled by a relation of users and a relation of friendships, the first operation would be a query on the particular field (possibly indexed) for the searched value; then all the values extracted, belonging to the primary key of the user record, would be used to search on the friendships relation all the primary keys for the friends, which by means of a JOIN, would be used to extract and display all the information of the users found.

For a graph database, instead, the first search on the field is obviously performed; but after that, all friends can be found by directly following the relationships attached to the node; without needs of JOINs.

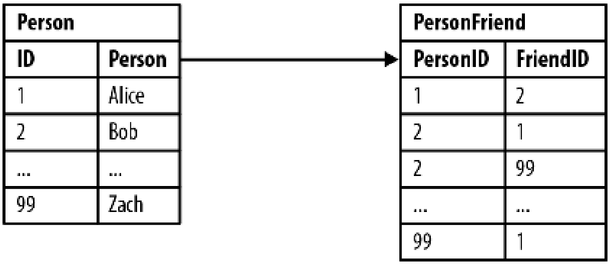
Relationships are not treated with full importance only at the storage layer; rather also at the conceptual layer (how data shows to be modeled). This fact brings another important point for graph databases: the dedicated and simplified way of handling relationships in all the operations that the user (or the application) wants to perform on the underlying database. By making the relationship manageable as a stand-alone object (thus not indirectly identified as a secondary object by the two vertexes connected by that edge) gives simplicity in its manipulation for what regards: the update of its properties; the removal or creation of new edges; and its interrogation.

All these aspects are seen as lacks of the relational model when one tries to work with graph data: relationships are not directly identifiable; they must be computed at query time; and also there are not easy ways for expressing queries that run on the underlying graph data. This does not mean the relational model is badly designed, but because it is designed for other kinds of data domains; and when one tries to force a graph domain in it, some difficulties emerge.

The other strong argument of graph databases is the following: "*The true value of the graph approach becomes evident when one performs searches that are more than one level deep.*" What does it means in a relational solution? It means that an entity has a recursive relationship with itself (for example, for representing friendship relations among the users of a social network). In that situation, it may be possible to found a chain of relationships that involves some of the records of the same entity. For example, friends of John's friends can be found.

Not in all cases, when there is a recursive relationship, there also is some graph-structured data in the database: it depends on the cardinality constraints specified on the branches of the recursive relationship. In particular, if the maximum cardinality value is 1 for both of them, then there is no problem: each record can have at most only one relationship with another record, and thus only flatten chains - maybe longer than 1 relationship - are possible. When one branch has maximum cardinality to 1 and the other has N, then the underlying data can have the form of a tree. In fact, each record can have relationships with more then one record attached to it with the role of "children", but can have only one record attached to it with the role of "father". It is only when both cardinality constraints have maximum value equal to N, however, that we may be working with some graph data. Each record can have multiple "children" and also multiple "fathers".

Let's now consider a problem when a relational database is used for a social network dataset. Figure 2.18 shows both the relations involved for the representation of such domain. There is a table holding Person records and a table holding the friendships among Person records.



Person	
ID	Person
1	Alice
2	Bob
...	...
99	Zach

PersonFriend	
PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

Figure 2.18: Modeling friends and friends-of-friends in a relational database [48].

If we ask "who are the friends of Bob's friends?", the question is quite easy, but its formulation is not so straightforward by using SQL. In fact, hierarchies in SQL use multiple joins, which make the query syntactically and computationally more complex.

The way in which SQL allows us to perform hierarchical queries, indeed, is typically by means of nested JOINs. In particular, when asking "two levels deep" in a recursive relationship, the query consists of a JOIN inside the FROM clause of an outer SELECT clause where another JOIN is performed. This implies that, for each desired depth level, one should prepare a query made of how many nested SELECT + JOIN statements how many are the desired depth levels to be traversed. For example, asking for "Who are Alice's friends-of-friends" involves four nested SELECT statements: one for searching Alice and its ID; on for searching her friends' IDs (first JOIN); one for searching the friends' IDs of her friends (second JOIN); and the last for retrieving the names of the friends-of-friends (third JOIN).

So things get more complex and more expensive the deeper we go into the network. Though it is possible to get an answer to the question "who are my friends-of-friends-of-friends?" in a reasonable period of time, queries that extend to four, five, or six degrees of friendship deteriorate significantly due to the computational and space complexity of recursively joining tables [48].

The way provided by most relational databases to overcome this situation made of nested

JOINs on sub-queries and with the depth level hard-written within the query itself by means of the same number of nested statements, is by using a special kind of CTE (Common Table Expression) that is invoked by the *WITH RECURSIVE* clause. In Section 3.2.4 we will give additional details about it.

This will be the way by which we will realize queries for our relational database.

One last difference between the relational and the graph model is that in graph databases the separation between schema and data (instances) is less marked than in the relational model [7]. Graph databases indeed manage to model a connected reality in a more direct and natural way: the transformation between the "conceptual schema" and the "logical schema", belonging to the typical database development process, is typically very little.

For a relational database, instead, the transformation is sometimes less straightforward, also because of the normalization steps that have to be applied to the ER schema in order to map it to a relational "logical schema". In addition, such normalization steps often force to slowly go far from the reality we want to represent, and also sometimes degrade performance. This is also the reason why de-normalization steps are sometimes applied in a successive moment.

Such quicker transition from conceptual to logical schema brings to less impedance between the analysis project phase and the implementation phase.

2.5.4 Differences from document stores

Not only relational databases may struggle when working with graph data. Most NOSQL databases indeed store sets of disconnected documents, values or columns. This makes it difficult to use them for connected data and graphs.

Let's consider the case for document stores.

The simplest way for encoding the presence of a relationship between two document records is by embedding, within the outer document, the document which is pointed by the other one. Hierarchical situations can thus be represented with this approach; however it immediately display the problem that if more documents point to the same document record, then such record would be replicated for both the outer documents; with all the bad side effects that come because of this.

The other well-known strategy for adding relationships to such stores is to embed a record identifier to the other record, instead of embedding it entirely. This approach resolves the duplication of records and foreign keys may be applied. However, this second approach requires joining records at the application level because it has to build relationships from these flat, disconnected data structures; which quickly becomes expensive. Furthermore, some consistency mechanisms have to be implemented to ensure that the application updates or deletes these foreign record references in tandem with the rest of the data. If this doesn't happen, the store will accumulate dangling references, which can harm data quality and query performance [48].

Another weak point of this solution is this: because there are no identifiers that "point backward" (the foreign "links" are not reflexive), the ability to run queries that go to the opposite direction is lost.

Let's see it with an example (Figure 2.19). Note that here we consider friendship relationships asymmetrical, thus more similar to a "is_following" kind of relationship. However, this only affects the relationship name: the links realized with the second approach are inherently asymmetrical.

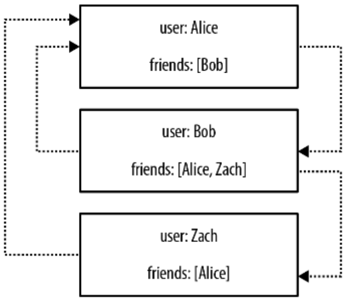


Figure 2.19: Social network example encoded in a document store [48].

With this structure, it is easy to find "who are Bob's immediate friends?", because references to such records are collected by Bob itself. However, if we would like to ask the specular query, i.e. "who is friends with Bob?", the question would be more difficult to get answered. In this case the only option would be to perform a brute-force scan across the whole dataset looking for friends entries that contain a reference to Bob [48].

2.6 General approach to process graph queries

When working with graphs, one does have to change its way of thinking about queries; especially for those who come from SQL and the relational model (i.e. almost everyone). Differently from queries on relational databases, expressed and thought in a "SQL form" (and thus involving reasoning with tables, columns, joins, etc.), here the main focus is on vertices and how to move upon the graph by following the right edges.

Graph queries can conceptually be distinguished in two components: the *landing phase*, and the *exploration phase*.

The **landing phase** searches for the vertex on the graph by which the query is anchored, and it usually employs indices, labels/classes and properties for vertex individuation. This vertex will represent the starting point for the following exploration phase.

However, the landing phase does not only work with a single vertex. More vertices can be pointed and then highlighted on the graph; and the same goes for relationships.

The landing phase can represent the most costly operation on large graphs. It depends in fact on the number of vertices belonging to the category of search; on the indexes defined; and on the statistics of attributes' values. However, for complex queries, it could represent only a fraction of the total query time, due to a complex exploration phase.

The **exploration phase** is what follows the individuation of all the elements that can be targeted by the parameters given with the query; and it is about graph navigation (or better traversal). One example is the search for all the neighbors of the initial vertex, and maybe only reached by outgoing edges; or again the search for paths that go from a start node to a target node, if they exist.

The exploration phase thus embeds the graph navigation algorithms of graph theory, and collide with its known problems.

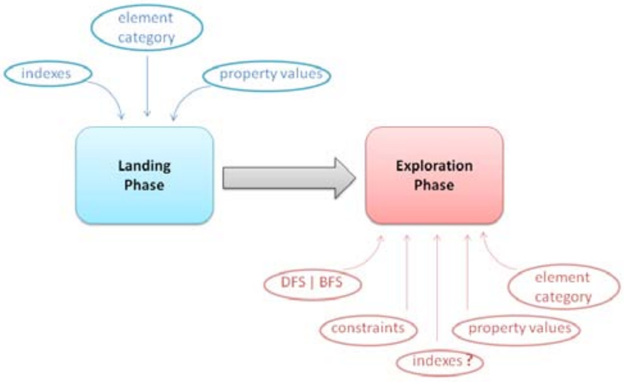


Figure 2.20: Graph query phases.

In Figure 2.20 is represented the graph query divided in the two components. In particular, we highlight that the first phase typically works by exploiting the information about the category of the anchor element searched, the values for some of its properties, and the eventual use of some indexes. The second phase, which follows the first one, also works by exploiting the defined category of the elements searched, or the allowed values for some of their properties; however, it also takes the information about what should be the exploration order (DFS or BFS), and what should be the constraints defined for the exploration, which will be presented in Chapter 6. In addition, it is highlighted that such phase may or not be based on indexes. Those graph databases that support the *index-free adjacency* property will not use any index.

Regarding the effective distinction of the two phases in query formulation, some of the graph query languages permit a quite clear individuation of the elements targeting the two phases;

others instead do not make a so highlighted distinction within the syntax of the query, but the query execution follows the same philosophy.

Chapter 3

DBMSs comparison

In this chapter we will give some general information about the graph databases currently available. In addition, we will present a detailed overview for each of the three graph DBMSs chosen for the study. We will also give a quick presentation for a relational database product, in order to compare them with the relational model.

In Section 3.1 we then present the most known graph databases of the moment. In Section 3.2 we compare the four DBMSs. In Section 3.3 we present a feature matrix that quickly show the main differences among them.

3.1 Which are the major Graph DBMSs

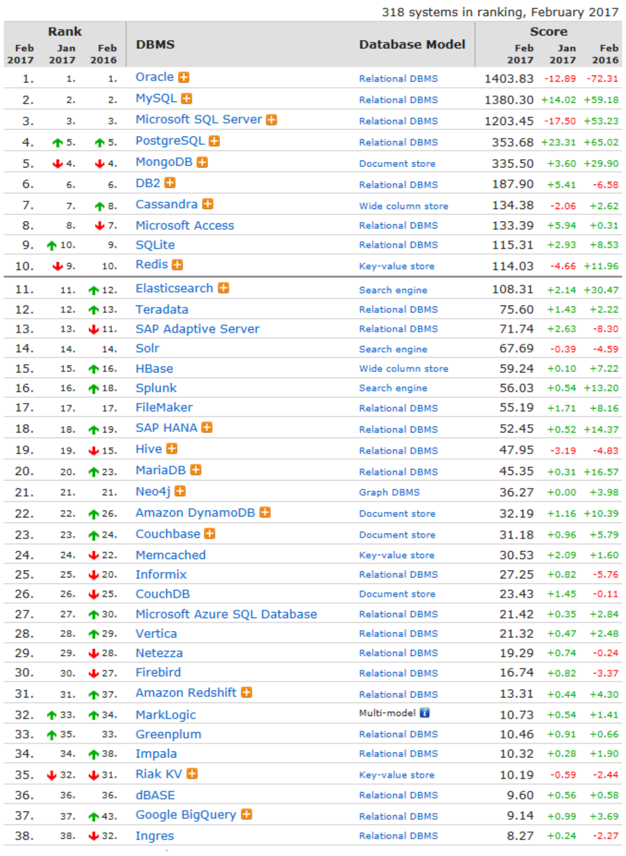
On the website www.DB-ENGINES.com a list of the most known existing DBMSs is provided. These DBMSs are ranked by their popularity and the list is updated monthly.

In Figure 3.1 it is shown the ranked list of all the DBMSs tracked, while in Figure 3.2 it is shown the trending chart for some of them; in particular, we selected the leading relational DBMSs and some other graph databases. In Figure 3.3, instead, it is shown the ranking list for only the graph DBMSs tracked.

As it can be seen, relational databases are the dominant products within the field; furthermore, graph databases are instead ranked well under the first positions. The first ranked graph DBMS, which is Neo4j, is the 21st of the overall databases ranking list.

The popularity measure, computed for each DBMS, is obtained by using the following parameters:

- Number of mentions of the system on websites, measured as number of results in search engines queries.



318 systems in ranking, February 2017

Rank			DBMS	Database Model	Score		
Feb 2017	Jan 2017	Feb 2016			Feb 2017	Jan 2017	Feb 2016
1.	1.	1.	Oracle	Relational DBMS	1403.83	-12.89	-72.31
2.	2.	2.	MySQL	Relational DBMS	1380.30	+14.02	+59.18
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1203.45	-17.50	+53.23
4.	↑ 5.	↑ 5.	PostgreSQL	Relational DBMS	353.68	+23.31	+65.02
5.	↓ 4.	↓ 4.	MongoDB	Document store	335.50	+3.60	+29.90
6.	6.	6.	DB2	Relational DBMS	187.90	+5.41	-6.58
7.	7.	↑ 8.	Cassandra	Wide column store	134.38	-2.06	+2.62
8.	8.	↓ 7.	Microsoft Access	Relational DBMS	133.39	+5.94	+0.31
9.	↑ 10.	9.	SQLite	Relational DBMS	115.31	+2.93	+8.53
10.	↓ 9.	10.	Redis	Key-value store	114.03	-4.66	+11.96
11.	11.	↑ 12.	Elasticsearch	Search engine	108.31	+2.14	+30.47
12.	12.	↑ 13.	Teradata	Relational DBMS	75.60	+1.43	+2.22
13.	13.	↓ 11.	SAP Adaptive Server	Relational DBMS	71.74	+2.63	-8.30
14.	14.	14.	Solr	Search engine	67.69	-0.39	-4.59
15.	15.	↑ 16.	HBase	Wide column store	59.24	+0.10	+7.22
16.	16.	↑ 18.	Splunk	Search engine	56.03	+0.54	+13.20
17.	17.	17.	FileMaker	Relational DBMS	55.19	+1.71	+8.16
18.	18.	↑ 19.	SAP HANA	Relational DBMS	52.45	+0.52	+14.37
19.	19.	↓ 15.	Hive	Relational DBMS	47.95	-3.19	-4.83
20.	20.	↑ 23.	MariaDB	Relational DBMS	45.35	+0.31	+16.57
21.	21.	21.	Neo4j	Graph DBMS	36.27	+0.00	+3.98
22.	22.	↑ 26.	Amazon DynamoDB	Document store	32.19	+1.16	+10.39
23.	23.	↑ 24.	Couchbase	Document store	31.18	+0.96	+5.79
24.	24.	↓ 22.	Memcached	Key-value store	30.53	+2.09	+1.60
25.	25.	↓ 20.	Informix	Relational DBMS	27.25	+0.82	-5.76
26.	26.	↓ 25.	CouchDB	Document store	23.43	+1.45	-0.11
27.	27.	↑ 30.	Microsoft Azure SQL Database	Relational DBMS	21.42	+0.35	+2.84
28.	28.	↑ 29.	Vertica	Relational DBMS	21.32	+0.47	+2.48
29.	29.	↓ 28.	Netezza	Relational DBMS	19.29	+0.74	-0.24
30.	30.	↓ 27.	Firebird	Relational DBMS	16.74	+0.82	-3.37
31.	31.	↑ 37.	Amazon Redshift	Relational DBMS	13.31	+0.44	+4.30
32.	↑ 33.	↑ 34.	MarkLogic	Multi-model	10.73	+0.54	+1.41
33.	↑ 35.	33.	Greenplum	Relational DBMS	10.46	+0.91	+0.66
34.	34.	↑ 38.	Impala	Relational DBMS	10.32	+0.28	+1.90
35.	↓ 32.	↓ 31.	Riak KV	Key-value store	10.19	-0.59	-2.44
36.	36.	36.	dBASE	Relational DBMS	9.60	+0.56	+0.58
37.	37.	↑ 43.	Google BigQuery	Relational DBMS	9.14	+0.99	+3.69
38.	38.	↓ 32.	Ingres	Relational DBMS	8.27	+0.24	-2.27

Figure 3.1: Ranking list of some of the DBMSs tracked by DB-ENGINES.com [1].

- General interest in the system, measured as frequency of searches in Google Trends.
- Frequency of technical discussions about the system, measured as the number of related questions and the number of interested users on the well-known IT-related Q&A sites Stack Overflow and DBA Stack Exchange.
- Number of job offers, in which the system is mentioned, on the leading job search engines Indeed and Simply Hired.
- Number of profiles in professional networks, in which the system is mentioned, on LinkedIn and Upwork.
- Relevance in social networks, measured as the number of Twitter tweets in which the system is mentioned.

DB-Engines calculates the popularity value of a system by standardizing and averaging the

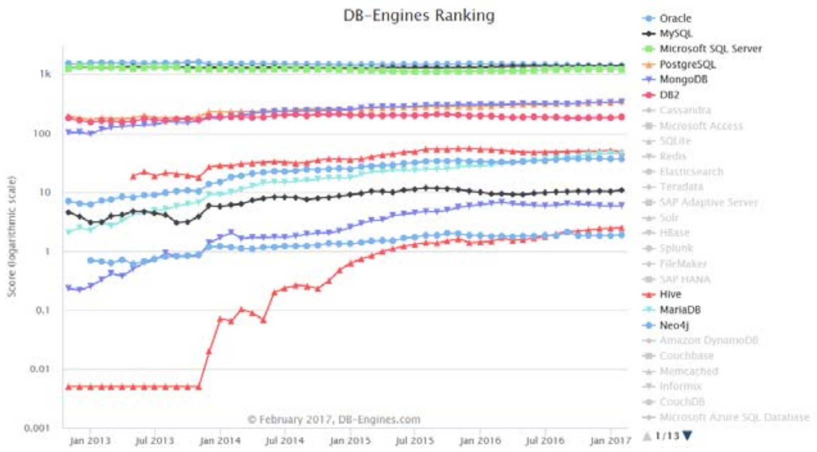


Figure 3.2: Trending chart of some of the DBMSs tracked by DB-ENGINES.com [1].

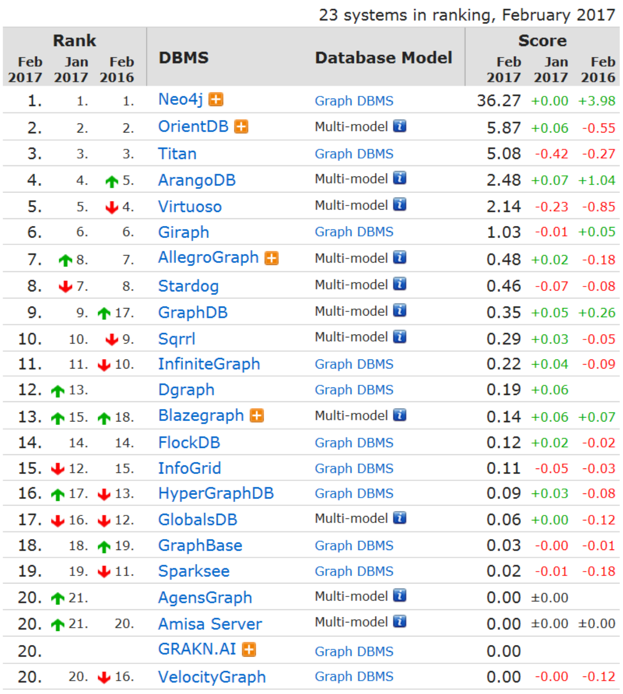
individual parameters. These mathematical transformations are made in a way so that the distance of the individual systems is preserved. That means, when system A has twice as large a value in the DB-Engines Ranking as system B, then it is twice as popular when averaged over the individual evaluation criteria [1].

The website has been useful for an initial analysis on which and how many are the DBMSs that support the graph data model.It has been also useful for getting a rough idea on the size of their relevance and presence on the databases' field; as well as to guide the choice on the three sample graph DBMSs that will be compared in this work.

As you can see in Figure 3.3, the five most popular graph DBMSs are Neo4j, OrientDB, Titan, Virtuoso and ArangoDB.

Thanks to the chart in Figure 3.4 it can be inferred that Neo4j is stable (and still growing in popularity) on first position since time; while OrientDB and Titan are quite similar for behaviour and growing pace; Virtuoso seems to be in a flat period; and ArangoDB, that's born quite recently, gained a rapid enhancement in its popularity.

Note, however, that **popularity** should not be confused with **adoption**. One may think that popularity and adoption grow in parallel and proportionally with each other; however this is not true. There is indeed a disclaimer on DB-Engines.com related to the topic: *"The DB-Engines Ranking does not measure the number of installations of the systems, or their use within IT systems. It can be expected, that an increase of the popularity of a system as measured by the DB-Engines Ranking (e.g. in discussions or job offers) precedes a corresponding broad use of the system by a certain time factor. Because of this, the DB-Engines Ranking can act as an early indicator"* [1].



23 systems in ranking, February 2017

Rank			DBMS	Database Model	Score		
Feb 2017	Jan 2017	Feb 2016			Feb 2017	Jan 2017	Feb 2016
1.	1.	1.	Neo4j	Graph DBMS	36.27	+0.00	+3.98
2.	2.	2.	OrientDB	Multi-model	5.87	+0.06	-0.55
3.	3.	3.	Titan	Graph DBMS	5.08	-0.42	-0.27
4.	4.	↑ 5.	ArangoDB	Multi-model	2.48	+0.07	+1.04
5.	5.	↓ 4.	Virtuoso	Multi-model	2.14	-0.23	-0.85
6.	6.	6.	Giraph	Graph DBMS	1.03	-0.01	+0.05
7.	↑ 8.	7.	AllegroGraph	Multi-model	0.48	+0.02	-0.18
8.	↓ 7.	8.	Stardog	Multi-model	0.46	-0.07	-0.08
9.	9.	↑ 17.	GraphDB	Multi-model	0.35	+0.05	+0.26
10.	10.	↓ 9.	Sqrrl	Multi-model	0.29	+0.03	-0.05
11.	11.	↓ 10.	InfiniteGraph	Graph DBMS	0.22	+0.04	-0.09
12.	↑ 13.		Dgraph	Graph DBMS	0.19	+0.06	
13.	↑ 15.	↑ 18.	Blazegraph	Multi-model	0.14	+0.06	+0.07
14.	14.	14.	FlockDB	Graph DBMS	0.12	+0.02	-0.02
15.	↓ 12.	15.	InfoGrid	Graph DBMS	0.11	-0.05	-0.03
16.	↑ 17.	↓ 13.	HyperGraphDB	Graph DBMS	0.09	+0.03	-0.08
17.	↓ 16.	↓ 12.	GlobalsDB	Multi-model	0.06	+0.00	-0.12
18.	18.	↑ 19.	GraphBase	Graph DBMS	0.03	-0.00	-0.01
19.	19.	↓ 11.	Sparksee	Graph DBMS	0.02	-0.01	-0.18
20.	↑ 21.		AgensGraph	Multi-model	0.00	±0.00	
20.	↑ 21.	20.	Amisa Server	Multi-model	0.00	±0.00	±0.00
20.			GRAKN.AI	Graph DBMS	0.00		
20.	20.	↓ 16.	VelocityGraph	Graph DBMS	0.00	-0.00	-0.12

Figure 3.3: Ranking list of the graph DBMSs tracked by DB-ENGINES.com [1].

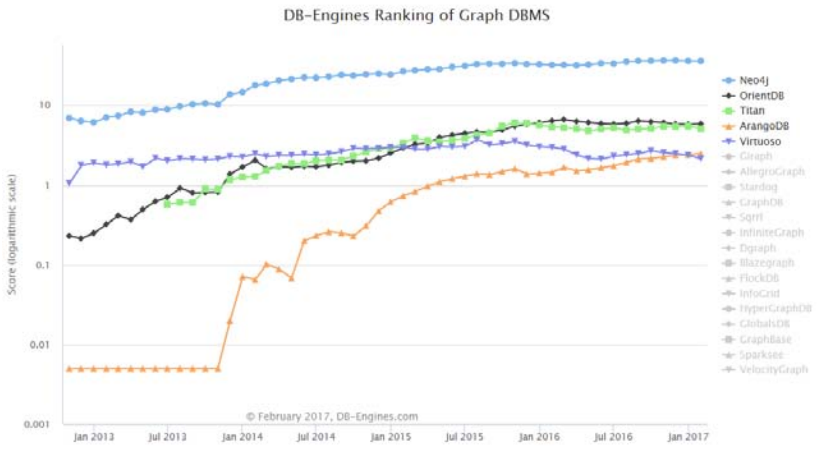


Figure 3.4: Trending chart of the top 5 graph DBMSs tracked by DB-ENGINES.com [1].

3.2 The compared DBMSs

In this section we will examine three of the most known graph DBMSs that support the property graph model, which are: ArangoDB, Neo4j and OrientDB. In particular, we will present how the examined DBMSs face some of the typical databases' problems; with special attention on some of the topics that may be interesting because of the graph nature of the data domain, like: storage techniques, caching, querying, scalability, etc. The same comparison will be also done for PostgreSQL.

For each of them, during our test presented in Chapter 5 we will use the free "community editions"; so the following comparison will mainly be based on such product version. The "enterprise editions" typically come with additional features or enhancements in the database configuration and administration; or with other minor tools regarding what stands around a DBMS. Furthermore, Enterprise Edition licenses are often more about support subscriptions than "turned upside down" DBMS engines.

3.2.1 ArangoDB

ArangoDB is a NoSQL multi-model database management system supporting graphs, key/value pairs and documents. It is realized by ArangoDB GmbH and triAGENS GmbH (Germany) and its first release was in 2012.

According to DB-Engines.com, at the time of writing, it is ranked 4° in Graph DBMS category, 15° in Document Stores category, 12° in Key-Value Stores category, and 79° overall (over all the DBMS tracked by the site).

It is a schema-free DBMS, implemented in C/C++ and JavaScript, and supports several operating systems (Linux, OS X, Windows, Raspbian and Solaris). ArangoDB is designed to serve documents to clients; these documents are transported in JSON format using the HTTP protocol; so a REST API is provided to interact with the database system. The database can also be accessed by means of a web interface and an interactive shell.

It supports several programming language drivers (C#, Clojure, Java, JavaScript, PHP, Phyton, Ruby, Go, etc) and allows to define stored procedures in JavaScript.

It states to have a native multi model approach, and not, for example, a graph database realized as an abstraction layer on top of a document store. In this way "it "does not switch between the models behind the scenes in order to execute queries [26].

ArangoDB represents the "*emergent*" figure in this work and also in the graph databases field: it is one of the youngest commercial products that realize such model, and it also brings added value since it is multi-model. It aims to compete with graph databases as well as document or key/value store.

During this work, we will use the 3.0.10 version released in 26 September 2016. However, given the fact that version 3.1 has been released during the development of the work, some hints on new interesting features will be pointed out now and then.

Logical data organization

ArangoDB organizes data in terms of *databases*, *collections* and *documents*.

Databases are sets of collections, which are the sets that collect records, which in turn are also called documents. Multiple databases can be defined in order to have isolation among collections. There always is a special database, called "_system", created by default and that cannot be removed, which is used as the administration database in order to perform operations like users and collections management.

Collections are the equivalent of tables in RDBMS, and documents can be thought of as rows in a table. Being schema-less, there is no need to define what attributes a collection - and thus its documents - can have before inserting data in them; rather every single document can have a completely different structure and still be stored together with other documents in a single collection [16, 15]. There are two types of collections: document collection (also refered to as **vertex** collections in the context of graphs) as well as **edge** collections. Edge collections store documents as well, but they include two special attributes, *_from* and *_to*, which are used to create relationships between two vertex documents.

Documents in ArangoDB follow the JSON format, although they are stored in a binary format, which is called *VelocyPack* (we will see some details later in Section 3.2.1). A document contains zero or more attributes, each of these attributes having a value. A value can either be an atomic type, i. e. number, string, boolean or null, or a compound type, i.e. an array or embedded document; so that arbitrarily nested data structures can be represented in a single document. Each document has a unique primary key which identifies it within its collection and, in general, across all collections in the same database. The combination of the document key and the collection name forms what is called the *document handle*.

All documents contain three special attributes: the document handle is stored as a string in *_id*, the document's primary key in *_key* and the document revision in *_rev*. The value of the *_key* attribute can be specified by the user when creating a document; however *_id* and *_key* values are immutable once the document has been created, while *_rev* value is maintained by ArangoDB automatically [15].

ArangoDB also defines the "named graph", which is the way by which ArangoDB handles graphs. A named graph is created by specifying the edge collections to be used for the construction of the graph; the involved vertex documents will be automatically detected by the pointers contained in the edge documents. Such operation for the graph creation is termed as “edges definition” [17].

The edge direction is of course given by the two fields *_from* and *_to*. Within queries it is possible to define in which direction the edge should be followed, which are OUTBOUND: $\_from \rightarrow \_to$; INBOUND: $\_from \leftarrow \_to$; and ANY: $\_from \leftrightarrow \_to$ [17].

Physical data organization

ArangoDB uses JSON as its default data format [151]. It can natively store a nested JSON object as a data entry inside a collection; therefore, there is no need to disassemble the resulting JSON objects for their storage, and thus the stored data simply inherits the tree structure of such document [5].

However, internally ArangoDB uses VelocyPack. VelocyPack is a compact binary format for serialization and storage of documents, query results and temporarily computed values [23]. More specifically, VelocyPack is an (unsigned) byte oriented serialization format, where its values (not necessarily aligned) are simply platform independent sequences of bytes [24].

Its primary goal is to reduce storage space requirements for “small” values (such as boolean, integers, short strings) in order to speed up several operations inside queries. VelocyPack document entries stored on disk are self-contained, in the sense that each stored document will contain all of its data type and attribute name descriptions. While this may require a bit more space for storing the documents, it removes the overhead of fetching attribute names and document layout from shared structures. It also simplifies the code paths for storing and reading documents [12].

The arguments of why ArangoDB developed this data format are as follows:

These days, JSON (JavaScript Object Notation) is used in many cases where data has to be exchanged. Lots of protocols between different services use it, databases store JSON (document stores naturally, but others increasingly as well). It is popular, because it is simple, human-readable, and yet surprisingly versatile, despite its limitations.

ArangoDB developed this format because none of the several known JSON formats used by other applications (e.g. Universal Binary JSON, MongoDB's BSON, MessagePack, BJSON, Apache Thrift, Google's Protocol Buffers, etc.) manages to combine compactness, platform independence, fast access to subobjects and rapid conversion from and to JSON [23].

The way by which ArangoDB stores a graph model is simply by using these particular types of JSON documents (which are of vertex or edge type) that are stored with the optimized VelocyPack format. There is not, thus, a particular data structure which models a graph (as it is done by Neo4j, for example); rather a specialized utilization of the JSON format.

The relationships between the edge elements and their referenced vertex elements are reconstructed by means of indexes constructed over them at server boot. These indexes are called

Edge indexes and provide quick access to documents by either their *_from* or *_to* attributes.

Internally, the Edge Index is implemented as a hash index, which stores the union of all *_from* and *_to* attributes. They are thus used every time an edge is taken in consideration by the traversal operation, and point to where the information relative to the vertexes are stored.

It is now clear that ArangoDB does not realize the index-free adjacency property. In fact, an index lookup is needed to get from a node to an edge and vice versa. However ArangoDB uses for this operations a hash index; and the time for the lookup is O(1). So, up to a very small constant factor, the time to get from a node to the edge (and vice versa) is not so different from as if the address of the edge were directly stored in the node itself, maybe as a property, and serialized to secondary memory [2].

During this work a new version of ArangoDB was released: the 3.1. It includes an additional possibility for optimizing graph traversals, which is *Vertex centric Indexes*.

Its basic idea is to index a combination of a vertex, the direction of the connected edge, and any arbitrary set of other attributes on the edges. For example, consider a social network situation where there exist different types of relationship among users, like *friend_of* or *follows* etc. In this situation, we will have an attribute called *Type* on the edges. Using the built-in Edge Index, ArangoDB can find the list of all edges attached to the vertex fast, but it still has to walk through this result list and check if all of them have the attribute *Type* == "friend_of". Using a vertex-centric index would allow ArangoDB to find all edges for the vertex having the attribute *Type* == "friend_of" in the same time; and thus avoiding the necessity to verify the condition on all the resulting edges [10].

Documents are stored to disk by means of memory-mapped files. By default setting, these memory-mapped files are synced regularly to disk, but not instantly. This comes as a trade-off between storage performance and data durability. If this level of durability is too low for an application, the server can also sync all modifications to disk instantly; this will give full durability but will come with a performance penalty because each data modification will trigger a sync I/O operation [14].

In addition, instead of overwriting existing documents, ArangoDB creates a new version for each modified document (MVCC - Multi-Version Concurrency Control); and this is also the case when a document gets deleted. The two benefits are that objects can be stored coherently and compactly in the main memory; and that isolated writing and reading transactions allow accessing these objects for parallel operations.

Data Integrity

Data integrity is certainly one of the most important aspects of a database; it is even more important in a scenario where complex data models are involved, just like here. In this section we will talk about both data integrity and "*graph integrity*", where with graph integrity we intend a consistent state for its edges.

For what regards data constraints ArangoDB, being a NoSQL schema-less DBMS, does not expect a schema to be declared before data insertion, so data constraints are typically not imposed. We may have documents with different arguments within the same collection, and also edges within the same edge collection that point to more and different vertex collections. However, automatic indexes on system attributes (like _key, or _from and _to) assure unique constraint; which can be similarly imposed on other fields by the user/application by creating indexes on them.

The moments in which graph (and in general data) integrity could fall is in correspondence of insertion, update and delete operations.

After a named graph is created, its defining collections are still accessible by using the standard methods; thus classical insert/update/delete operations are still possible in the classic way, and graph inconsistency may arise. In fact, the deletion of a vertex should not be done so airily, because it could bring to dangling edges. However, if these collections are accessed by functions of the graph module, the following guarantees are assured:

- all modifications are executed transactional;
- if a vertex is deleted, all connected edges will be deleted; thus avoiding loose ends;
- if an edge is inserted, it is checked that such edge matches the edge definitions, so that edge collections will only contain valid edges [17].

Since ArangoDB's named graphs are graph definitions done on collections that can live independently of the graph, it could be also possible that the same vertex collection is used by two different named graphs at the same time. However, the graph module manages also this situation and dangling edges will be avoided even in this case. This, of course, involves more operations inside the database which obviously don't come for free [17].

Talking instead of data consistency and concurrent operations, the way by which data integrity is preserved is by using transactions. ArangoDB transactions are different from transactions in SQL. In particular, a SQL transaction starts with explicit commands (like BEGIN or START TRANSACTION); then it follows a series of data retrieval or modification operations; and it ends with a COMMIT command, or rolls-back with a ROLLBACK command.

For ArangoDB transactions there are no individual BEGIN, COMMIT or ROLLBACK transaction commands; instead, a transaction is started by providing a description of the transaction to the db._executeTransaction() JavaScript function. This function will then automatically start a transaction, execute all required data retrieval and/or modification operations, and at the end automatically commit the transaction. If an error occurs during transaction execution, the transaction is automatically aborted, and all changes are rolled back [20]. In addition, a transaction is always a server-side operation, and is executed on the server in one go, without any client interaction.

During transaction execution, however, ACID properties will be given by exploiting some techniques like document revision; collections locking for those involved by the transaction;

transactions interruptions disabled; etc. [20].

Transactions lean on WAL (Write-Ahead Log) files, i.e. files where all modifications are appended before they are applied and persisted to disk. This approach permits to just search for a file's valid start-section after a failures or server crashes. If instead of doing this, the DBMS would overwrite existing data, it would have to verify each block's validity before allowing the database to be accessible, like other databases typically do. So this approach is used to run data recovery after a server crash, and can also be used in a replication setup when slaves need to replay the same sequence of operations as on the master [14]. However, the assumption at the basis of this approach is that the server is subjected to few crashes [40].

One limitation of ArangoDB transactions is that they cannot be invoked by AQL. Another limitation is that a transaction operation information (record pointers, revision numbers and rollback information) must fit into main memory. In addition, to ensure progress of the Write-Ahead Log garbage collection, transactions should be kept as small as possible, and big transactions should be split into multiple smaller transactions. However, transactions in ArangoDB cannot be nested, and if an attempt is made to call a transaction from inside a running transaction the server will throw an error.

Transactions vary their guarantees depending on the database structuring. Using a single database instance, multi-document and multi-collection queries are guaranteed to be fully ACID; and also in cluster mode, single-document operations are fully ACID. However multi-document and multi-collection queries in a cluster are not ACID [20].

Query language and graph functions

SQL was designed to give answer to some of the relational model needs and it is the powerful declarative query language that made history in databases' world. However, when working with graph structures, SQL rapidly shows to be limited. If graph data is mapped on a relational database, it is of course possible to realize queries with SQL that are able to "move" on the graph and retrieve desired data; but they will increase in complexity (and maybe decrease in efficiency) quite shortly. For these reasons, all three analyzed graph DBMSs developed their own graph-oriented query language.

ArangoDB's query language is called AQL, which is an acronym for ArangoDB Query Language. AQL is a declarative language designed to manage all three data model handled by its DBMS. It supports reading and modifying collection data, but it does not support data-definition operations such as creating and dropping databases, collections and indexes; thus it is a pure data manipulation language (DML) and not a data definition language (DDL) or a data control language (DCL).

The syntax of AQL queries is different from SQL, even if some keywords overlap [9]. It comes with several aggregate, ordering, filtering, and sub-querying functions; furthermore it has an EXPLAIN clause for obtaining query execution insights. It has kind of support for JOIN operations between documents even if, because of its schema-less nature, a null value could be returned in the case of missing attribute on one of the involved documents [18]. It does not have a SELECT clause for choosing the data to be returned like SQL; instead it brings the two keywords FOR and RETURN for choosing what to return as result.

For example, the simple SQL query *SELECT * FROM users* becomes *FOR user IN users RETURN user* with AQL. Furthermore, in order to simulate JOIN operations, nested FOR statements can be used.

For what regards graph traversals, there is a well-defined query syntax that specifies this operation [21], which is reported in Code snippet 3.1.

```
FOR vertex[, edge[, path]]
  IN [min[..max]]
  OUTBOUND|INBOUND|ANY startVertex
  GRAPH graphName
  [OPTIONS options]
  RETURN [...]
```

Code snippet 3.1: AQL graph traversal.

As can be seen, a named graph is passed and, specified the start vertex and edges directions, it is possible to explore the surrounding nodes with depth values specified by the *min* and *max* parameters. With the FOR line, names to the graph elements can be given and later used with the RETURN clause. The OPTIONS clause allows to specify some policies for graph traversal (e.g. in BFS order instead of the default DFS order); furthermore other clauses like FILTER or LIMIT can be specified before invoking RETURN.

AQL can be invoked by the Aardvark web interface and the *arangosh* shell that come with the DBMS; and also by HTTP API and Foxx Services (Foxx is a JavaScript framework for writing data-centric HTTP micro-services that run directly within the ArangoDB server) [9, 26].

Additional details will be given later when talking about realized queries in Section 5.5.

ArangoDB, apart from simple retrieval document queries and graph traversals, allows also to request for shortest path and geo-spatial functions directly with AQL [9].

In addition, several JavaScript functions are also provided, with regards to minimum weight path between two nodes, all paths between two nodes, common descendants of two nodes, nodes that share the same common properties, distance among two nodes, graph radius and diameter, vertices closeness, vertices eccentricity, vertices betweenness.

By using JavaScript, it is also possible to define new functions and also create new visitor or expander methods [28].

ArangoDB does not directly support Gremlin for the specification of its queries [29]; there exist however some GitHub projects for meeting such purpose [36, 27].

Caching

Caching has become a fundamental part of database management systems. Generally, caching means that useful data (typically because of previous accesses from persistent memory) remains in main memory and can be accessed again more rapidly. It is thus clear that queries run faster when the portions of the graph needed to satisfy them already reside in main memory. More generally, caching has three main goals: reducing disk access, reducing computation (i.e. CPU utilization), and speeding up the time as measured by how long it takes a user to see a result.

Three types of caching are generically possible for databases: *query results*, *query plans*, and *data* itself.

The first, *query result caching*, means that the exact output of a read-only query will be stored in main memory for the next time that the exact same query would be performed. This saves the database from doing any disk access, practically removes CPU usage, and returns quickly the result. This is particularly useful if data-reading queries repeat a lot and there are not many write queries.

The second, *query plan caching*, involves saving the results of the optimizer, which is responsible for figuring out how the database is going to fetch the requested data. This type of caching usually involves a "prepared" query, which has almost all of the information needed to run the query with the exception of one or more "placeholders" (i.e. spots that are populated with variables at execution time). Because the plan is already known, the optimizer does not need to be called, which saves CPU usage and time.

The third, *data caching*, involves putting data (usually in terms of graph data structures / tables or indexes) into memory so that it can be read quickly. This saves disk access, which basically means that it saves time.

Each one of the three caching types should complement the other, and a query may be able to use one, two, or all three of the caches [60].

In general, a DBMS may exploits the file system cache for keeping useful graph in memory rather than continuously asking to the disk; or it can even use a dedicated memory area for realizing such page cache. The file system cache, it is managed by the operating system based on some policies like LRU (Least Recently Used) or others.

ArangoDB's processes inherits the file system page cache. In addition to this, ArangoDB realizes a query cache mechanism. The mechanism proposed is transparent to users so they do not need to manually invalidate results in it if underlying collection data are modified. Query cache can be enabled or disabled on the entire server service, or requested/disabled on demand by AQL queries. The query cache is organized as a hash table, so looking up whether a query result is present in the cache is relatively fast.

The query cache mechanism will consider two queries identical if they have exactly the same query string, also including whitespaces etc. The query string will be hashed and used as the cache lookup key; if a query uses bind parameters, these will also be hashed and used as the cache lookup key, so that the same query on new parameters will not be found as already executed. A query will be considered eligible for caching - i.e. it can be saved for later uses - if, among other conditions, is a read-only query; no warnings were generated; and only uses deterministic functions (a random number generator or a timestamp generator are not).

Query cache results are fully or partially invalidated automatically if queries modify the data of collections that were used during the computation of the cached query results; this means that there will be additional cache invalidation checks for each data-modification operation [30].

Scalability approach and data partitioning

Because of the graph data domains could be huge (e.g. think about social networks or energy distribution networks or railway networks), some questions about data scalability arose. While scaling vertically is a way to overcome the problem - and an ingredient for this is to enlarge the secondary memory space and give more power to the server - this typically comes with some disadvantages (most of all of economic nature). On the other hand, NoSQL databases introduced the possibility to scale horizontally, and achieved this in quite economic ways, so it would be nice to be able to apply the same approach for even graph databases.

Horizontal scalability is the ability to distribute workload among several clusters/servers/locations etc. So rather than putting all the burden on one server, you can delegate at least some of the work to other servers to speed things up even with many accesses. In addition, it also enhance resilience by replication and fail-over management.

Sharding is a way to realize horizontal scalability by splitting the data storage itself in different partitions; each of them is stored on a different machines. The cluster of machines however works in such a way that all instances are shown as a single database. The fact of being able to manage a database that is split and placed on different machines leads to a great horizontal scalability opportunity, allowing thus to handle very big amount of data.

This ability is one of the most strong points of NoSQL DBMSs; however for what concerns those realizing the graph database, the question is more delicate. The problem arises by the fact that, as known by the graph theory, the problem of partitioning a graph in distinct partitions is NP-complete [48, 53, 155], and thus impractical. A naïve solution to the problem can lead to unpredictable query times as a result of graph traversals unexpectedly jumping between machines over the (slow) network.

ArangoDB can be scaled horizontally by using many servers, typically based on commodity hardware. The data models realized by ArangoDB offer different opportunities for scalability because of their nature; in particular, the possibility to scale decreases from key/value over documents (documents with joins) to graphs.

The key/value store data model is the easiest to scale, and the reason is that a document collection always has a primary key (*_key* attribute), and in the absence of further secondary indexes the document collection behaves like a simple key/value store. The only operations that are sensible in this context are single key lookups and key/value pair insertions and updates. If _key attribute is the only sharding attribute then the sharding is done with respect to the primary key and all these operations scale linearly.

For the document store case, even in the presence of secondary indexes essentially the same arguments apply, since an index for a sharded collection is simply the same as a local index for each shard. Each shard only holds the part of an index which is needed by this shard. Therefore, single document operations still scale linearly with the size of the cluster.

However, since the AQL query language allows queries that use multiple collections, secondary indexes as well as joins, scaling can be a challenge if the data to be joined resides on different machines, because a lot of communication has to happen. The same happens if working on graph data.

To achieve good performance at scale, it is therefore necessary to set the distribution of the graph data across the shards in a well-studied way. ArangoDB asks the users to specify which attributes to use for the graph data to be sharded. The minimum suggested step for the users is to make sure that the edges originating at a vertex reside on the same cluster node as the vertex. However, for helping with this aspect, ArangoDB Enterprise Edition offers the SmartGraph feature, which understands how to realize graph partitioning based on community detection in order to minimize network transmissions [26].

The distributed architecture is managed by a multi-master model and a number of ArangoDB instances talk to each other over the network and play different roles (Agents, Coordinators, Primary and Secondary DBservers) [26].

Graph visualization

ArangoDB comes with a useful web interface called Aardvark. With this interface the database administrator can see some statistic measures with a dashboard; manage databases, collections and documents; manage named-graphs; perform (and save) AQL queries; manage the database schema (and then impose constraints or create indexes); store and launch services / procedures; look at the database logs; and get a graph-representation of the data contained in the form of a graph model, as shown in Figure 3.5. By using the graph visualization tool, the graph can be dynamically explored by expanding vertexes' connected edges, click-by-click. In the snapshot taken, we searched the desired vertex by applying a filter on vertexes' attributes. We then expanded the first level around the obtained node, and we get the representation of both nodes reachable by incoming and outgoing edges, grouped by similarity (we will not go in details about this). In addition, the graph visualizer allows to create new vertices and new edges in an interactive way. Nodes can be colored in different ways, but only based on static information (i.e. values of their properties).

The version of ArangoDB used in this work lacks the possibility to get a graph visualization of the result of a query. However, during the development of this project, ArangoDB 3.1 has

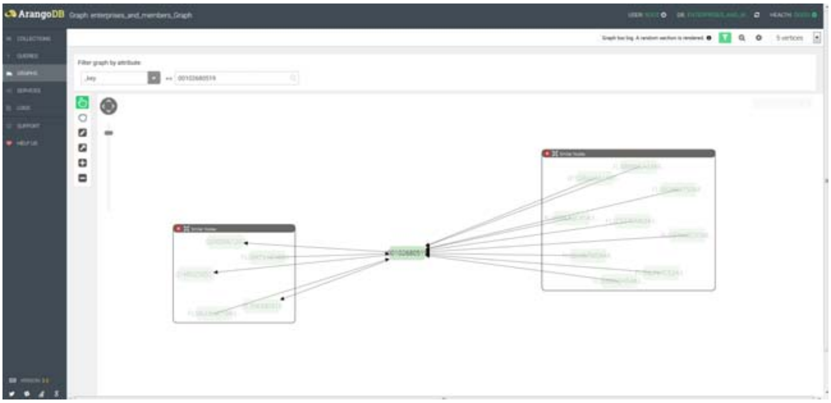
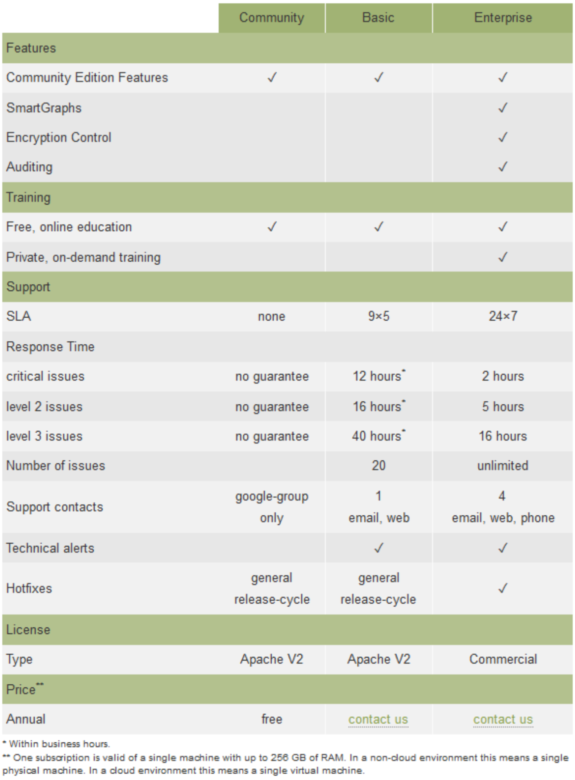


Figure 3.5: ArangoDB web interface



	Community	Basic	Enterprise
Features			
Community Edition Features	✓	✓	✓
SmartGraphs			✓
Encryption Control			✓
Auditing			✓
Training			
Free, online education	✓	✓	✓
Private, on-demand training			✓
Support			
SLA	none	9×5	24×7
Response Time			
critical issues	no guarantee	12 hours*	2 hours
level 2 issues	no guarantee	16 hours*	5 hours
level 3 issues	no guarantee	40 hours*	16 hours
Number of issues		20	unlimited
Support contacts	google-group only	1 email, web	4 email, web, phone
Technical alerts		✓	✓
Hotfixes	general release-cycle	general release-cycle	✓
License			
Type	Apache V2	Apache V2	Commercial
Price**			
Annual	free	contact us	contact us

* Within business hours.
** One subscription is valid of a single machine with up to 256 GB of RAM. In a non-cloud environment this means a single physical machine. In a cloud environment this means a single virtual machine.

Figure 3.6: ArangoDB subscription levels [25].

been implemented which overcomes such shortage.

Licensing

ArangoDB offers, at the time of writing, three different levels of subscription, as shown in Figure 3.6.

As it can be seen, the Community option is the one without direct support and it is released under the Apache v2 license. The other two subscriptions give some level of technical support and are released under different licenses. The Enterprise Edition also includes SmartGraphs feature for graph sharding, auditing feature and increased control over SSL encryption [26].

3.2.2 Neo4j

Realized by Neo Technology Inc (Sweden and USA), Neo4j is an open-source NoSQL graph database implemented in Java and Scala. With development starting in 2002, it has been publicly available since 2007.

According to DB-Engines.com, at the time of writing, it is ranked 1° in Graph DBMS category, and 21° overall.

Neo4j is a native graph storage that realizes the *index-free adjacency property*. Being written in Java, it is portable on all those systems that support it. It is accessible from software written in other languages using the Cypher Query Language through a transactional RESTful HTTP endpoint with JSON based data format; or by the various drivers provided (Go, Groovy, Clojure, Java, JavaScript, PHP, Phyton, Ruby, Scala, etc). It also allows to define stored procedures in Java. In addition, Neo4j can be accessed by its native Java API, and a web user interface.

Neo4j represents the "*historical*" figure in this work: it has been one of the first to be released as a graph database commercial product, and is one of the leading products on the field for popularity and adoption.

In this work, we will use the 3.0.6 version released in 16 September 2016.

Logical data organization

Neo4j's logical organization simply describes the *property graph* model by means of constructs that directly represent the basic elements of such kind of graph model. In fact, its data model is made of node objects (which could be labeled), connected by named and directed relationships, with both nodes and relationships serving as containers for properties.

Even though all relationships have a direction, they can be equally well traversed in both directions, so there is no need to create duplicate relationships in the opposite direction, just like with ArangoDB [76].

Physical data organization

In Figure 3.7 are shown the architectural layers of Neo4j.

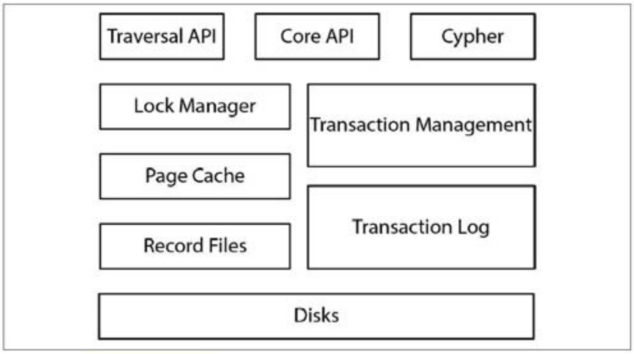


Figure 3.7: Neo4j architecture layers [48].

Neo4j stores graph data in different store files; each store file contains the data for a specific part of the graph (e.g., there are separate stores for nodes, relationships, labels, and properties). In particular, there is a division of storage responsibilities - i.e. the separation of graph structure from property data - which was designed to facilitate graph traversals [48].

As for as the physical structure of single nodes and relationships on disk is concerned, let's exploit Figure 3.8, which describes nodes and relationships store file records.

The node store file, which is named *neostore.nodestore.db*, stores every node created by the user or application. It is a fixed-size record store, where each record is nine bytes in length. Fixed-size records strategy has been applied in order to enable fast lookups for nodes in the store file. In fact, if we have a node with id 100, then we already know that its record begins 900 bytes into the file. Based on this format, the database can directly compute a record's location, at cost O(1), rather than performing a search, which instead would cost O(log n) if a

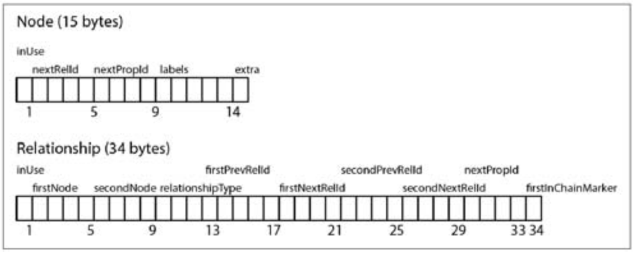


Figure 3.8: Neo4j node and relationship store file record structure [48].

typical binary-tree index is used.

The first byte of a node record is the *in-use* flag; this tells the database whether the record is currently being used to store a node, or whether it can be reclaimed on behalf of a new node. Neo4j's *.id* files are those that keep track of unused records. The next four bytes represent the ID of the first relationship connected to the node, and the following four bytes represent the ID of the first property for the node. The five bytes for labels point to the label store for this node (labels can be in-lined where there are relatively few of them). The final byte *extra* is reserved for flags; one such flag is used to identify densely connected nodes, and the rest of the space is reserved for future use.

The node record is then quite lightweight, indeed it's just a handful of pointers to linked lists of relationships, labels, and properties [48].

Correspondingly, relationships are stored in the relationship store file, which is *neostore.relationshipstore.db*. Like the node store, the relationship store also consists of fixed-sized records. Each relationship record contains the IDs of the nodes at the start and end of the relationship, a pointer to the relationship type (which is stored in the relationship type store), pointers for the next and previous relationship records for each of the start and end nodes, and a flag indicating whether the current record is the first in what's often called the relationship chain.

Let's now consider an example of a social network where two nodes are connected by an edge, as shown in Figure 3.9.

Each of the two node records contains a pointer to that node's first property and first relationship in a relationship chain. To read a node's properties, we follow the singly linked list structure beginning with the pointer to the first property. To find a relationship for a node, we follow that node's relationship pointer to its first relationship. From here, we then follow the doubly linked list of relationships for that particular node (that is, either the start node doubly linked list, or the end node doubly linked list) until we find the relationship we're interested in. Having found the record for the relationship we want, we can read that relationship's properties (if there are any) using the same singly linked list structure as is used for node properties, or

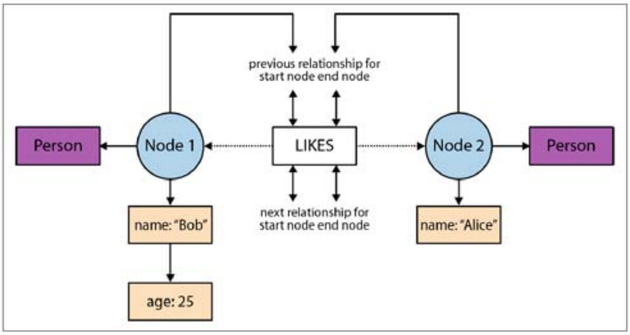


Figure 3.9: Example Neo4j social network data structure [48].

we can examine the node records for the two nodes the relationship connects using its start node and end node IDs [48].

It must be told that, because of the way edges and properties are stored and retrieved, the first inserted property for a node or an edge is always the first to be found, while other properties would be found only after being passed through the first one. The same holds for the chain of relationships connected to a node. First edges and properties are then a little bit privileged with respect to others. This fact should not impact in query execution too much if the linked lists are quite small; nevertheless it could be a good point for those who would push to the limit the optimization of the database, if it is known that some kind of relationships or properties are requested more times than others.

By thinking of a relationship record as "belonging" to two nodes: the start node and the end node of the relationship, it becomes clear that there would be the problem of storing the same relationship record twice, which would be wasteful, if done improperly. This is the reason why, instead, there are pointers (aka record IDs) for two doubly linked lists: one is the list of relationships visible from the start node; the other is the list of relationships visible from the end node.

Furthermore, the fact that each list is doubly linked enables to rapidly iterate through that list in either direction, and insert and delete relationships efficiently. Choosing to follow a different relationship involves iterating through a linked list of relationships until we find a good candidate (e.g., matching the correct type, or having some matching property value). Once we have a suitable relationship we're back in business, multiplying ID by record size, and thereafter chasing pointers [48].

To traverse a particular relationship from one node to another, the database performs several cheap ID computations (these computations are generally cheaper than searching global indexes, as we'd have to do if faking a graph in a *non-graph native* database):

- From a given node record, locate the first record in the relationship chain by computing its offset into the relationship store—that is, by multiplying its ID by the fixed relationship record size. This gets us directly to the right record in the relationship store.
- From the relationship record, look in the second node field to find the ID of the second node. Multiply that ID by the node record size to locate the correct node record in the store [48].

In case of constrain, during traversal, based on relationships with particular types, there will be performed a lookup in the relationship type store. Again, this is a simple multiplication of ID by record size to find the offset for the appropriate relationship type record in the relationship store. Similarly if it is chosen to constrain by label, the label store will be referenced.

In addition to the node and relationship stores, which contain the graph structure, there are property store files, which persist the user's data in key-value pairs. Properties can, as already stated, be attached either to nodes and edges. The property stores, therefore, are referenced from both node and relationship records. Records in the property store are physically stored in the *neostore.propertystore.db* file. As with the node and relationship stores, property records are of a fixed size. Each property record consists of four property blocks and the ID of the next property in the property chain (remember, properties are held as a singly linked list on disk as compared to the doubly linked list used in relationship chains). Each property occupies between one and four property blocks; a property record can, therefore, hold a maximum of four properties.

A property record holds the property type (Neo4j allows any primitive JVM type, plus strings, plus arrays of the JVM primitive types), and a pointer to the property index file (*neostore.propertystore.db.index*), which is where the property name is stored. For each property's value, the record contains either a pointer into a dynamic store record or an in-lined value. The dynamic stores allow for storing large property values [48].

Neo4j supports store optimizations, whereby it inlines some properties into the property store file directly (*neostore.propertystore.db*). This happens when property data can be encoded to fit in one or more of a record's four property blocks. This results in reduced I/O operations and improved throughput, because only a single file access is required.

In addition to inlining certain compatible property values, Neo4j also maintains space discipline on property names. For example, in a social graph, there will likely be many nodes with properties like *first_name* and *last_name*. It would be wasteful if each property name was written out to disk verbatim, and so instead property names are indirectly referenced from the property store through the property index file. The property index allows all properties with the same name to share a single record, and thus for repetitive graphs Neo4j achieves considerable space and I/O savings [48].

For the way by which nodes are bond with properties, it can be seen a little analogy with how RDF graphs are codified by triples. Indeed, the analogy is that nodes do not own properties within themselves, because as already stated data is separated by the underlying structure. However, these properties are reachable by nodes because nodes contains pointers (or links) to them; just like an RDF node does. Yet, this little (philosophical, if you like) analogy does not holds further.

Data Integrity

For what regards data consistency, unique property constraints can be applied to ensure that property values are unique for all nodes with a specific label; however unique constraints do not mean that all nodes have to have a unique value for the properties, because nodes without the property are not subject to this rule. In this case, property existence constraints (only available in Neo4j Enterprise Edition) can be used to ensure that a property exists for all nodes with a specific label or for all relationships with a specific type; and all queries that try to create new nodes or relationships without the property, or queries that try to remove the mandatory property, will fail. The creation of a unique property constraint will collaterally add an index on that property, and Cypher will use that index for lookups just like other indexes. If the unique property constraint is dropped, also the index will be, so in case of index needs on such property, a new index has to be defined [66]. Once an index has been created, it will automatically be managed and kept up to date by the database whenever the graph is changed. Neo4j will automatically pick up and start using the index once it has been created and brought online. However, there is not the possibility to define which kind of index to create (the type is probably hash index) [68, 75].

For what regards graph consistency, the Cypher DELETE command is the only way to edit the graph for deleting both nodes, edges and even paths. The command assures that it is not possible to delete a node without also deleting relationships that start or end on said node; so, in order to delete the note, one way is to explicitly delete the relationships before doing it; the other is by using DETACH DELETE which also deletes the edges involved. For Cypher data insertion checks, which occur at edge creation moments, the syntax for using the CREATE Cypher clause assures that edge creation is done only by specifying the two nodes involved, which have to be preliminarily obtained by a MATCH clause; so no dangling edges can be created on the graph [66]. Furthermore, relationships are also granted to never hang freely by the DBMS itself, because of checks made at edge creation [76].

Neo4j also behaves like an ACID DBMS when on a single instance. Any query that updates the graph will run in a transaction: if no transaction exists in the running context, Cypher will create one and commit it once the query finishes; in case there already exists a transaction in the running context, the query will run inside it, and nothing will be persisted to disk until that transaction is successfully committed. However this implies that only one single operation can be done within a transaction, so in order to execute multiple operations together and have the commit only in case of overall success, an approach of enclosing the multiple updating queries within the same transaction and committing all of them in one go can be used. Note that a query will hold the changes in memory until the whole query has finished executing; so a large query will need a JVM with big enough heap space [72].

One such example is provided by the Neo4j transactional HTTP endpoint, which allows to execute a series of Cypher statements within the scope of a single transaction; the transaction may be kept open across multiple HTTP requests, until the client chooses to commit or roll back [71].

Another example for realizing transactions is by using the Java API, where all database operations that access the graph, indexes, or the schema must be enclosed in a try block, as shown in Code snippet 3.2. As we exit the block, the transaction will automatically be closed by the invocation of *tx.close()* which will commit the transaction if the internal state indicates success or else mark it for rollback [77].

Code snippet 3.2: Neo4j Java API transaction block.

```
try ( Transaction tx = graphDb.beginTx() )
{
    // operations on the graph

    tx.success();
}
```

The DBMS will manage incoming transactions, and in case two or more of them attempt to change the same graph elements concurrently, it will detect a potential deadlock situation, and serialize the transactions. For what concerns their implementation, each transaction is represented as an in-memory object whose state represents writes to the database. This object is supported by a lock manager, which applies write locks to nodes and relationships as they are created, updated, and deleted. On transaction rollback, the transaction object is discarded and the write locks released, whereas on successful completion the transaction is committed to disk.

Committing data to disk in Neo4j uses a Write Ahead Log, whereby changes are appended as actionable entries in the active transaction log. On transaction commit (assuming a positive response to the prepare phase) a commit entry will be written to the log. This causes the log to be flushed to disk, thereby making the changes durable. Once the disk flush has occurred, the changes are applied to the graph itself. After all the changes have been applied to the graph, any write locks associated with the transaction are released [48].

Query language and graph functions

Cypher is the intuitive declarative query language provided by Neo4j. It is fully designed for working on graph data and defines the structure of the pattens to be searched or created over the graph data. It borrows its syntax by the way one would generally draw graphs; in fact, it uses ASCII art for the representation of the graph patterns. Let's see a short example: suppose we have a graph as shown in Figure 3.10; the equivalent ASCII art representation in Cypher is shown in Code snippet 3.3.

```
(emil)<-[:KNOWS]-(jim)-[:KNOWS]->(ian)-[:KNOWS]->(emil)
```

Code snippet 3.3: Graph ASCII art representation.

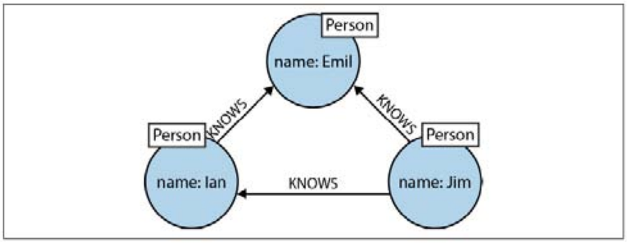


Figure 3.10: A simple Neo4j example graph [48].

However, the previous Cypher pattern describes a simple graph structure, and it does not yet refer to any particular data in the database. To bind the pattern to specific nodes and relationships in an existing dataset we must specify some property values and node labels that help locate the relevant elements in the dataset.

```
(emil:Person {name: 'Emil'})<-[:KNOWS]-(jim:Person {name:
    'Jim'})-[:KNOWS]->(ian:Person {name: 'Ian'})-[:KNOWS]->(emil)
```

Code snippet 3.4: Binding with the graph.

A Cypher query anchors one or more parts of a pattern to specific locations in a graph using predicates, and then flexes the unanchored parts around to find local matches. The anchor points in the real graph are determined based on the labels and property predicates in the query.

An example of a Cypher query is given below, the request is to find the friends of a user named John:

```
MATCH (john:Person {name:'John'})-[:KNOWS]->(friend)
RETURN friend
```

Code snippet 3.5: Cypher MATCH clause.

Cypher is then quite different from SQL; it replaced the SELECT clause with the pair MATCH & RETURN; and comes with different keywords for specifying insertion, update or removal of graph data. It also comes with several functions which works with data types, data lists, etc. Cypher also gives the possibility to invoke stored procedures, pass them some arguments and elaborate the results obtained. It is also a language for modifying meta-data like indexes and constraints.

Cypher statements can be executed by the web interface and the shell tool provided with the DBMS. It can also be executed within stored procedures (and in general by the Java API); by other language drivers and by the HTTP API.

The default neighbors exploration strategy for Neo4j is Depth-First ordering. It is also the only possibility when working with only Cypher, as will be explained in Chapter 5. It is not possible, in fact, to force a Breadth-First approach with Cypher, so one can only simulate it's execution by means of *a posteriori* sorting and filtering operations on obtained results.

In addition, while matching the patterns specified with Cypher, Neo4j makes sure to not include matches where the same edge is found multiple times in a single path [66]. So, like ArangoDB, also Neo4j imposes by default the unique visit of edges on path scope.

Each Cypher query is turned into an execution plan by the execution planner, and the execution plan tells Neo4j which operations to perform when executing the query. Two different execution planning strategies are included in Neo4j: based on rules (using indexes but not statistical information); and based on costs (using statistical information), which is the default configuration in Neo4j 3.0 [70].

Cypher, apart from allowing graph traversals, also allows to find the shortest path and all possible paths between two nodes directly by Cypher.

However, the Java Traversal API allows to realize highly-customized user-defined ways to traverse the graph, by also implementing new visitor methods and similar. In addition, Java stored procedures can be implemented, and they can use the Traversal API for defining how to explore the graph.

In addition, the APOC GitHub repository (Awesome Procedures On Cypher) collects a set of stored procedures already implemented that can be imported and called directly by Cypher. These procedures can regard both general database management and graph queries, like the possibility to call the Dijkstra algorithm, or the A-star algorithm, betweenness, closeness, pageRank, clique detection, etc [8].

Neo4j does not have native support for Apache TinkerPop & Gremlin, however there is an implementation for this purpose [128].

Caching

Neo4j uses a dedicated page cache in order to keep graph data and indexes in RAM, so that disk hits are heavily reduced. The page cache is an LRU-K page-affine cache, meaning the cache divides each store into discrete regions, and then holds a fixed number of regions per store file. Pages are removed from the cache based on a least frequently used (LFU) cache policy, nuanced by page popularity. That is, unpopular pages will be evicted from the cache in preference to popular pages, even if the latter haven't been touched recently [48].

The page cache should be utilized as much as possible, and ideally the entire store files should be loaded in this cache for best performance; of course this may be impossible if working with huge graphs. However, in that case, a "cache sharding" approach can overcome such problem.

The page cache has to be defined based on 4 parameters: available RAM, store files weight, JVM heap space and operating system's memory usage. In particular, the best configuration is the one that has enough RAM available to contain all store files within the page cache, the heap space, and the memory left for OS operations. Depending on the machine usage, the OS memory usage may varies; however and a rule-of-thumb for a not heavy loaded server is to leave nearly 1GB for the OS. If configuring page cache and heap space equal to or greater than the available RAM, or if not leaving enough head room for the OS, the OS will start swapping to disk, which will heavily affect performance [69].

So when the service starts up, its page cache is empty and needs to be warmed. This can take a while, especially for large stores. The DBMS also flushes its page cache in the background, in order to maintain durability.

In addition to the page cache, Neo4j also employs a query plans cache for rapidly retrieve query plans already executed.

Scalability approach and data partitioning

The solution proposed by Neo4j for dealing with large datasets and maintaining good performance is to use a technique called "*cache sharding*". Cache sharding is realized by a High-Availability cluster comprised of a single master instance and more slave instances. All instances in the cluster have full copies of the data in their local database files; furthermore each instance contains the logic needed in order to coordinate with the other members of the cluster for data replication and election management.

Cache sharding is thus a form of load balancing for incoming queries. The principle is to have different partitions of the huge underlying graph distributed on the main memories of the slave instances, and a special node routes all those incoming queries targeting the same specific node / graph region to the same slave instance, so that the query will (with high probability) be resolved without accessing the disk. In this way, read operations are highly available and the ability to handle read load scales with more database instances in the cluster [64].

In any case, it must be highlighted that each of the cluster's instances will have a full copy of the data, and only cached data will be somehow differentiated on the different database instances; so this is not a properly called sharding mechanism. Furthermore, clustering features are only available in Neo4j Enterprise Edition.

Graph visualization

Also Neo4j comes with a web interface, which is called Neo4j Browser. Whit this interface, one can perform (and save) queries; manage the database schema; get some basic statistics; and explore the graph by expanding nodes, as shown in Figure 3.11.

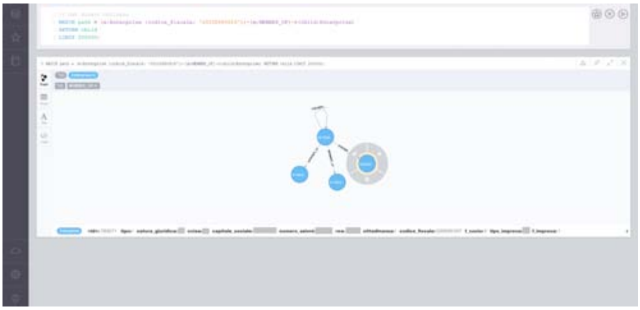


Figure 3.11: Neo4j web interface

For what regards queries visualization, Neo4j Browser always shows its results in a visual way by means of nodes and edges; which may also be colored differently based on static information (like the property values, or the labels defined, etc). By the nodes returned by the query, it is possible to further expand the neighbors node, or hide some of them, in an interactive way, click-by-click. The visualizer does not allow to interactively define new vertexes or edges, such task is left to the above area where queries are written and launched. Together with the graph visualization, the interface returns data also in tabular or JSON formats.

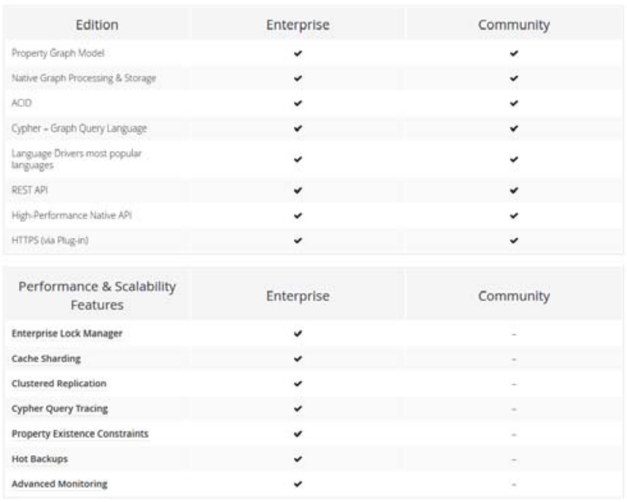
One limit of this visualization tool is that it does not highlight the anchor nodes (i.e. the ones by which the exploration has started), and also does not allow to draw nodes or edges with different colors based on dynamic information, that is something like "color the result nodes with different shades of red according to the number of edges linked to them".

Licensing

At the time of writing, Neo4j offers a Community Edition that is "intended for learning and smaller do-it-yourself projects that do not require high levels of scaling." [74] It excludes professional services and support and it is released under the GPL v3 license.

Neo4j also offers several different licenses for the Enterprise Edition of the software. The Enterprise Edition is intended for businesses and comes with additional features.

Figure 3.12 shows some features difference between Community and Enterprise editions, including cache sharding and clustered replication, database monitoring, hot backups and enhancement in concurrency management.



Edition	Enterprise	Community
Property Graph Model	✓	✓
Native Graph Processing & Storage	✓	✓
ACID	✓	✓
Cypher – Graph Query Language	✓	✓
Language Drivers most popular languages	✓	✓
REST API	✓	✓
High-Performance Native API	✓	✓
HTTPS (via Plug-in)	✓	✓

Performance & Scalability Features	Enterprise	Community
Enterprise Lock Manager	✓	-
Cache Sharding	✓	-
Clustered Replication	✓	-
Cypher Query Tracing	✓	-
Property Existence Constraints	✓	-
Hot Backups	✓	-
Advanced Monitoring	✓	-

Figure 3.12: Neo4j editions comparison [61].

Within the Enterprise Edition there are different subscription options targeting commercial, evaluation or educational purposes; with also different levels of support [63].

The Community Edition is free but is limited to running on 1 node only due to the lack of clustering and is without hot backups; furthermore, only one database instance can be managed at a time; and the web interface does not show some statistics measures.

3.2.3 OrientDB

OrientDB - initially developed in Italy [51] and now with venue in UK as OrientDB Ltd - was born as an Object Oriented database, and it is now a multi-model database management system operating on objects, documents, key/value pairs and graphs. It inherits the schema-less nature of documents store and employs the relationships management approaches of the graph database world [112].

According to DB-Engines.com, at the time of writing, it is ranked 2° in Graph DBMS category, 6° in Document Stores category, 6° in Key-Value Stores category, and 45° overall.

It is developed in Java - thus portable - and designed to be horizontal scalable and to work in a distributed environment. It has a variety of drivers for different programming languages

(C/C++, C#, Clojure, Java, JavaScript, PHP, Phyton, Ruby, Scala, etc) and exposes a RESTful HTTP API and JSON based data in order to interact with web applications. In addition, the DBMS can be accessed via its native Java API.

Like ArangoDB, it also claims to be a "true multi-model DBMS". In this sense, it wants to emphasize the fact that it does not realize its multi-model characteristic by means of different interfaces on the database engine; but rather the engine itself is built to differently support all its four models [87].

It comes with a sort of dialect of the SQL query language; extension that is necessary in order to give the possibility to define queries on a graph data domain.

Also OrientDB comes with a web interface for managing its databases instances and visually exploration of the graph.

OrientDB represents the "*historical alternative*" to Neo4j, with also the added value of being a multi-model DBMS, which makes it a multi-purpose NoSQL database able to enrich the graph data.

During this work, we will use the 2.2.11 version released in 3 October 2016.

Logical data organization

The smallest unit that can be loaded from and stored into the database is the *Record*, which can be of four types: Document; RecordBytes (BLOB); Vertex and Edge. Vertex and Edges are thus managed differently from general purpose documents, and they also represent the smallest data unit manageable.

The *Class* is a concept drawn from the Object-oriented programming paradigm. It is a type of data model that allows to define certain constraints for the records that belong to it. In the traditional document database model, it is comparable to the collection, while in the relational database model it is comparable to the table.

Classes can be schema-less, schema-full or a mix (called schema-hybrid); they can also inherit properties from other classes, creating this way a hierarchical tree of classes.

Each class must have at least one cluster defined, which functions as its default cluster; however a class can support multiple clusters. When a query is executed against records of a specific class, the DBMS automatically propagates the execution to all clusters that are part of that class [87].

The *Cluster* is a place where a group of records are stored. Like the Class, it is comparable with the collection in traditional document databases, and in relational databases with the table. However, this is a loose comparison given that unlike a table, clusters allow to store the data of a class in different physical locations [85].

To each generated record, the DBMS auto-assigns it a unique ID within the database called

Record ID or RID, which is composed of two parts: #<cluster-id>:<cluster-position>. The DBMS automatically selects the cluster where it stores the physical data of that record. Some different strategies can be configured in order to specify the cluster selection process.

There exist two particular types of cluster:

- Persistent Cluster: Also called Physical cluster, it stores data on disk.
- In-Memory cluster: The information stored in "In-Memory clusters" is volatile; this can be used for storing temporary data [87].

The combination of clusters and RIDs is the way by which OrientDB makes it possible to have a distributed dataset. In fact, RID is not only a unique identifier value, but gains even more importance because it embeds the physical position of such record inside the database. What this means is that when loading a record by its RID, the load is fast because its location is already known by means of a mapping function between RIDs and cluster locations. Records are organized in classes and clusters [87].

Key-value model
The simplest NoSQL model implemented by OrientDB is the key-value model, by which everything in the database can be reached via a key; and where the values can be both simple and complex types. OrientDB supports Documents and Graph Elements as values.

Document model
The document is the most flexible record type available. Documents are softly typed and are defined by schema classes with defined constraints (schema-full), but it is also possible to use them in a schema-less mode or in a schema-hybrid mode.

A document is a set of key/value pairs (also referred to as fields or properties), where the key allows access to its value. As for ArangoDB, values can hold primitive data types, embedded documents, or arrays of other values.

OrientDB's document model also adds the concept of *link* as a relationship between documents, so that it can be decided whether to embed documents or to link them directly. On a later moment, when a document is fetched, all the links are automatically resolved [87].

Object model
This model has derives from Object Oriented programming and supports some of its peculiarities like *inheritance* between types, *polymorphism*, and *direct binding* from or to the objects used in programming languages.

One of the most interesting characteristic is *inheritance* among records' classes, because it allows to perform queries on records belonging to different classes with just one statement.

Graph model
For what regards OrientDB's logical representation of graphs, again it exploits its Object-Oriented database origins for creating Vertex and Edge elements.

Vertex elements are objects of class (or subclasses of) V; edge elements are analogously objects of class (or subclasses of) E. The description given for documents also holds for these two kind of record types.

Since also OrientDB complies with the *property graph* model, it allows to define fields representing properties for both vertexes and edges. Labels (or better the possibility to perform a categorization) of vertexes and edges is instead represented by a differentiation made with the classes they belong to. Each Vertex object, besides all the properties defined on it, owns the already mentioned unique identifier that allows to identify it within the collection; plus a set of incoming Edges and a set of outgoing Edges. An Edge element, instead, owns a link to an incoming Vertex; a link to an outgoing Vertex; and a "label" (or better class) that defines the type of relationship between the two vertexes.

Edges can be regular or lightweight. The Regular Edge is saved as a Document, while the Lightweight Edge is not. In fact, Lightweight Edges do not have Record IDs, but are physically stored as links within vertices. Note that OrientDB only uses a Lightweight Edge when the edge has no properties, otherwise standard Edge should be used.

Edges in OrientDB can be equally traversed in both directions just like with ArangoDB and Neo4j.

Physical data organization

OrientDB claims to support three storage types where documents can be stored: *plocal*, *remote* and *memory*.

- *Plocal* is the persistent storage, and it refers to the localhost for storing data.
- *Remote* storage is the one to be used when a remote storage have to be accessed. It refers then to a remote plocal storage; so it is more a way to access a database rather than a proper storage mechanism.
- *Memory* storage is local and endures only as long as the JVM is running. For the application programmer, it is identical in use to Paginated Local Storage. Transactions remain atomic, consistent and isolated (but are not durable). The memory available for the database is the memory allocated by the JVM.

Let's view in detail the fundamental storage type, which is plocal.

The Paginated Local Storage, **plocal**, is a disk based storage which works with data using the pagination model; every file is spit into pages, and each file operation is atomic at a page level. It works by exploiting together the 2-level Disk-Cache with the WAL. The 2-level cache consists of a Read Cache and a Write cache managed by means of two distinct algorithms (2Q for Read Cache and WOW for Write Cache).

By using Disk-Cache and WAL OrientDB achieves a the classical durability/performance trade off; because it does not need to flush every page to the disk and still can achieve durability using much cheaper append only I/O operations [96], like for ArangoDB.

Here we show the list of plocal storage components and a short description for each of them:

- Clusters are managed by two kinds of file:
  - *.pcl* files contain the (cluster) data;
  - *.cpm* files contain the mapping between record's cluster position and real physical position.
- Write-Ahead Log (WAL) is managed by two kinds of file:
  - *.wal* to store the log content;
  - *.wmr* contains timing about synchronization operations between storage cache and disk system.
- SBTree Index, it uses files with extensions *.sbt*.
- Hash Index, it uses files with extensions *.hit*, *.him* and *.hib*.
- Index Containers to store values of single entries of not unique index (Index RID Set). It uses files with extension *.irs*.
- File mapping, maps between file names and file ids (used internally). It's a single file with extension *.cm* [87].

OrientDB thus stores data and auxiliary information in some customs files; data records (Documents, Vertex and Edge objects) are then directly serialized to *.pcl* files.

One of the most important file is the mapping file, with extension *.cpm*, which is used for mapping records cluster positions to real physical positions. These files are basically lists: each entry in this list is a fixed size element, which is the pointer to the physical position of the record in the data file. Because the data file is paginated, this pointer will consist of 2 items: a page index (long value) and the position of the record inside the page (int value). Each record pointer consumes 12 bytes.

When a new record is inserted, a pointer is added to the list; the index of this pointer is the cluster position. The list is an append only data structure, so if you add a new record its cluster position will be unique and will not be reused [96].

It is true that by using this mapping approach, the linking information among nodes is directly stored persistently and does not have to be computed at run-time by performing JOIN operations; however a lookup on this mapping file has to be performed because the pointers to the connected nodes are not physical pointers (as for Neo4j). This however comes useful in a distributed environment, because clusters can be stored and referenced among different storage devices.

So given the fact that also indexes are mapping files between values and physical positions, if you want, rather than a strictly speaking *index-free adjacency*, OrientDB realizes a "*join-free adjacency*".

Data Integrity

OrientDB defines three kinds of schema mode: schema-full, schema-less and schema-hybrid. The schema mode is defined at class-level and is valid for all records belonging to the class. The schema-full mode sets all defined fields to be mandatory; the schema-less mode instead defines classes with no fields, so records can have arbitrary properties; the schema-hybrid mode instead creates classes and defines some of the fields, but let the records to define other custom fields [91]. The access to the schema is possible both by the SQL-like query language and the Java API.

In order to impose a unique constraint on some fields, an index (of any kind) with "unique" specification on its definition should be created. Note that this gives also the possibility to specify a unique constraint on the pair made of source and target node for Edge records, which implies that only one edge can be defined between the same two nodes. Other constraints such as NOT NULL can be set with the schema defined on the record's class.

OrientDB, like ArangoDB, support different data models, but exposes a unique query language for working on them. However, differently from AQL, its SQL dialect have multiple ways to perform the deletion of a record; and since vertexes and edges are records, this may produce graph inconsistency.

The first delicate situation is at edge creation: instead of the generic INSERT clause, the CREATE EDGE clause is the one to be used for achieving the goal, which checks that specified vertexes already exist before creating the edge between them. The second delicate situation is at vertex deletion. Instead of using the general DELETE command for deleting a generic record of the database, the DELETE VERTEX clauses should be used, which also deletes all the edges linked to such vertex. However, a third case exists, which is at edge deletion: also in this case the DELETE command should not be used, because references to edge elements contained in involved vertexes will break; so the command DELETE EDGE should be used instead, which clears such references [95].

The default behaviour for the DBMS is to execute commands without transactions; however the optimistic mode can be used, which involves the Multi Version Control System (MVCC) that can permits multiple reads and writes on the same records. Records maintain their own version number, which increments on each update. The integrity check is made on commit: if the record has been saved by another transaction in the interim - that is if the version number has changed - an exception will be thrown and the application should choose whether to repeat the transaction or to abort it [98].s, based on the size of the available memory to the JVM.

When using Graph API for managing the graph, transactions begin automatically; with Document API instead the beginning is explicit by invoking the *begin()* method.

OrientDB does not support nested transaction; and if further *begin()* invocations are performed, a stack trace will be maintained in order to commit all the operations at the last *commit()* found. In addition, as for Neo4j, for a heavy transaction it should be preferable to split it in multiple smaller transactions.

It is though possible to disable transaction behaviour even with Graph API by instantiating the database as an object of the class OrientGraphNoTx; and this may be useful in some circumstances.

Creating graphs, indeed, consists in creating vertices and connecting them with edges. However, adding a single edge to an existing database consists in three operations:

- creating the edge document;
- updating the left vertex to point to the edge;
- if an edge is inserted, it is checked that such edge matches the edge definitions, so that edge collections will only contain valid edges

and, at low level, the operations are:

- load vertex1 by key (index lookup);
- load vertex2 by key (index lookup);
- create edge document setting *out* = vertex1.@rid and *in* = vertex2.@rid;
- add the edge RID to vertex1.*out_EdgeClass*;
- add the edge RID to vertex2.*in_EdgeClass*.

As a result, the creation of an edge can be considered an expensive operation compared to a simple document creation; thus, in some circumstances, the batch graph creation can be made faster by disabling transactions or at least enhancing the interval of graph operations before a transaction commit [90].

Query language and graph functions

OrientDB, instead of creating *Yet-Another-Query-Language* for its graph query purposes, preferred to focus on the standard, consolidated and well-known SQL. It thus extended SQL in order to include graph functionality with only minor changes on the rest of the standard syntax.

OrientDB SQL dialect can be used both as DDL and DML, and also provides statements for creating databases or managing users. The main difference from SQL is that the JOIN command does not exist in this dialect, because OrientDB does not need do perform such operation since records already contain references to the connected records by means of LINKs. In this way, connected records are reachable by just using the "dot notation" typical of the high-level programming languages, like $< field\_name\_1 > . < field\_name\_2 >$; where *field_name_1* is the field of the start record and *field_name_2* is the desired field of the linked record. A similar approach is also used when traversing edges.

The combination of standard SQL and LINKs allows to specify multi-level queries in an easier way with respect to using JOINs. However, the number of edges to be traversed is hard-written within the query by the number of “dot dereferentiations” performed.

In order to provided a more flexible and simplified way to express graph traversals, OrientDB defined two statements which allow to specify variable depth values, which are MATCH and TRAVERSE. The MATCH command behaves like Cypher’s Match clause, i.e. by initially specifying the anchor node, and then starting exploring the surrounding graph by following the desired edges, with the indicated direction, and to the depth level specifiable in the *while* statement. One quick example is provided in Code snippet 3.6. Note that a *where* statement can also be evaluated on reached nodes (i.e. those indicated by the variable *friend*).

```
MATCH {class: Person, where: (name = ‘John’)}.both("Friend"){as: friend,
    while: ($depth < 3)} RETURN friend
```

Code snippet 3.6: OrientDB SQL MATCH command.

The TRAVERSE command behaves more like AQL Traversal; however the principle remains the same as the previous command. The main differences are that it is possible to specify the exploration order (BFS or DFS) and the absence of a *where* statement, both for edges and for reached nodes.

```
TRAVERSE out('Friend') FROM (SELECT FROM Person WHERE name = 'John')
    WHILE $depth < 3 STRATEGY BREADTH_FIRST
```

Code snippet 3.7: OrientDB SQL TRAVERSE command.

OrientDB SQL queries can be invoked by the web interface and the shell tool provided; by the Java API, the drivers for other programming languages, and the HTTP API.

The default neighbors exploration strategy for OrientDB is Depth-First; however it is possible to force a Breadth-First approach, but only by using the TRAVERSAL

In addition, OrientDB imposes by default the unique visit on vertexes with path scope, thought for avoiding possible loop situations caused by path cycles.

The SQL dialect provides also shortest or minimum weight path algorithms. In addition, OrientDB allows to define new JavaScript functions, which can be called by SQL or REST API; and also comes with a Java Graph API that allows to manage and query graphs [95].

OrientDB also follows the TinkerPop Blueprints standard and uses it for the Java Graph API [89]; and it also natively support Gremlin [92].

Caching

OrientDB uses two caches: the level 1 cache acts at thread level, while the level 2 cache acts at JVM level. The caching architecture can be seen in Figure 3.13.

If the database is mainly used for read operations, the configuration can be left like this; however in a multi-threaded scenario and with many writes, disabling the L1 cache would be the best choice, so that inconsistencies between L1 caches are limited. For the same reason, if in a multi-JVM scenario, it should be considered to disable the level 2 cache also [121].

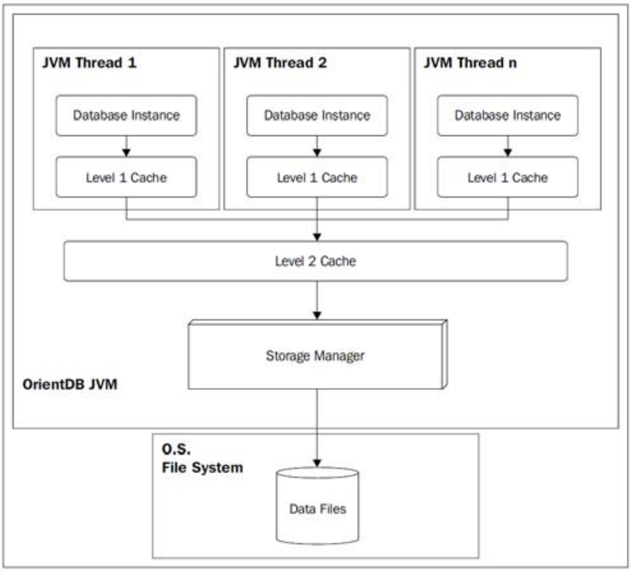


Figure 3.13: OrientDB caching architecture [121].

The L1 cache (also referred to with the name "local cache") aims to keep the used records at the higher level possible for each database instance. If there are not "usage" pointers to a particular record, or if the JVM requires more heap space, the record will be removed from this cache level.

The L2 cache caches file reads with the goal of reducing disk hits and keep most used files in main memory. OrientDB generally refers to it also with the term "Disk cache", and it corresponds to the Page cache described with Neo4j. OrientDB Disk cache consists of two separate cache components that work together: Read Cache, based on 2Q algorithm; and Write Cache, based on WOW cache algorithm. The goal of the write cache is to collect those file pages that need to be persisted on disk in order to manage and optimize the I/O accesses with a background process [84].

OrientDB also realizes a command cache that accomplishes the same goals of ArangoDB and Neo4j query caches. OrientDB by default turns off this mechanism; the main reason is probably because this approach can be dangerous in case of big result sets in a low RAM environment [86].

Scalability approach and data partitioning

OrientDB supports data sharding at class level; this is done by using multiple clusters per class, and each cluster holds its own list of servers where data is stored or replicated.

Thanks to the RID structure, it is known where each record is stored, and since Vertex records contain Edges' RIDs for reaching them, records can be stored in different memory spaces (different machines) and still able to point to their linked records. However OrientDB does not provide a mechanism that tries to find out the best clustering configuration for its records in order to limit expensive network data exchanges during queries; so such task is in charge to the application or the user. The the minimum hint to be done for data partitioning to not slow down query times too much is to store within the same machine at least one of the two vertices that are connected by the edges in the edges subset contained within that machine [85].

In order to manage a distributed environment, OrientDB realizes a multi master architecture, with several slave nodes.

Graph visualization

OrientDB comes with a web interface, called OrientDB Studio. With this interface it is possible to manage databases and users; to manage class schemas; to get the data organization in classes; to perform and save queries; to store and call procedures; and also to visualize and explore the graph, as shown in Figure 3.14.

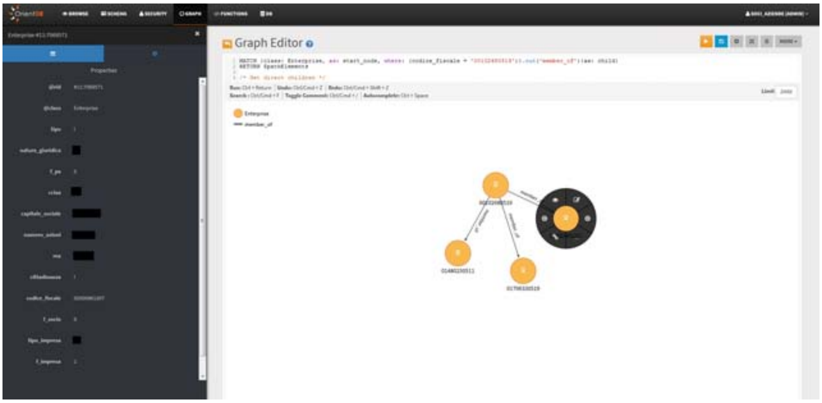
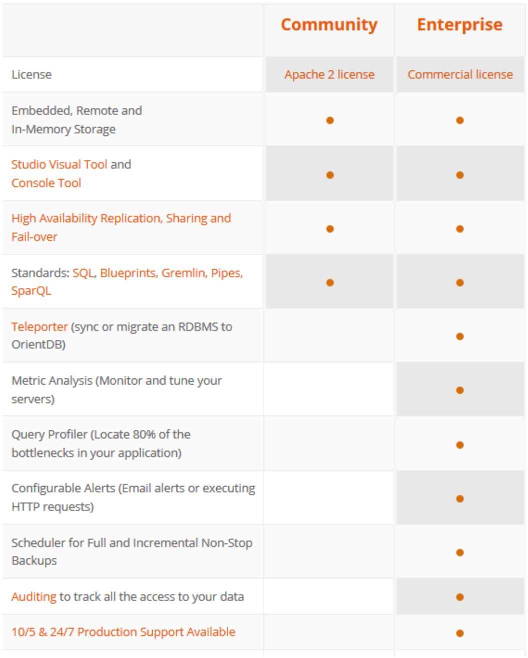


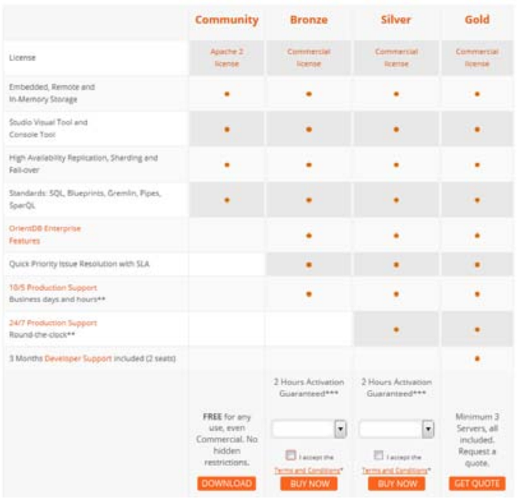
Figure 3.14: OrientDB web interface

OrientDB Studio can show a query result in a graph way. However this is only possible if some special keywords are used within the return statement of the query, keywords that corresponds to some specific query variables (such as $elements, $pathElements, etc). Nodes can be



	Community	Enterprise
License	Apache 2 license	Commercial license
Embedded, Remote and In-Memory Storage	●	●
Studio Visual Tool and Console Tool	●	●
High Availability Replication, Sharing and Fail-over	●	●
Standards: SQL, Blueprints, Gremlin, Pipes, SparQL	●	●
Teleporter (sync or migrate an RDBMS to OrientDB)		●
Metric Analysis (Monitor and tune your servers)		●
Query Profiler (Locate 80% of the bottlenecks in your application)		●
Configurable Alerts (Email alerts or executing HTTP requests)		●
Scheduler for Full and Incremental Non-Stop Backups		●
Auditing to track all the access to your data		●
10/5 & 24/7 Production Support Available		●

Figure 3.15: OrientDB editions comparison [99].



	Community	Bronze	Silver	Gold
License	Apache 2 license	Commercial license	Commercial license	Commercial license
Embedded, Remote and In-Memory Storage	●	●	●	●
Studio Visual Tool and Console Tool	●	●	●	●
High Availability Replication, Sharding and Fail-over	●	●	●	●
Standards: SQL, Blueprints, Gremlin, Pipes, SparQL	●	●	●	●
OrientDB Enterprise Features		●	●	●
Quick Priority Issue Resolution with SLA		●	●	●
10/5 Production Support Business days and hours**		●	●	●
24/7 Production Support Round-the-clock**			●	●
3 Months Developer Support included (2 seats)				●
	FREE for any use, even Commercial. No hidden restrictions. DOWNLOAD	2 Hours Activation Guaranteed*** I accept the Terms and Conditions* BUY NOW	2 Hours Activation Guaranteed*** I accept the Terms and Conditions* BUY NOW	Minimum 3 Servers, all included. Request a quote. GET QUOTE

Figure 3.16: OrientDB support subscriptions comparison [83].

expanded by following their outgoing or incoming edges, just by clicking on the buttons that compare around the selected node. Nodes can also be dragged so that the graph changes its aspect, or can be detached by the other nodes, or be hides. Nodes can also be edited in an interactive way, and it is also possible to create new vertexes and edges. Even for OrientDB, nodes and edges can be colored only according to some static information, so it is not possible to color in different ways the anchor nodes, or the nodes with more edges, etc, . If the query is not performed on the Graph visualization page, but on the Browse page, the result is given in tabular or JSON format.

Licensing

OrientDB offers a Community Edition under the Apache v2 license, and three levels of subscription for the Enterprise Edition, which is under a Commercial license. The main differences between Community and Enterprise editions are collected in Figure 3.15. They are focused on hot backups, data importing, monitoring, alerting, auditing and query profiling.

In Figure 3.16 are reported the differences among the different subscription levels for the Enterprise Edition. Subscription prices can be found on the official web site.

3.2.4 PostgreSQL

We now add a relational database to our comparison, so that we have a benchmark about the technology that dominated the databases field. We then cover, in a quicker way, the same topics presented for the previous three graph databases.

PostgreSQL is the well known object-relational open-source database management system (ORDBMS) based on the previous Ingres and Postgres projects which were developed at the University of California at Berkeley Computer Science Department during the years 1977-1994. Its ancestors brought great improvements in research for what concerns object and relational databases, and pioneered many concepts that became available in also other commercial database systems. It is one of the leading products in the relational databases field, as can be seen in Figures 3.1 and 3.2 [108, 59].

It is developed in C, transactional and ACID-compliant; it support referential integrity, allows to define stored procedures and triggers, supports and extends SQL, uses secondary indexes, and many other typical characteristics of relational databases.

PostgreSQL represents the *historical relational* figure in this work, used as reference point for comparing a relational solution with graph database solutions.

During this work, we will use the 9.6.1 version released in 26 October 2016.

Logical data organization

PostgreSQL organizes data as the classical relational model does, that is by means of *relations*, also called tables.

Generally, relations (with different names and fields) are used to represent those discrete entities that can be recognized in the dataset (persons, cars, animals, cities, etc); and each row for that relation contains data for one instance of such entity (Mark, Ferrari, Dolphin, New York, etc).

The database structure (that is, the set of tables, their fields and related data types) is defined by a schema, which also describes the presence of particular data constraints. The foreign key constraint is the one that realises referential integrity between rows of different tables; so relationships among relations' elements are controlled to be consisted by means of such constraints.

Relationships are effectively reconstructed at query-time by means of JOIN operations between tables, which are operations that perform a merge on those rows, belonging to different tables, that contain the same values on the fields over which the join is expressed. Since JOINs may be quite heavy operations - especially with tables with a big amount of records - some indexes should be constructed on those fields where JOINs are likely to be performed in order to speed up their execution.

One interesting aspect is that relationships are by nature bidirectional. However, for a fast enough reconstruction at query-time, indexes on both the source and destination fields of the JOIN should be defined. In fact, suppose we have a table for Persons with a field called *place_of_birth*, and that on such field it is defined a foreign key constraint that points to the primary key of another table, called Cities. It is simple and fast to retrieve their place of birth by performing a JOIN between *place_of_birth* and Cities' primary key, because the JOIN has been performed on a destination field already indexed, and an index defined on the field *place_of_birth* would not have brought any performance enhancement. However, if some kind of opposite query would be performed, that is for example "who are those persons that are born in Padua", the JOIN would the same way be performed on the same two fields, but the direction changes, and if the *place_of_birth* has not an index defined on it, a full scan of the Persons table will be executed, which generally implies performance decreases.

Physical data organization

PostgreSQL stores tables, indexes and any other support data by means of files organized in different directories of the file system. In particular, each of the tables and indexes is stored by means of at least one file located under the subdirectory of the database to which it belongs.

Different kinds of data are thus organized in different types of directories and files; the whole list can be found on PostgreSQL official documentation at the Section Database File Layout.

PostgreSQL realizes a disk-based storage and makes use of heap files for collecting tables' data.

Heap files are lists of unordered records of variable size; records are simply added according to their insertion order. More specifically, an heap file is structured as a collection of pages (or blocks), each containing a collection of items. The term item refers to a row that is stored on a page. To identify a tuple within the table, a tuple identifier (TID) is used; in particular a TID is made of a pair of values: the block and the offset number. The block number is the number of the page that contains the tuple; while the offset number is the number of the line pointer that points to the tuple. By organizing files in this way, the DBMS is able to retrieve its tables' records, even if they have variable size [33, 41, 46].

The indexes that can be defined above database tables are secondary indexes; that is sparse indexes that allows binary search with complexity O(log n).

Data Integrity

As known, PostgreSQL ensures both data integrity and referential integrity by means of constraints defined on their schema (not null, unique, primary key, foreign key, data types). Both insertion, update and delete operations are influenced by these constraints, and some behaviour specifications can be imposed in case of data modification on a record of a table with a foreign key constraint defined on one of its fields. Think for example about a delete operation: the behaviour can be different based on the specification given at foreign key creation. In fact, one possibility is to propagate the deletion on all involved records by using the CASCADE specification; another possibility is the NO ACTION specification, or the SET DEFAULT or the SET NULL one. So, thinking about graph consistency, referential integrity should be preserved by imposing such constraints. However, when some more complex logic is desired for data insertion, update or removal operations, transactions and triggers could come in help.

SQL transactions must be declared by enclosing them with the keywords BEGIN TRANSACTION and COMMIT. The ROLLBACK command is callable in case of unwanted situations emerged during the transaction execution. Furthermore it is possible to define savepoints within each transaction in order to allow a sequential progress and to give the possibility to rollback to such points while executing the single transaction [41].

PostgreSQL transactions exhibits ACID properties and uses MVCC and WAL techniques to achieve this and handle concurrency.

Query language and graph functions

PostgreSQL has nearly Full Conformance to the Core SQL:2011 standard [103]. SQL is both a DDL and DML declarative query language, mainly designed for working with the relational model and also providing some additional features useful for application purposes and database management [41].

SQL allows several grouping, sorting and aggregate functions; and allows to define also temporary tables for supporting complex query execution.

For what regards the use of SQL in a graph data domain, there is not a dedicated way to realize graph traversal with the sense that previous query language show, and this operations is typically simulated by exploiting existing statements.

The query could obviously be implemented by means of the classical SELECT statement. However, as explained in Section 2.5.3, this approach would make use of nested SELECT statements in other SELECT statements, with nested JOINs operations. This is not a problem when one already knows the depth level to reach, and also when such depth value is not very big (lets say, under 10 levels). However, for greater depth values, or in general when more than one depth value would be of interest, or again for unknown depth value at query formulation time, this approach is clearly not suitable.

One useful statement for overcoming such pitfalls is “WITH RECURSIVE”. This particular clause allows to recursively execute the query formulated within it, and gives the possibility to parametrically specify the depth value, which corresponds to the number of iterations the recursion will be performed. It thus comes useful when working with hierarchical or tree-like data.

“WITH” alone provides a way to write auxiliary statements for use in a larger query. These auxiliary statements, which are often referred to as Common Table Expressions (CTEs), can be thought of as defining temporary tables that exist just during the query’s life. Each auxiliary statement in a WITH clause can be a SELECT, INSERT, UPDATE, or DELETE; and the WITH clause itself will be attached to a primary statement that can also be a SELECT, INSERT, UPDATE, or DELETE. The optional RECURSIVE modifier changes WITH from a mere syntactic convenience into a feature that accomplishes things not otherwise possible in standard SQL. Using RECURSIVE, a WITH query can refer to its own output. The general form of a recursive WITH query is always a non-recursive term, then UNION (or UNION ALL), then a recursive term, where only the recursive term can contain a reference to the query’s own output [109].

The example provided in the official documentation is shown in Code snippet 3.8.

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
    SELECT g.id, g.link, g.data, 1,
      ARRAY[g.id],
      false
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
      path || g.id,
      g.id = ANY(path)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph LIMIT 100;
```

Code snippet 3.8: WITH RECURSIVE example.

So the clause allows to recursively work on the temporary table generated, which is populated during the simulation of the graph traversal. Note that some optimization mechanisms are suggested in order to limit eventual situations that could cause loops during the execution (in particular, path cycles).

Note also that the recursive query evaluation algorithm produces its output in breadth-first order; however results could be displayed in depth-first order by adding in the outer query an ORDER BY specification on the *depth* column.

PostgreSQL uses a query optimizer for the examination of each of the possible execution plans of a query; and ultimately selects the execution plan that is expected to run the fastest, which will be the one with least cost [104].

A useful property of WITH queries is that they are evaluated only once per execution of the parent query, even if they are referred to more than once by the parent query or sibling WITH queries. They may thus be used to avid redundant work when some expensive calculations are needed in multiple places. The side effect of this aspect is that the optimizer is less able to push restrictions from the parent query down into a WITH query than an ordinary sub-query. The WITH query will generally be evaluated as stated, without suppression of rows that the parent query might discard afterwards (apart from the case when a limit on the number of result rows is expressed) [109].

Caching

PostgreSQL uses two cache layers: "shared buffers" and OS page cache.

By exploiting the OS page cache, any read/write operations will pass through it; the DBMS writes data on OS page cache and, based on the *synchronous_commit* parameter configuration, the effective moment for the flush to disk may vary. In particular: if *synchronous_commit* is enabled, the DBMS will force the OS for the write to disk and will notify it to the application only after the actual execution of the operation has been performed; otherwise with *synchronous_commit* disabled the DBMS will notify the application as the write operation has already been performed, but its effective execution on disk will be managed by the OS, which will write to physical disk in its own pace. Because of this, typical recommendations are about having faster disks or better disk cache if *synchronous_commit* option is disabled [113].

What PostgreSQL saves within its *shared_buffers* are tables, indexes, and query execution plans. *Shared_buffers* is an array of 8KB blocks which kind duplicates what the OS does and is managed with a LRU/Clock sweep cache algorithm, which is built to handle OLTP workloads so that almost all of the traffic is dealt within memory. Because a plain LRU algorithm would not work well since it does not maintain historical memory of the previous runs, the DBMS keeps track of page usage count for each scan cycle of the array, so (explained superficially) if a page usage count is found to be zero or the page is "dirty", it is evicted from memory and written to disk.

Shared_buffers is independent from the contents of the OS cache, and both of them may hold the same pages at a certain time. This may lead to space wastage; however the main reason is that the OS may (as usual) only use a simple LRU algorithm, which works worse than the algorithm implemented with *shared_buffers*. However, once the pages take a hit on *shared_buffers*, the reads never reach the OS cache, and if there are any duplicates, they get removed easily, so the two cache regions will have different contents [35, 118].

Two configuration parameters for cache management are the so called *shared_buffers* and *effective_cache_size*. While the *shared_buffers* parameter sets the amount of memory the database server uses for shared memory buffers, the *effective_cache_size* should be set to an estimate of how much memory is available for disk caching by the operating system and within the database itself. More precisely, this value does not implies an effective memory allocation, and is used only by the query planner to figure out whether the plans it is considering would be expected to fit in RAM or not [107, 111].

PostgreSQL also realizes query plans caching, which generally lasts as the session in which queries are executed. Queries will be cached only if implemented with the PREPARE statement, or in PL/pgSQL [102, 105, 106].

In addition, no query results caching mechanisms are officially provided by the DBMS; even if a plug-in has been released for the purpose [110].

Scalability approach and data partitioning

PostgreSQL is not a distributed database, mainly due to its data model and storage techniques and because it must preserve ACID transactions, which in a distributed environment would become "less ACID".

PostgreSQL thus typically uses partitioning and vertical scalability for facing the problems caused by big datasets.

However, in order to go in horizontal scalability direction, it employs solutions that realize replication and load balancing with a master/slave architecture.

In any case, data sharding is still not supported by the DBMS.

Licensing

PostgreSQL is released under BSD license, which means that is can be used both for open-source and commercial purposes freely. In addition, PostgreSQL does not directly come with an Enterprise Edition of its product; however some third-party products have been implemented in order to meet some of enterprises needs. One such product is EnterpriseDB.

3.3 Feature matrix

In Table 3.1 is proposed a feature matrix that compares some aspects of the analyzed DBMSs.

The feature matrix highlights the chronological appearance of the four DBMSs on the market, displays the data models they implement, and shows that the three graph databases selected implement the *property graph* model for the representation of graph data contexts. They are native graph stores, which means that they purposely designed a way for handling graph data; however only Neo4j surely implements the *index-free adjacency* property. We decided to state that OrientDB "nearly implements" such property because, as explained in Section 3.2.3, the fact of being able to distribute data over different machines implies that some mapping operations are performed, based on information stored in dedicated files. ArangoDB instead do not implements such property, in fact it recovers adjacent nodes by means of indexes between edges and vertexes.

They are implemented with different programming languages, and all four DBMSs support secondary indexes. ACID properties are ensured on a single database instance for each DBMS, and referential integrity constraints are provided and checked when data is modified. We added a *sim* for ArangoDB and OrientDB because such guarantee is ensured when data is accessed by graph functions. Since they are multi-model, and since they allow to access graph data even with the functions used with generic documents, such guarantee may fall if such functions are used.

They all define some data types for the attributes given to the records, even if only PostgreSQL is a "only schema-ful" database.

They all come with a purposely designed query language, and they also allow to implement

Feature	ArangoDB	Neo4j	OrientDB	PostgreSQL
Category	NoSQL	NoSQL	NoSQL	Relational
Initial release	2012	2007	2010	1996
Database Model	Graph Document Key-Value	Graph	Graph Document Key-Value Object	Relational Object
Graph model	Property graph	Property graph	Property graph	–
Native graph	Yes	Yes	Yes	–
Index-free adjacency	No	Yes	∼Yes	–
Implementation	C++ JavaScript	Java Scala	Java	C
Indices	Yes, secondary	Yes, secondary	Yes, secondary	Yes, secondary
Transactions (Single instance)	Yes, ACID	Yes, ACID	Yes, ACID	Yes, ACID
Data scheme	Schema-less	Schema-less	Schema-less Schema-ful	Schema-ful
Referential integrity	∼Yes (edges)	Yes (edges)	∼Yes (edges)	Yes
Data typing	Yes	Yes	Yes	Yes
Query Language	AQL	Cypher	SQL "extended"	SQL
Stored procedures	AQL JavaScript	Java	SQL JavaScript Groovy	PL/pgSQL PL/Tcl PL/Perl PL/Python
Graph functions	Yes	Yes	Yes	No
Drivers	JavaScript Java PHP Python Perl .Net ...	Java C/C++ JavaScript PHP Ruby Python ...	Java JDBC JavaScript PHP Ruby Python ...	C/C++ JDBC PHP Ruby Python ODBC ...
Access methods	RESTful HTTP	RESTful HTTP Java API	RESTful HTTP Java API Binary	JDBC C API
Triggers	∼Yes (via FOXX Queues)	∼Yes (via Event Handler)	∼Yes (Hooks)	Yes
Concurrency	Yes	Yes	Yes	Yes
User concepts	Yes	Yes	Yes	Yes
Durability	Yes	Yes	Yes	Yes
Community license	Apache v2	GPL v3	Apache v2	BSD
Replication conflict resolution	Master/Master Master/Agent	Master/Slave	Master/Master Master/Slave	Master/Slave
Data sharding	Yes	No	Yes	No
Caching	Data Query results	Data Query plans	Data Query results	Data Query plans

Table 3.1: Feature matrix

stored procedures with more powerful languages. The graph DBMSs also provide some functions for working with graph data.

They all provide a lot of drivers for accessing the database with different programming languages, and in particular the graph DBMSs officially provide a RESTful HTTP API, so that databases are easily accessible by remote web applications. ArangoDB is the one that does not directly provide a native API to access its core methods.

Some sort of triggers are provided also by the graph DBMSs, even though they are not definable by the query languages provided.

They all manage concurrent queries, provide user and role definitions, provide durability to the data stored, and allow to configure a distributed architecture.

They all come with a version of the product released under a free license, even though Neo4j does not allow to use it for commercial purposes.

Only ArangoDB and OrientDB allow to perform data sharding on different machines, Neo4j and PostgreSQL don't.

They also provide different caching mechanisms, like query plan caching and query result caching.

Chapter 4

Use case

The objective of this chapter is to understand whether graph database technologies can be adopted in the context of relationships among companies.

In Section 4.1 we will present the domain of interest, i.e. the source of the dataset we will use in this work. In Section 4.2 we will give the representation of the dataset according to the ER model; we will also explain how the relational database by which the dataset has been extracted is implemented . In Section 4.3 we then present how the dataset can be represented with a graph. We will give the general best practices used to model data according to the *property graph* model, we will talk about the modeling choices taken, and we will give some numbers about the graph dataset used. In Section 4.4 we will present how data varies on the dataset given; and in Section 4.5 we will talk about some of InfoCamere's applications that could benefit by the adoption of a graph database.

4.1 The domain of interest

Our project aims to handle part of the data contained in the Italian Business Register, gathered by the Italian Chambers of Commerce and maintained for them by InfoCamere.

The Italian Chamber system (called "Sistema camerale") consists of 105 Chambers of Commerce (one for each province) and a number of other offices and associations. The mission of the Chambers of Commerce is to deal with the general interests of the production system; promote local development, market regulation and transparency; and to provide links between the business system and the Public Authorities. In Italy the Chamber system is represented by Unioncamere, the Italian Union of Chambers of Commerce, whilst at the European level there is the association of chambers Eurochambres [50]. Even other European countries have a system and a public business register like the Italian one.

The Italian Business Register ("Registro Imprese") is a public register which has been fully

Figure 4.1: Italian Chamber system map [50].

implemented since 1996. It is unique and unifies some ancient and separate registers. It can be defined as the register of company details: it contains all the main information regarding the companies (name, articles of association, directors, headquarters...) and all subsequent events that have occurred after registration (for instance amendments to the articles of association and company roles, relocations, liquidations, insolvency procedures, etc.). The data collected is about companies with any legal status and within any sector of economic activity, with headquarters or local branches within the country, as well as any other subjects as required by law.

The Italian Business Register thus provides a complete picture of the legal position of each company and is a key archive for drawing up indicators of economic and business development in each area to which it belongs [49].

The main function of the Italian Business Register is to ensure an organic system of legal disclosure for companies, ensuring the provision of timely information throughout the country. It also serves as a legal disclosure instrument for any official documents registered therein. The information and documents kept within the Italian Business Register may be consulted and acquired by anyone via the Chamber of Commerce counters, via official distributors, or via the Internet through the official site. All companies must be register in this register by law.

As of now, there is not a unified European business register that collects all data about European companies; however there is a service called EBR - European Business Register which provides real-time access to available information of the various business registers of the Member States participating to the initiative.

The Italian Business Register is implemented with a quite complex relational database in terms of tables number and constraints. Our work will not be based on all information stored within this database; it will be focused on data that contains companies' information and relationships among them based on "*equity participation*" / "*investments in an associate*". The information about the percentage of investment is present within the Italian business register, but will not be used during this analysis for privacy reasons.

In particular, based on the information stored, the two types of entities that can be recognized are: *companies* and *physical persons*. The reason why both these kinds of entity are present is because of the existence of different types of enterprise for the Italian law; and because equity participations can be hold by both individual persons and companies. The information available for such entities is:

- *codice_fiscale*: a unique identifier, representing:
  - codice fiscale: a national "fiscal code" that uniquely identifies every Italian citizen. It is made of a sequence of sixteen characters, which are digits and literals in a special ordering, and it encodes: name, surname, date, and place of birth.
  - partita iva: a national code that uniquely identifies every Italian company. It is made of eleven digits.

  Note that there is a special type of enterprise, the individual enterprise, which is identified by means of a codice fiscale rather then by means of a partita iva.
- *rea*: an identification code for the search on the REA register (Repertorio Economico Amministrativo).
- *cciaa*: a code relative to the Chamber of Commerce the company is registered in.
- *tipo*: a tiny code that differentiates person records from enterprise records.
- *denominazione*: the denomination - or business name - for companies; and the name for persons.
- *cittadinanza*: the registration country for companies, or the citizenry for persons.
- *natura_giuridica*: a tiny code representing the legal nature of enterprises.
- *tipo_impresa*: a tiny code representing the enterprise type.
- *capitale_sociale*: the amount of company's share capital.
- *numero_azioni*: the number of financial stocks.
- and three other fields, representing flags, for application purposes.

We anticipate that the information given is realized by two relational tables called "Ents" and "Soci". More details will be given in Section 4.2.1. *Ents* stands for *enterprises* and collects the tuples for both companies and physical persons; *Soci* stands for *business partners* or *associates* or *members* and collects the tuples for representing the participations.

Beginning from now, we will also refer to the table "Ents" with the terms "Enterprise" or "Companies" (treated as synonyms); while the table "Soci" with the terms "member_of" or "memberships".

4.2 Entity-Relationship representation

Within the dataset we have that both physical persons and societies (or corporates or companies) have participations on other societies. In addition, both the two entities have a unique identifier, a citizenry attribute, and a text attribute which corresponds to the name of a person, or the denomination of a corporate. So we can say that for each entity representing a corporate or company, the information stored is:

- partita iva
- rea code
- cciaa code
- denominazione
- cittadinanza
- natura_giuridica
- tipo_impresa
- capitale_sociale
- numero_azioni.

For each physical person instead, we can say that the information stored is:

- codice_fiscale
- denominazione
- cittadinanza.

So, by performing a sort of reverse engineering, we can say that the initial ER schema for our data domain could be the one shown in Figure 4.2. However, this schema has not already been subjected to the restructuration phase.

Within the schema shown, a common entity named Enterprise collects those attributes that are in common among its two specialized Entities. In addition, the specialization is complete and disjointed (then a partition), as shown by the specialization arrow filled in black. The identifier for such entity is represented by the *codice_fiscale* attribute.

The (recursive) Relationship is made of two attributes:

- *socio*: the value of the *codice_fiscale* attribute of the associate entity, i.e. the one that owns equity shares.
- *partecipata*: the value of the *codice_fiscale* attribute of the subsidiary company.

The restructured Entity-Relationship schema for this data domain is the one shown in Figure 4.3; in particular it corresponds to the schema for the already existing relational solution, by which data has been extracted.

The restructured schema has been obtained by the previous one by performing a single

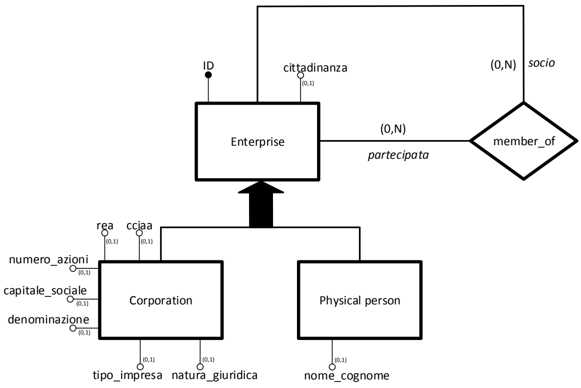


Figure 4.2: ER schema before restructuration

operation: the introduction of a discriminative attribute, called “tipo”. As known, this restructuration approach have some drawbacks related to the potential great number of NULL values for those fields that do not belong to one of the specialization entities; which in turn can come with performance decreases because of the increased number of records on a single table.

In the existing database, the information about the equity share - not reported here - is contained by the Soci relationship as a decimal value between 0 and 1.

Note that no multiple entities with the same *codice_fiscale* value are possible, because that attribute is the identifier for such entity. Furthermore, no multiple relationships between the same two entities (intended with the same direction of participation) are possible.

A relevant aspect is that none of the records that have the value “P” for the field *tipo* - which are those representing physical persons - can be participated by other records, so it cannot have members. This is because of law terms, and in fact it is intuitive that, while talking about physical persons, equity participations should not be possible on them.

4.2.1 The existing database solution

From now on, we will not refer to the Italian Business Register as the source relational database of our data, but to a derived one. This is the relational database used by *BRACCO*: an application that works on these kinds of data in order to extract in “batch-mode” the corporate groups present within the dataset. A short explanation will be given in section 4.5.

The extracted tables for BRACCO are a super-set of the scenario shown by the ER schema

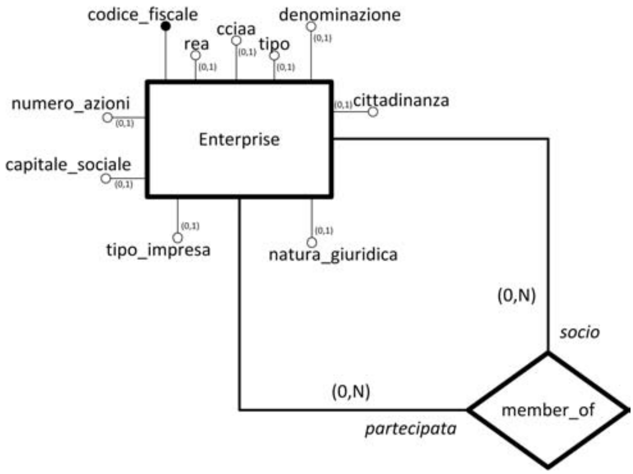


Figure 4.3: ER schema

seen before; however the main tables are exactly the ones we will work on, i.e. the table representing companies and the table storing relationships among them.

The relational database is based on MySQL; its tables' definition (for the two tables of interest) can be resumed like this:

For "Ents":

- *codice_fiscale*: the PRIMARY KEY and defined as char(16) NOT NULL DEFAULT ""
- *rea*: varchar(21) DEFAULT NULL
- *cciaa*: char(2) DEFAULT NULL
- *tipo*: varchar(2) DEFAULT NULL
- *denominazione*: varchar(305) DEFAULT NULL
- *cittadinanza*: varchar(3) DEFAULT NULL
- *natura_giuridica*: char(2) DEFAULT NULL
- *tipo_impresa*: char(2) DEFAULT NULL
- *capitale_sociale*: varchar(17) DEFAULT NULL
- *numero_azioni*: bigint(20) DEFAULT NULL
- three other fields representing flags defined as tinyint(1)

For "Soci":

- *socio*: char(16) NOT NULL DEFAULT ""
- *partecipata*: char(16) NOT NULL DEFAULT ""
- together *socio* and *partecipata* are PRIMARY KEY

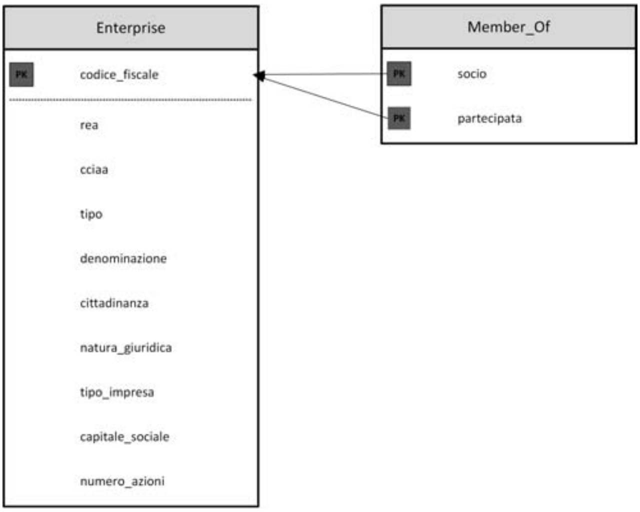


Figure 4.4: Logical schema for a relational database

The logical schema for this database is the one given in Figure 4.4. The same logical schema is also valid for the PostgreSQL solution that will be used during our comparison tests described in Chapter 5. An analog tables definition had be given to the schema definition of the PostgreSQL solution, with only some adjustments on data types definitions.

4.3 Graph representation

The data we will work on has a graph structure. This is understandable by both the cardinality constraints expressed in the ER schema (those on the branches of the recursive relationship); and some examples of the possible data situations. In fact, suppose we pick up a generic company among all the companies collected. This company can have either no equity participations, only one equity participation, or more equity participations on different companies. Suppose we are in the most general situation, that is the company owns more than an equity share of other companies. Now, the same holds for all the companies by which the starting company is an associate, i.e. every subsidiary may have zero or more participations. Suppose that more than two subsidiaries have in turn more than one subsidiary; here we have a graph made of equity investments among companies.

But why is this dataset a graph, and not a tree? Because nodes, in addition to the possibility of having multiple children, can also have multiple fathers.

In addition, direct cycles on a node - that is an edge that goes out and enters in itself - are valid situations. However, situations where the cycle is made of a directed path with more than one edge could represent an abnormal and interesting situation. Furthermore, by the primary key constraint on “Soci” table, it is clear that in this graph at most only one edge (for each direction) can be found between the same two nodes.

Another specification that follows from what has been said before is that only nodes representing enterprises will have incoming edges, while nodes representing a physical person will not. This is not assured by particular constraints defined on our database schemas, and in order to assure this kind of data consistency they may be imposed by the developer/application at insertion or update time.

4.3.1 Design Best Practices

When working with graph data, the main doubts that may emerge while defining the data model are about what should be treated and represented as a node, what as a property and what as an edge.

The main guidelines on such topic are:

- Nodes should be used for representing the discrete entities or facts that can identified within the data domain and that typically can be categorized or grouped.
- Properties should be used on nodes for describing the attributes of entities, both coming from the data domain itself and from the application.
- Edges should be used for representing the network of interconnections between entities and to establish semantic context for them. The created edges will have a direction, as of the majority of the use cases and for the property graph model. If the application requires to use edges without a direction, it is better to declare the bidirectional traverse at query-time rather than further creating a second edge with the same name that goes to the opposite direction in order to simulate a bi-directed graph.
- Properties should be used on edges for describing relationships attributes like edge weight, strength, quality, etc. Again, also these properties should be used for attaching meta-data to edges (like timestamps etc.).
- When in doubt among the use of a node or an edge for modeling something, check that such thing cannot be related to more than two other entities before choosing for the edge representation.
- If the entity property is a composed attribute, it might be convenient to model such attribute as a node instead of mapping all its components to the node it is attached to.
- The choice between the utilization of a node instead of a property is generally done by consistency and query reasons. In particular, suppose we have the information about the city where a company has a venue. We may then choose whether to represent such information as a property (a value own by those records that have such information); or as a node, with edges that point to it for all nodes with such information. The difference

between the two approaches is that with a dedicated node we are sure that two nodes that point to the same node refer to the same city; while with a property we loose such guarantee.

- A doubt may arise when choosing for edges modeling, that is whether a certain edge type should be specialized and distinct by one of its properties; or whether different edge types should instead be defined. For example, supposing we are modeling city companies' venues with a dedicated node, and that we want to store both the main venue and all its other venues for a given company, one possibility could be to have a single edge of type *Venue_in* with a property named *type* that could contain a value 'main' for the main venue, and 'secondary' for other venues; or instead have two separate edge types *Main_venue_in* and *Secondary_venue_in*. Fine grained edges bring enhanced performance, because unwanted edge types would typically not be looked at all when traversing only a certain edge type. However generic edges with properties qualification bring a simpler graph with easier formulation of generic queries. The suggested method in this case is typically the one with fine grained edges, because it reduce the amount of data retrieved and analyzed during a traversal.

- Even for nodes a similar doubt may arise; that is whether nodes - that could be distinct in two or more categories - should be defined with distinct labels/classes; or whether they should be defined with the same category and be distinct by a property defined on them. The straightforward representation would expect the use of different classes; especially where we are on a data model that can handle classes hierarchies (think about RDF ontologies).

- Complex attributes (like an address) may be represented by both complex data-structure properties on the node, or by dedicated nodes with simple data-type properties. The choice should be typically taken by the power provided by the DBMS for managing complex data-types, together with the desired structure for the graph.

Another interesting hint about graph modeling is that when facing complex graph domains, it is often possible to recognize more different and "independent" subgraphs connected by some kind of "joint" edges, as expalined before in Section 2.2. In this case, it could be take in consideration the hypothesis of separately modeling the two sub-domains by means of stand-alone graphs, and only in a second moment to link them by those binding edges previously identified. This also means that, potentially, data coming from different data sources and about different kinds of informative content may be merged in a unique graph. Obviously, the hard work would consists on having available the data about such "joint" edges, and on correctly managing data consistency and similar problems that arise when working with different data sources.

4.3.2 Logical design

Since our dataset has a graph nature, its representation with a graph is very straightforward: since we have equity participations among enterprises, enterprises are going to be represented by vertexes; participations by edges.

Vertexes, according to the "property graph model", will have properties representing the various fields listed before; edges instead do not own properties (at least in this analysis, however in case of effective deployment they will) because the information about the shares percentage is missing. However, the information about the participation direction is present, so we know which is the edge direction.

What remains to choose is whether the nodes should be distinct in different categories (Corporates and Physical persons); and consequently whether edges should be distinct in different categories according to the nodes connected by them.

Since the kind of queries we will formulate will handle both companies and persons the same way, we chose not to distinguish the two entities in two categories. In fact, our queries will not ask for searching something only among physical persons, or only among companies; both they will be equally taken in consideration.

The typical strong argument for performing a distinction among nodes is for performance enhancement during the "landing phase" of the query execution (see Section 2.6); however when we will present our queries we will see that we do not give any hint about the nature of the node, i.e. whether the *codice_fiscale* for the given node is about a person or a company; so the categorization of such nodes would not be directly used in the query formulation. Hence, despite of the fact that two separate and smaller indexes would be created on the two node categories, the query planner would generally have to look at both the indexes created for the individuation of the desired node. So in order to avoid maybe not so useful complications for the kinds of query we will perform, and since ArangoDB does not support queries on multiple or hierarchical collections [161, 21], in order to have equal situations for any DBMS analyzed we preferred to keep all the records in a single category.

In any case, eventual filtering operations during traversals can still be done by inspecting the value of the discriminative property at query-time; so nodes can equally well be distinguished by means of properties rather then on labels/collections/classes. However, the previously argument holds only in this particular case of hierarchical data classes; in the general use case, where queries can expressly suggest what kind of node to retrieve, the best choice is always to define nodes distinct in different classes.

Talking about edges, their modeling is straightforward, because all relationships represent the same concept: equity participations. In addition, during our queries, edges will all be treated the same way, both if connecting two corporates and if connecting physical persons and corporates. Note that connections between two physical persons are instead not possible.

One may think that edges between two corporates, and edges between persons and corporates, could be defined differently; however this would be only a stretch, because both represent the same concept, and there are not distinct properties for the two kinds of connection that could substantiate their effective differentiation.

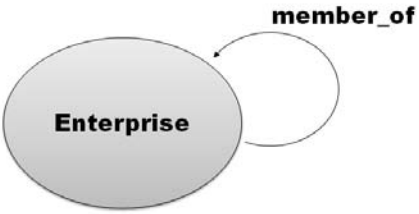


Figure 4.5: Graph data model.

In conclusion, the reality is simply modeled by defining “enterprises” (both companies and persons) as vertexes of the category *Enterprise*, while equity participations are represented by edges of the category *Member_Of*. It thus comes out that the graph data model is a lot similar to the relational model, which is made of a single table for enterprises and a single table for their relationships.

4.3.3 Some graph description metrics

Our dataset is made of about 10.5 million of entities representing companies and persons; and about 5 million of relationships among them. It is a snapshot of the situation of the Italian companies involved in this dataset at October 2016.

By comparing the two numbers, it is clear that the graph is not dense; in fact there are less edges than nodes, and by a high point-of-view, the ratio of nodes over edges is near 2, meaning that every node has (in average on all the graph) one edge connected, of any direction. The number of edges per enterprise is 1 and not 2 because every edge involves 2 (mainly different, except isolated cases) nodes; so the same edge is counted two times: one per vertex.

Apart from this high level average value, we will see later some cases that go far from it; for example we will see a node that has 1815 connected edges (both directions) and another node that has 6067 ingoing edges.

Obviously, there is also a great number of enterprises that are not linked to any other node.

For what regards paths length, they generically are not so big values because of the data nature itself. In fact, in a real situation of participations among companies, it should be quite singular to find paths that are hundreds of edges long. One of the longest paths we found was 29 edges long. Higher values would be possible if cycles are considered; however, if asking the length of a chain of companies, generally one would like to discard cycles and visit the same company along the path only once. Furthermore, the number given is based on the following assumption:

if there exist more than one path from a node to another, the most significant path is the one with minimum length. So (supposing there are not paths longer than that) 29 is not the length of the longest path on all the graph; it is the length of longest minimum path.

4.4 Data behaviour

Our test dataset is a static snapshot of what is stored in the relational MySQL database presented above; so it does not dynamically evolve during time. However, some information about the data behavior have been collected in order to understand how the dataset typically varies.

The first thing to consider is that, in terms of nodes number, our dataset can only grow. One of the fields of each enterprise record, in fact (which has not reported within our dataset) contains the information about the fact that such enterprise is out of business, i.e. it is closed down. Companies will never be deleted, and in case of closed down, the value of this field will be set properly; in this way the number of graph vertexes cannot decrease.

For what regards the number of relationships, instead, the same does not hold. The number indeed can grow because of new equity participations occurrences among new pairs of companies, or can the same way decrease because of their revocation and sale of relative shares. Of course, it can also remain the same because of lack of changes, or because the changes involved only the shares percentage values and maintained all the existing participations. However, in the case an enterprise closes, all relationships linked to it will be deleted. This are the policies chosen for the existing database, which clearly influence how data behave.

Talking about the rate of data update, we have to get a closer look on what is the data source for the current database (which, we repeat it, is the one used by the application called BRACCO).

The database is updated once per day during night, and rather than directly inspect changes occurred on the Italian Business Register, it exploits the title searches generated and collected at update occurrences on the register during the previous day. Title searches are XML documents containing all information about a company; and are those documents that are typically requested by clients interested in getting business information. This approach allows to reduce the load on the register database, and to cherish of the already performed computations, so that they are not performed twice.

The number of daily generated title searches gives an estimate of the number of updates that are performed on the extracted database used by BRACCO; such average number is typically between 25000 and 30000 per day. This updates may consists only of enterprise information, or also on equity shares; thus both “Ents” and “Soci” tuples may be updated by this process.

Please note that these are not necessarily updates of already present information, but can also represent new insertions of both companies or equity participations; or even companies closures or complete sells of equity shares.

For example, for the week that went from the 20th to the 25th of February 2017, the average value was of 23295 updates per day on the entire database, with a standard deviation value of 7795. For the single table about relationships, the average number was 1593, with a standard deviation value of 765. Over a longer period of time, that is the three months from January to March 2017, the number of records of the database increased by 115772: 115444 about anagraphical records and 328 about relationships records.

Since this dataset is an extraction of the official register, it is clear that queries for data modification on such database do not make any sense. Data updates have to be performed on the official data source; so this database should be designed in order to exclude write permission for its users, and thus it will act as a read-only data source.

4.5 Applications that could benefit from a graph database usage

InfoCamere is interest in exploring the field of graph databases in order to understand their potentialities for future projects.

However, there already exists some interest in a graph database approach for the application called *BRACCO*, dedicated to the Italian finance police. The main goal of this application is to bring a powerful and easy-to-use tool so that investigations on associations among companies can be performed, with particular attention for for what concerns the computation of the corporate group of a company, and the individuation of its controller - if exists - which is defined also by the amount of equity shares owned.

BRACCO is based on the relational database whose schema has been partially provided before, and whose data has been used to perform this investigation on graph databases.

In particular, it daily updates the relational database based on the title searches generated during the previous day by the execution of data modification operations on the business register. After the update process for the data stored in the database, for each of the "Ents" records updated it recomputes the corporate group for the given record by means of a batch job. The results computed by the batch job will be used for all the current day. In particular the application allows to get the precomputed corporate groups both in flat form (tabular output), and in graph form (by visualizing all nodes and relationships of the corporate group of interest). The graph visualization is helpful for the final user because it represents an intuitive and quicker way for exploring and maybe highlighting particular situations.

The process that computes the corporate groups makes use of an iterative algorithm implemented in Java. It works bottom-up, in particular for each row belonging to Ents which has been subjected to an update - i.e. for each updated enterprise record - it inspects the Soci table in order to obtain the information about its probably new associates or the new shares they own. The *codice_fiscale* values for such associates are then added to a list. Such list thus collects all the nodes that should be inspected in order to recompute the corporate group. The same operation is the applied to each of the nodes within the list. The process stops when no candidate nodes are present in the expansion list. This can be due to two situations: the first is that the node in exam does not have incoming edges (so it has not participations on it); the second is that the sum of all incoming edges' share values is not greater than 50%. Please note that this is a key condition for avoiding exploration and computations on useless graph areas.

We already explained that our dataset does not contain such information about equity shares, and we will not try to replicate the same use case. However, we will work on some queries with similar nature, such as queries that explores subgraphs. This is also the reason why we will not give a detailed explanation on how the corporate groups computation algorithm works; and more generally, why we will not have as goal the realization of the same goals of BRACCO.

Another of the InfoCamere applications that could be interested in such a change of the database adopted is "*ri.visual*".

Ri.visual has been developed over a relational database and presents to the final user the existing relationships among companies in a graph form. The graph nature of data, and the kind of queries that can be performed by this application, make the graph database an interesting solution for its data storage. In particular, its adoption would be interesting if it can offer the possibility to perform fast "multi-hop" interrogations without the need of storing or pre-computing graph patterns for their visualization. Also, the main interest could be given by the possibility to not only expand each single node with the nodes linked to it at one level of depth; but directly ask for all its ancestors, or only those within the depth specified, etc. In addition, shortest path queries or the search for the existence of any path between two given nodes would bring additional value to the current application.

4.6 Rationale for the choice of the graph DBMSs

The three graph database selected are ranked on top five positions on the DB-Engines.com Graph-DBMS ranking, provide ACID characteristics and CRUD operations, and an initial and quick investigation pointed out their good performance results. Furthermore, they all realize the *property graph* model, which is the desired one for the description of our graph data.

"*Why do we discarded a DBMS that only realizes the RDF model for a graph, i.e. a triple-store such as Virtuoso or Jena or GraphDB?*"

The main reasons are those expressed in Section 2.4.2; which are: the limited data model expressiveness for what regards relationships properties; and SPARQL's limited expressiveness for graph queries.

RDF is a powerful instrument with big potentialities, and some of its potentialities may be useful for other InfoCamere applications; however, for the kinds of query thought and for the current purposes, this solution has not been further analyzed.

"*Why didn't we choose Titan or Virtuoso?*"

The main reason behind the rejection of Titan is that it (or better the startup behind it, called Aurelius) was acquired one and a half year before the beginning of this graph databases exploration work. In particular, the new owner of Titan, DataStax, decided to stop its development and instead begin with the implementation of a new graph database product: DataStax Enterprise Graph, based on Apache Cassandra [37, 160, 82].

For what regards Virtuoso, it is a virtual database (VDB), which basically provides the ability to search across several databases with a single query. Virtuoso's VDB engine provides unified and transparent access to relational data residing on any ODBC/JDBC compliant DBMS. In addition the way by which it realizes the property graph model is with the RDF model, and the way by which it queries them is by using SPARQL [135, 134].

"*Why didn't we use graph analytics frameworks?*"

Graph analytics frameworks surely are products of interest for the field; however the work was intended to inspect products that can assure graph databases' persistence. In addition, the queries addressed by this work are not global-graph queries; are rather focused on graph traversal, which is the kind of queries that graph databases handle very well.

They anyway represent products that may be taken in consideration for future developments.

Chapter 5

Evaluation and comparison of Graph Databases

In this chapter we will describe both the preliminary phases and the performance tests performed. We start by describing, in Section 5.1, what are the principal aspects about our use case. In Section 5.2 we will present the environmental setup, both on hardware terms and on databases' configurations terms. In Section 5.3 we will explain the data import phase; and in Section 5.4 we will talk about warming-up the cache. Finally, in Section 5.5 we describe the performance results obtained from the execution of some selected queries.

5.1 Use case

In following sections we will apply some graph queries to the dataset presented in Chapter 4. We saw that the growth rate of the dataset is not so high; and the same holds for the update rate. The users of the application are financial police officers - so not a big crowd of different users - and the number of accesses per day is some hundreds. Furthermore, the database represents a read-only data source for the final users; in fact it can be only used for read-only queries. Update operations would arrive to the database only from the happening of update operations on the Italian Business Register, the official source of the data. In addition, the current dataset update method is asynchronous, i.e. it is done in bulk-way every night.

For these requirements, we understand that what we are mainly interested on the study of how the analyzed DBMSs respond when they are queried, rather than studying how much they are able to manage high-concurrent update or insert operations.

The main points of interest for our work are:

– the database would be used by an application that shows relationships among companies.

– the way by which the application would show such relationships is in a tabular and in graph manner.
– the database should be able to answer to the queries in hundreds of milliseconds, or however in time values that are comparable to the ones obtained by a relational database.
– the database should be able to handle some tens of million of records.
– the database should provide some mechanisms for checking the integrity withing the graph; so that we do not have dangling edges or other abnormal situations.

5.2 Experimental setup

All tests have been done during the internship in InfoCamere. For this purpose, one virtual instance of a RedHat Linux server has been used; such virtual machine was realized by one InfoCamere server dedicated to development purposes, and was accessed via a desktop computer through the intranet of the company.

We tried to define the test environment so that none of the DBMSs would be heavily privileged or penalized; both in terms of "hardware" configuration and in terms of their database server configuration. For this purpose, according to the three graph DBMSs, the suggested configuration is to have enough RAM to cache good part of the dataset and its indexes so that disk hits are minimized [22, 69, 94]. Obviously, another suggestion is to have fast disks (that is SSD instead of HDD) for reducing to the minimum I/O latency. However, since we had available a disk with HDD technology, we made sure that at least the requirement of not using a too small amount of RAM would be met.

Rather then testing these DBMSs with clearly poor hardware environments, and also maybe totally apart from a hypothetical configuration for a production product, we decided to follow their recommendations and give a discrete amount of RAM to the system, even though without exceeding. We however intentionally did not provide a "super-charged race car" with a lot of RAM, SSD technology for disks, and a CPU with enormous frequency values; so that our work presents an initial hint on what are the attention aspects of the machine tuning, and also to obtain typical results on moderate hardware.

The test machine specifics are:

- RAM: 6GB DRAM
- CPU: AMD Opteron 6276, x86_64, dual-core, single-thread per core, 64bit, 2.3GHz
- Disk: 100GB HDD
- OS: Red Hat Enterprise Linux Server release 6.8 (Santiago)
- JDK: OpenJDK 1.8.0_101 64-Bit Server VM (Java version 8u101)

ArangoDB server configuration

As for as the server configuration of ArangoDB is concerned, we used the default configuration; and it could not be done different, given the fact that there is no possibility to manually set configuration parameters about memory management. One parameter that we ensured to be disabled is the use of the query result cache, so that all databases are equally treated.

Neo4j server configuration

Neo4j has been left with its default configuration apart from the settings which regards the usage of the main memory; which has been configured according to the specifics given by the official documentation [69]. In particular, the parameters to be set are about sizing the Page Cache and the Heap Space. The configuration should be made so that the available RAM is split in such a manner that swap operations to disk are limited to the minimum, and hopefully never performed.

The configuration starts by defining the amount of RAM to be left to the operating system, in order to let it work quite freely. The amount of memory to be reserved for this purpose, as suggested by the documentation, is around 1GB plus the size of the files within *index/* and *schema/* sub-directories of the database main directory; which were around 300MB together after the data import. By the reduced amount of memory for such directories and since only one database service at a time would be active, and also since there will not be other processes launched by the user during the tests, the amount of RAM left for the OS has been sized to 1GB.

The next parameter regards the Page Cache, which is the amount of memory used to cache the data as stored on disk. The best configuration would be the one that reserves space for all graph data files. The memory consumption for such files is around 2.4GB, and the Page Cache has been sized to 3GB (the total size of the database is about 2.7GB).

The last parameter is related to the Heap Space, which should be large enough to sustain concurrent operations, and more in general query elaborations. The parameter corresponds to the definition of the JVM heap space by means of the *-Xms* and *-Xmx* arguments; which are suggested to be sized with the same value. Heap space has been sized with the remaining amount of RAM, which is 2GB. The documentation page talks about values like 16GB for its sizing, however our setting is also supported by the fact that no concurrent calls will be performed during our tests.

OrientDB server configuration

OrientDB has been left with its default configuration apart from the settings that regards the main memory usage. The parameters to be set are Heap Space and Disk Cache Buffer. For doing this, when the amount of available RAM is not in exceed, OrientDB suggests to assign less heap space and more disk cache buffer, which is the same idea at the basis of the Neo4j configuration [94].

Heap Space and Disk Cache have been set directly by the *server.sh* script, which is the one that launches the server process. There are not particular hints about how to size the two parameters based on the files' sizes; so an approach similar to Neo4j have been adopted. However, here the situation is different, because the total size of the database is about 6GB. The size of the files *enterprise.pcl* and *member_of.pcl* is respectively 4.3GB and 490MB; the size of *enterprise.cpm* is 131MB and for *member_of.cpm* is 66MB; the size of the index on the codice_fiscale property on the Enterprise class is about 1GB. So we have that not all the files can be completely entirely loaded in main memory.

The server configuration has thus been set to 2GB for Heap Space and 3GB for Disk Cache. For the OS it has thus been left 1GB of RAM.

In addition, the JVM parameter *-XX:+PerfDisableSharedMem* has been set, like suggested by the documentation, in order to avoid debug information writing about the JVM state [94].

PostgreSQL server configuration

For PostgreSQL, the only configuration parameters changed by their default values are *shared_buffers* and *work_mem*. The parameter *shared_buffers* has been enhanced by the initial 128MB to the 1450MB value, which corresponds to the 25% of the available RAM, as suggested by the official documentation [111]. In addition, the *work_mem* parameter has been enhanced from 64KB to 8MB, in order to give more RAM space for results sorting operations.

The *effective_cache_size* has been left to the 4GB default value, which is a valid value given the amount of RAM available in our case.

After the import, the database occupies 2.7GB on disk. The parameter *shared_buffers* has not been set higher than that because, by the way by which PostgreSQL also relies on the operating system cache, it's unlikely to find using more than 40% of RAM to work better than a smaller amount; as reported in [111].

5.3 Data import

Our dataset is composed of two CSV (Comma-Separated Values) files, generated by the extraction of the two relational tables in the MySQL source database. The first file contains 10.5 million anagraphical records about enterprises and weights 1GB; the second file contains around 5 million records about relationships and weights about 170MB.

Before talking about the way by which we imported data on the different databases, let's think about how data import should work. The CSV files we have are made this way: each row corresponds to a record and the first row is special because it holds the name of the table fields; ordered just as the corresponding values on the next rows. For the CSV file containing enterprise records we thus have the header row and a series of rows where the first value corresponds to *codice_fiscale*; after it there are other 12 values, where one of them is a *denominazione* value, which can be a small piece of text. For the CSV file containing the relationships, instead, we only have, for each record row, two strings corresponding to two *codice_fiscale* values. This is also the reason why the two files weight so much differently.

The linking information is thus realized by means of pairs of *codice_fiscale* values; as done by the relational database. However, depending on the database approach, it is not taken for granted that the import would simply implies the copy of the csv files with only little additional formatting or meta-data generation; rather, some operations may be necessary in order to construct the graph, especially for the creation of edges.

The typical import strategy for the bulk creation of a graph is to firstly load all data about nodes (from the CSV file containing enterprises' data); then maybe perform some operations like imposing constraints or creating some indexes; and then load data about edges (from the CSV file containing relationships' data), with all the operations that such activity may involve.

It is highlighted that, by the fact the dataset comes from a relational database with integrity constraints defined on its schema, it is automatically ensured that the data we are going to load do not contain problematic situations like relationships without one of the two references to the anagraphical records; or that the relationship connects an existing record to one that does not exist, thanks to the referential integrity constraint. This obviously avoids some of the problems that may arise while trying to load edges. Another avoided situation is the one where more than one edge "of the same type" is defined on the same ordered pairs of nodes, thanks to the primary key constraint on relationships records. In addition, also some problems on node records are avoided, like the fact that there would not be more than one record with the same *codice_fiscale* value; due to the primary key constraint defined on such field.

Let's now see how we imported data: all four DBMSs provide tools designed for the import of CSV files, and we used them to populate the databases.

ArangoDB data import

For ArangoDB the import has been done with *arangoimp*, which is a command-line tool provided for bulk import operations on JSON or CSV or TSV (Tab-Separated Values) files [13, 11]. The *arangoimp* utility needs to be invoked once for each import file; and the target collections can already exist or can be created by the import run [11].

ArangoDB asks that while importing the edge records, the values for the system fields *_from* and *_to* are already provided by the file to be imported. This has been accomplished by simply specifying that the first of the two values of the rows of the anagraphical CSV file corresponds to the *_from* field, while the second corresponds to the *_to* field. This is done by simply renaming the header row with such two field names. However, the values on following rows cannot be any values: they must be record identifiers within the collection of interest, i.e. values of the field *_key* for the vertex records already imported. So before importing the vertex records, the (optional) *_key* field and related values have been added to the CSV file by editing it. The values for such attribute have been taken from the *codice_fiscale* field. However, the values allowed for the *_key* attribute cannot hold some particular characters (like underscores etc. [19]) which were in turn present on some few special records (inserted for application purposes). So the values for the *_key* attribute are not simply copies of those belonging to the *codice_fiscale* attribute, but have been edited such that the particular characters would have been replaced by admitted ones, and also in such a way that such identifiers would remain unique. The editing of such a heavy file requires some time to be performed. Our approach was based on passing multiple progressive elaborations to the file by means of the Vim editor, and it required about 20 minutes of elaboration. However, there would surely be better solutions for it, like using alternative Linux commands or utilities (like the *sed* command) or by purposely defining the way by which the export of the relational database should be done.

One relevant aspect is that the usage of the importer tool does not require that a schema definition is already provided before the import of the data.

Resuming, ArangoDB simply loads data as they appear, and it just encodes it with the VelocyPack format. This turns out to be a fast import approach, in fact the time required to import the two collections is about 6 minutes: 4:30 minutes for enterprise records and 1:30 minutes for relationship records.

The graph however is not yet realized. As explained in Section 3.2.1, the named graphs are created by means of the "edges definition" procedure, which can be performed by the interactive shell (*arangosh*) or the web interface. This is however a very quick operation, made in some milliseconds.

Since ArangoDB already indexes the system attributes *_key, _from and _to*, and since we will use the attribute *_key* for identifying enterprises' *codice_fiscale*, no further indexes or constraints have to be created.

The total memory consumption for the database is about 4GB.

Neo4j data import

For Neo4j the import has been done by using the official *neo4j-import tool*, which is a command-line data importer. Such tool has been designed for bulk loading csv data and it should work better than the LOAD CSV Cypher command for such purpose [73, 67]. Some specifications on the header row of the csv files have to be given to the importer so that it knows what field to use as identifier, what labels to use for each record, and even what properties should be defined with data types different from string (e.g. INT for numbers etc.). In addition, some specifications have also to be given within the header row of the edges csv, so that the importer knows what to treat as the source of the edge (:START_ID) and what to use as the destination of the edge (:END_ID).

The tool expects that the database where data is going to be imported is empty; this implies that it is not possible to first import the enterprises nodes, create and index on them, and only after that to import the edges. Such operation should instead be possible if the import of the edges would instead be done with the LOAD CSV command. So the *neo4j-import* tool imported both the two kinds of records (nodes before edges) in a single import process. Such operation required about 6 minutes: 5 minutes for nodes and 1 minute for edges.

Again, also for Neo4j the "schema" definition is directly performed while importing data and does not require to be defined before such operation.

At the end of the import, a UNIQUE constraint (and hence an index, with the logic of Neo4j [65]) had been imposed on the property *codice_fiscale*. Again, such operation did not bring troubles because the unique guarantee was provided by the source of our dataset. The creation of the constraint required about 3 minutes.

The graph is thus created and no further operations have to be performed for the purpose.

OrientDB data import

For OrientDB various attempts have been done for a fast import of the dataset; which was done with the official ETL tool (Extract-Transform-Load). The ETL tool is a command-line utility which receives as parameter a configuration file in JSON format which defines the configuration variables and hence the way by which the ETL importer should load the given data. It accepts data in CSV, JSON and XML format, and also data source accessible with a JDBC connection [88].

A lot of parameters can be passed through the JSON configuration file, regarding how the source data should be read; how data should be handled or transformed (if necessary); and how they should be imported on the database (e.g. the class for the given record, etc). In order to do this, the parameters are grouped in five different sections: *config* for import process configurations, like the logging level etc; *source* for addressing the files to be loaded; *extractor* for the way by which such data files should be parsed; *transformers* for the specification of eventual intermediate operations on the data extracted; and *loader* for the definition of database-related options, like the connection type, the management of transactions and WAL, the usage of indexes, etc.

Rather then creating and updating the index while loading enterprises data (as done in some examples provided in documentation), it was decided to shift such operation once the complete list of enterprise records have been loaded; this was also decided based on the higher import speed shown. So the import had been split in: enterprises loading; index creation on the *codice_fiscale* property; and edges creation, which would then cherish of the defined index. In fact, OrientDB needs to search for the desired nodes in order to create the edge among them (as seen in Section 3.2.3); so for each relationship record, it searches for both the already loaded nodes and edits them (and the new edge record) so that the special fields which contains the references to the connected edges (or nodes respectively) are updated with the new reference values.

However, this approach seems to be very expensive. We made a big number of attempts for trying to understand why the import process was requiring so much time; however, even though transactions and WAL have been disabled and the flush to disk has been set every 10000 records, the times we found for the execution of such operations are the following: about 18 minutes for enterprises import; 12 minutes for the creation of a *property* on the *codice_fiscale* field (preliminary phase to the creation of an hash index over it; 27 minutes for the index definition; and 5 hours for edges creation.

The time required for the import of the edges is very big, also compared with the time required by the other DBMSs, even though the *plocal* connection protocol is used as suggested. We also tried another approach by using the Java API or the lightweight edges (even though they would not be usable for a future database employment because they do not allow the definition of properties on them) but without significant improvements. It seems like the index defined on Enterprise records is not well used with the import process. We hope to have made some mistakes while configuring such operations, even though the time spent on this topic has been quite relevant.

Even for OrientDB, and thus like the previous DBMSs, the schema of the classes containing the records is defined directly by the importer tool; so there is not need to define it before the import starts.

PostgreSQL data import

Before proceeding with the data import, a relational database requires that a schema is provided. It consists on the creation of the tables and the specification of the data types for each of their fields, with maybe also the definition of some constraints on them.

The approach followed for this purpose was to create the tables without the foreign key constraints and to define them only after the data had been imported. This approach is valid, again, because we are already sure that the constraint requirements would be met, given the source of the dataset. By defining constraints only after data have been imported, and not at each data to be inserted, permits better import performances.

The import of the dataset with PostgreSQL has been achieved by using COPY, which is a command defined on the extended PostgreSQL implementation of the standard SQL. First we loaded enterprise records; then we loaded the relationship records. However, with the approach adopted, we could have also swapped the import order without any trouble. The time required for the import of the enterprise records was about 4 minutes; while about 1 minute for relationships records. The two definitions of the foreign key constraints on the fields *socio* and *partecipata* of the table *membership* have been performed rapidly; in fact the first required about 40 seconds ant the second 20 seconds. In contrast, the import with the two constraints defined before the import required 7 minutes for importing relationships records. Note that the same approach may be also applied for the primary key constraint.

```
CREATE TABLE companies_and_memberships.companies (
    codice_fiscale char(16) NOT NULL DEFAULT '',
    rea varchar(21) DEFAULT NULL,
    cciaa char(2) DEFAULT NULL,
    tipo varchar(2) DEFAULT NULL,
    denominazione varchar(305) DEFAULT NULL,
    cittadinanza varchar(3) DEFAULT NULL,
    natura_giuridica char(2) DEFAULT NULL,
    tipo_impresa char(2) DEFAULT NULL,
    capitale_sociale varchar(17) DEFAULT NULL,
    numero_azioni bigint DEFAULT NULL,
    f_dich_controllo smallint DEFAULT '0',
    f_impresa smallint DEFAULT NULL,
    f_socio smallint DEFAULT NULL,
    CONSTRAINT codice_fiscale_pk PRIMARY KEY (codice_fiscale)
);

CREATE TABLE companies_and_memberships.memberships (
    socio char(16) NOT NULL DEFAULT '',
    partecipata char(16) NOT NULL DEFAULT '',
    CONSTRAINT socio_partecipata_pk PRIMARY KEY (socio, partecipata)
    --, CONSTRAINT socio_fk FOREIGN KEY ( socio ) REFERENCES
       companies_and_memberships.companies ( codice_fiscale ) ON DELETE
       CASCADE,
    -- CONSTRAINT partecipata_fk FOREIGN KEY ( partecipata ) REFERENCES
       companies_and_memberships.companies ( codice_fiscale ) ON DELETE
       CASCADE
);
```

In addition, we need indexes so that the queries are performed in reasonable times. The primary indexes for the two tables already provide an index which can be used during queries. However, we also need to index the fields *socio* and *partecipata*, so that, given a *codice_fiscale* value which on some of our queries would correspond to the value of *socio*, we are able to find it quickly and, in turn, to find out quickly also all the *partecipata* values attached to it; and vice versa.

So, two additional indexes have been created, and the times required were about 20 seconds on *socio* field and 50 seconds on *partecipata* field. The two indexes are of the kind B-Tree, like the index constructed with the primary key constraint.

5.3.1 Summary

Operation	ArangoDB	Neo4j	OrientDB	PostgreSQL
Enterprises import	4min 30s	5min	18min	4min
Relationships import	1min 30s	1min	5h	1min
Indexes creation	1min 30s	3min	39min	2min 10s
Database memory consumption	4GB	2.7GB	6GB	2.7GB

Table 5.1: Data import comparison.

We resume in Table 5.1 a conclusive overview on the import phase. We can see that ArangoDB, Neo4j and PostgreSQL are quite aligned on required times, even though for what concerns disk consumption ArangoDB requires more space. In addition, remember that ArangoDB keeps indexes on RAM, so their size is not counted here. OrientDB instead require both more space and time.

5.4 Cache warm-up

Our queries will be performed on hot cache; where with hot cache we intend the fact that we made sure that the data files which would be used to answer to a query are (likely) already present in main memory; so that disk accesses are highly reduced. Such approach respects what is suggested by the some of graph DBMSs analyzed [43, 147]. In a production scenario, one should try to avoid eventual penalizing situations which could be removed quite simply; so the warmed-up cache is an assumption that is simply brought to reality by launching a particular query at the each start-up of the server.

Some differences in execution times had been observed between warming-up the cache before starting querying, and working on data right from the start.

Such differences are typically due to the necessity of I/O operations for transferring data from disk to RAM, which penalize the first query executed on a given node / graph region.

The warm-up assumption is further supported by the fact that ArangoDB needs to construct indexes after the start-up for those collections involved by the queries. This implies that the first query executed would bring abnormal execution times only due to such preliminary operations, rather than measuring the effective times required to retrieve data and answer to the query.

The cache warm-up is typically done by means of queries that fully scan all records [43, 149] (as done with ArangoDB, Neo4j and OrientDB), or by means of a predefined function (as done with PostgreSQL).

The purposely implemented warm-up query for ArangoDB is the one reported below:

```
LET vertices = ( FOR vertex IN Enterprise RETURN vertex._key )
LET edges = ( FOR edge IN member_of RETURN edge._key )
RETURN 1;
```

It can be seen that full scans on the collections containing vertexes and edges would be performed. We do not know if this is the most efficient way to force indexes construction and also transfer data on RAM; the official documentation do not give hints about it.

The following query is the one proposed by Neo4j documentation for warm-up purpose:

```
MATCH (n)
OPTIONAL MATCH (n)-[r]->()
RETURN count(n.prop) + count(r.prop);
```

The property *prop* is fictitious and is used to force the optimizer to search for a node or edge with a property named like that, so that all nodes and edges, and relative properties, are accessed.

Another way to perform the warm-up is by means of a stored procedure provided by the APOC package [8], which displayed minor execution times for the same purpose on some quick tests performed in a second moment.

The implemented warm-up query for OrientDB is the one reported here:

```
SELECT COUNT(codice_fiscale)
FROM Enterprise
LIMIT 11000000
```

It only loads nodes because other attempts on loading edges and connected nodes gave some java.lang.OutOfMemoryError exceptions. OrientDB does not provide a suggested query for such purpose, and also does not talk about warm-up as a suggested operation.

For PostgreSQL the *pg_prewarm* function has been used, which was called on both tables for their loading in main memory.

```
SELECT pg_prewarm('companies');
SELECT pg_prewarm('memberships');
```

The times required to perform the warm-up is shown in Figure 5.1. In particular, we show the average value of twenty executions; with a confidence interval that corresponds to the standard deviation value.

Some tests have been done to compare the times obtained with and without the warm-up: OrientDB did not shown benefits by a warm-up performed with such query; while the

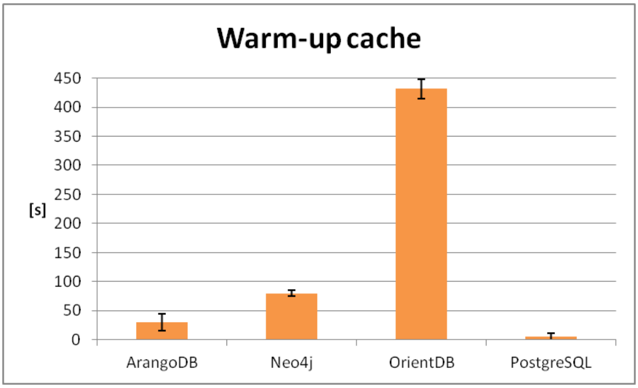
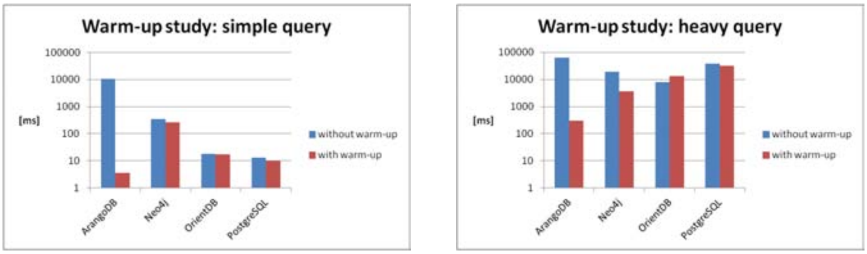


Figure 5.1: Warm-up cache, times chart



(a) Comparison on a light query (b) Comparison on a heavy query

Figure 5.2: Warm-up quick study

others shown benefits in query execution times. For example, Figures 5.2a and 5.2b show how performance changes with and without warm-up in two sample queries where complexity varies. As it can be seen, ArangoDB clearly shows big differences on execution times because it has to build indexes. Neo4j and PostgreSQL show some benefits. OrientDB instead seems to not cherish from the warm-up implemented with such query; in fact, in one case it shows reduced time with it, but on another case it performed worse. Please note that the time values are given in logarithmic scale. Anyway, such results should only be taken as indicative of the size of time required for such phase.

As an estimate of the memory usage for such phase, we looked at the results of the *top* Linux command. The resulting consumption is shown in Figure 5.3. By looking at the chart, and by considering how ArangoDB works, it is clear that ArangoDB is the DBMS that more needs RAM for working. While the other DBMSs store indexes on disk and load them (or portions of them) to RAM when needed, ArangoDB would like to construct and keep its indexes entirely in RAM. On one hand, it is true that indexes typically weights less than the amount of data they are defined on, and thus do not add so much load to the system with respect to the one already

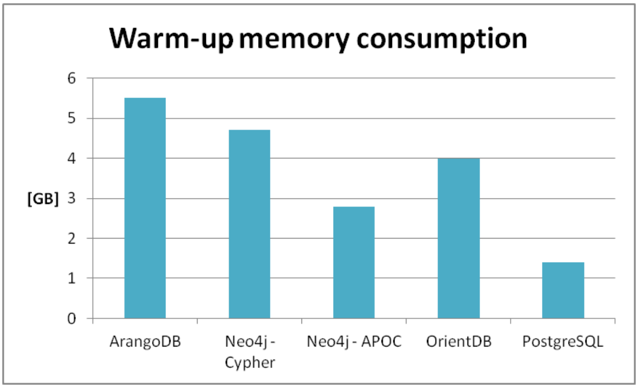


Figure 5.3: Warm-up cache, memory chart

due to data itself. On the other hand, it is clear that on very large datasets, this approach would cause performance degradation if indexes do not entirely fit in main memory.

5.5 Performance comparison

In this section we will present the queries we used for the performance comparison of the various DBMSs involved. For each query we will explain what data it searches; how it is implemented; and we will show some charts that show the time required for their execution.

During the work, several queries have been implemented and tested for the four DBMSs. Here we will report some of the most significant ones.

The queries we will see can be distinct in four categories: those about the search of one specific node; those working just on a node's neighbours; those that constitute multilevel traversals starting from a given node; and those involving two nodes. Such query typologies were chosen because we are interested on searching for the nodes that are somehow linked to the given node; and also for reasoning about the troubles that would come by realizing such queries via SQL in a relational database.

The queries had been implemented by means of the official query languages that the DBMSs provide; thus for ArangoDB we used AQL; for Neo4j we used Cypher; for OrientDB we used its extended SQL; and for PostgreSQL we used its implemented version of SQL. The reasons for such choice are:

- they are the most documented way for expressing queries by the official documentation sites;
- they are the suggested way of querying according to the DBMSs themselves;

- we want to understand peculiarities and differences among such query languages; in particular by comparing them with SQL;
- the majority of the questions asked on coding forums (like StackOverflow etc) seemed to be about these form of query implementation.

We then avoided queries' formulation by means of the native APIs right from start - or by means of procedures developed with the provided programming languages - because we think that the first approach should be made by means of the query language that is suggested by the database itself (which is also the solution typically subjected to the best optimization). In any case, the implementation of the queries directly from the native APIs or similar approaches would not have brought for us particular simplifications, given the fact that each API is written differently, and thus we would the same way have to study four different "languages". In addition, provided query languages are typically declarative and hence more intuitive then the other ways for expressing the queries. One "simplifying" possibility for queries' implementation would have been the usage of the Gremlin API from TinkerPop: by using it, the same query would have been valid for more than a graph DBMS. Both ArangoDB, Neo4j and OrientDB support this "industry" and "unofficial" standard; even though only OrientDB directly supports it without the need of additional libraries or plug-ins. However, such common characteristic has been discovered only later during the study, and since it requires additional libraries, and also because its usage would go away from the solution which is optimized by the DBMS (i.e. the provided query language); such queries implementation had been discarded. In addition, a quick analysis displayed that traversal constraints about nodes or edges visiting cannot be expressed.

Each query typology will be performed on three different nodes. Such nodes were chosen so that there is one node that represents "light-weight" workload for such query; one that represents an intermediate value; and one that represents a "heavy-weight" situation. For example, for the query about retrieving the direct neighbours of a node (*get_direct_neighbors*), the light situation is given by a node that has few edges connected to it; the medium case is given by a node with a good number of neighbours; the heavy case is given by a node that has a very large number of direct connections. However, since the light-weight situation for a query may not be light-weight for another, a node is not inherently light for all the query types; and thus the nodes may vary among the queries.

Note that, in some cases, we will refer to the node (or better, the *codice_fiscale* value which identifies it) as "small" for the light-weight situation; "large" for the heavy situation; and "medium" for the intermediate case.

Thee approach followed for the execution of the queries is the following:

- only one server at a time is up and working;
- the execution is managed by means of bash scripts purposely implemented, which launch the queries through the official shell tools provided;

- queries are launched sequentially, so there are not concurrent queries;
- we collect the times for the execution of the query at server-side rather then at client-side, so that network transport times are discarded;
- we perform a warm-up of the cache for all the DBMSs before starting querying them, in order to have an equal start situation for every database;
- we collect 20 samples per node for each query type, so 60 samples per query type;
- the 60 samples are taken consecutively: 20 for the first node, 20 for the second and 20 for the third;
- after the same query have been executed, the service is restarted and the warm-up is performed again before passing to the new query typology.

Such approach allows to get an estimate of both the time required by the first time a given query is executed, and by the time required at operating speed. In particular, we will see that there is notable difference between the times for the first execution and the following executions. We also quickly tested the execution of the queries in a cyclic manner, that is by changing the query (type and node) at each time without shutting down the server for 20 times for the same query, but the execution times displayed very close values.

We now see which are the queries used for the performance tests. The node '00102681419' will be used to graphically explain the queries. We will not show the queries' results because of privacy restrictions on the dataset; in addition, eventual *codice_fiscale* values have been substituted by fictitious values.

The collection of implemented and tested queries is the following:

- *single_match_by_cf*
- *get_direct_fathers*
- *get_direct_fathers_only_cfs*
- *get_direct_neighbors*
- *get_distinct_descendants_and_level*
- *get_distinct_descendants_and_level_only_cfs*
- *count_distinct_ancestors_per_level*
- *get_common_descendants*
- *get_common_ancestors*
- *get_shortest_directed_path*

In particular, for some of the queries, we implemented a version that only returns the *codice_fiscale* value (instead of the values for all the attributes) in order to analyze how much the retrieval of all values influences the execution time of the query.

In addition, some quick queries about global graph information computation have been implemented, like the search for root nodes, or the count of the outgoing edges for each node, or the search for the diameter of the graph, etc. However, as already stated, graph databases are not the right instrument for doing such computations; and in fact their executions were requiring so much time that we aborted them.

5.5.1 *single_match_by_cf*

The query searches for the particular node which is identified by the given *codice_fiscale* value; it thus corresponds to: "search the enterprise that has this *codice_fiscale*". The database will thus return a single node record and should exploit the index defined on such property for the resolution.

With ArangoDB's AQL the query is written as shown below:

```
LET search_cf = '00102681419'
FOR v IN Enterprise
  FILTER v._key == search_cf
  RETURN { 'vertex': v }
```

As explained in Section 5.3, we mapped the value of the *codice_fiscale* attribute to the *_key* property, which is already indexed, so rather then using a new index for searching on the *codice_fiscale* property, we perform the search directly by the identifier. It can also be noted, by looking at the RESULT statement, how AQL allows to handle and force in a simple way a JSON structure for the results of the query. Note that, rather then defining the *codice_fiscale* value by means of a LET statement, we could simply have written it on its place of use.

With Neo4j's Cypher the query is written as shown below:

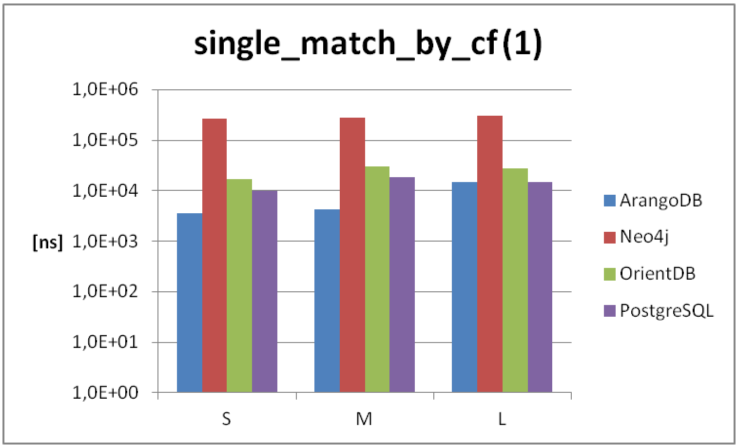
```
MATCH (e:Enterprise {codice_fiscale: '00102681419'})
RETURN e;
```

Cypher does not use the SELECT keyword, typical of SQL, but substitutes it with the MATCH keyword, because it is used to match graph patterns.

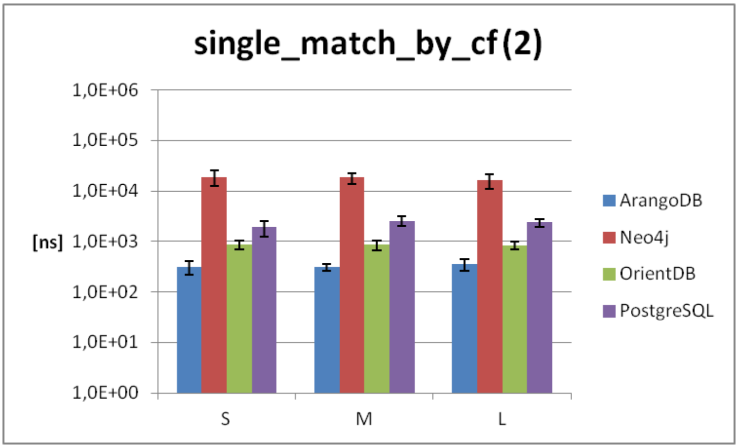
In OrientDB's Extended SQL the query is written as shown below:

```
SELECT *
FROM Enterprise
WHERE codice_fiscale = '00102681419'
```

which is "identical" to the query implemented for PostgreSQL:



(a) First query



(b) Following queries

Figure 5.4: Query single_match_by_cf charts

```
SELECT *
FROM companies_and_memberships.companies
WHERE codice_fiscale = '00102681419'
```

The times required to answer such query for the three selected enterprises are shown in Figure 5.4. Time values are displayed in a logarithmic scale so that both small and big values can be seen clearly. Note that even in this case we tested the query on three values, although the classification in small/medium/large does not hold because, being all the values indexed, any value brings the same workload to the database.

The obtained execution times show that, for any DBMS, the first execution of the query requires way longer than the following ones. Such behaviour was expected in absence of warm-up; however it is shown even in such case. In order to better study the two situations we preferred to split the chart in two sub-charts, where in the first one we show the time required by the first execution, and in the second one we show the average time and the standard deviation value for the remaining executions. In this way we avoid extremely big error bars due to the great difference between the first execution time and the following. In addition, with this method we better get an estimate of the different times required by the two situations.

We thus see that, for this quite "standard query", the time required is typically little and quite aligned among the DBMSs, except for Neo4j which takes generally more than the others both during the first query and the remaining ones. Anyway, for Neo4j the time went from 260-300ms for the first query to 16-18ms for the average value of the following ones; so there is great difference in execution times. The other DBMSs also shown a reduction of about an order of magnitude between the time of the first query and the average value of the following. There seems to be some room for improvement on this kind of query for Neo4j; the study of a different data model design maybe would have brought better results.

From the chart we can also appreciate the fact that right from the 2$^{\text{nd}}$ execution of the query, to the 20$^{\text{th}}$, the execution times are very similar; as can be seen by the short error bars.

5.5.2 *get_direct_fathers*

With this query we ask for the nodes which are direct fathers of the node identified by the given *codice_fiscale*. This corresponds to the search for "the enterprises which own equity shares of the given enterprise". The situation, for the example node of *codice_fiscale* '00102681419' is the one shown in Figure 5.5 which has been taken by the query result viewer provided by the web interface of Neo4j.

With AQL the query is implemented as shown below:

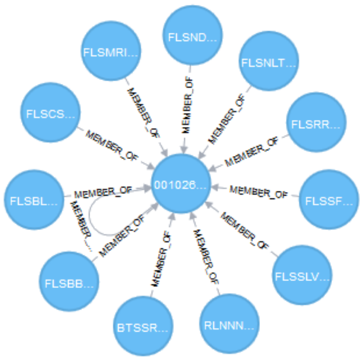


Figure 5.5: Fathers of the node with *codice_fiscale* = '00102681419'

```
LET startVertex = 'Enterprise/00102681419'
LET minLevel = 1
LET maxLevel = 1
FOR v
   IN minLevel..maxLevel
   INBOUND startVertex
   GRAPH 'enterprises_and_members_Graph'
   LIMIT 200000
   RETURN { 'vertex': v }
```

Here we see the AQL Traversal query structure cited in 3.2.1. In particular we search among the vertexes by following "inbound" edges, with the max depth value set to 1. The string that follows the GRAPH keyword is the name of the *named graph* defined by the collection that contains equity participation edges. Since our dataset will not have more than an edge defined on the same node pairs with a given direction, we are sure that each father node is already distinct. However, forcing the check can be done by adding the DISTINCT keyword after the RETURN keyword; or by defining an additional OPTIONS row with a unique constraint parameter for all vertexes visited. We will see such topic on following queries. Finally, we impose a very high LIMIT value just to stop the query when it seems to be retrieving too much nodes.

With Cypher the query is written as shown below:

```
MATCH (startNode:Enterprise {codice_fiscale:
    '00102681419'})<-[m:MEMBER_OF]-(father:Enterprise)
RETURN father
LIMIT 200000;
```

In Cypher we search for a pattern made of an edge of the kind 'MEMBER_OF' that connects the anchor node ("startNode") with another node ("father"), and we return such father node. Again, checking that there are not duplicate nodes in the result set can simply be done by adding a DISTINCT keyword after the RETURN keyword.

With OrientDB's SQL the query is implemented as shown below:

```
SELECT expand(in('member_of'))
FROM Enterprise
WHERE codice_fiscale = '00102681419'
LIMIT 200000
```

which is very similar to a standard SQL query. The *in()* function is the one that returns the nodes reachable by following incoming edges in the opposite direction. The *expand()* function is called on such nodes in order to obtain the values for all the properties of the nodes retrieved. The *in()* function, indeed, returns the RID values for the records on interest, and not the full list of records' property values. The DISTINCT keyword may be used to force uniqueness among results as general SQL; however when a list of results is expanded like here, the DISTINCT does not work. An alternative method is by using the MATCH statement; with it the query would become like shown below and results uniqueness would be automatically granted:

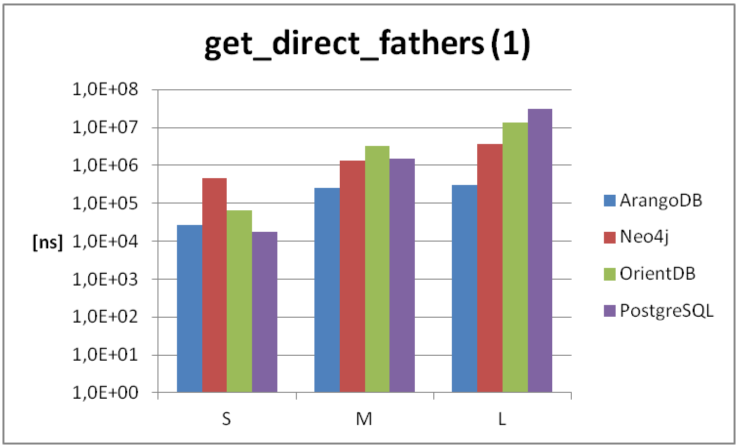
```
MATCH {class: Enterprise, as: start_node, where: (codice_fiscale =
    '00102681419')}.in('member_of'){as: father}
RETURN father
LIMIT 200000
```

However, by the nature of our dataset, we used the query implemented with the SELECT statement for performance tests.

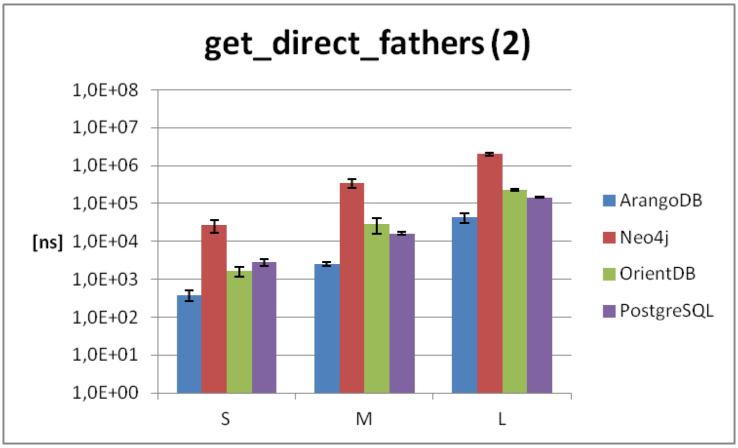
In PostgreSQL the query is written as shown below:

```
SELECT *
FROM companies_and_memberships.memberships AS memb
    INNER JOIN
    companies_and_memberships.companies AS comp
    ON memb.socio = comp.codice_fiscale
WHERE partecipata = '00102681419'
LIMIT 200000;
```

We start the query directly on the table containing the relationships among enterprises, and we then need to perform a JOIN with the anagraphical table in order to retrieve the values for



(a) First query



(b) Following queries

Figure 5.6: Query get_direct_fathers charts

all the fields. Again, the DISTINCT keyword is not needed, however it could be used on the right of the SELECT keyword, and may be defined only on the *codice_fiscale* field by using the DISTINCT ON (...) clause.

The number of result nodes for the enterprise with low number of direct father (S) is 4; for the intermediate case (M) the number is 418; for the case with a big number of nodes involved (L) is 6067. The times required to resolve such query for the three situations are shown in Figure 5.6. We see that ArangoDB needs less time than the others both for the first and for the following executions; we also see that there is not a big difference on the required times by PostgreSQL. In fact, here we are asking to traverse edges only a single time, and this is done well by a relational database because we already start the query on the relationships table; we only have to detect those *partecipata* values linked to the *socio* given, and then perform a JOIN for such found values.

In the second chart we see that ArangoDB goes under the millisecond value. It could seems that the result caching mechanism is applied, however we checked and it was not turned on. Also, in the second chart, the bars for PostgreSQL show comparable execution times to the times required by the graph DBMSs for both the small, medium and large amount of results retrieved.

5.5.3 *get_direct_fathers_only_cfs*

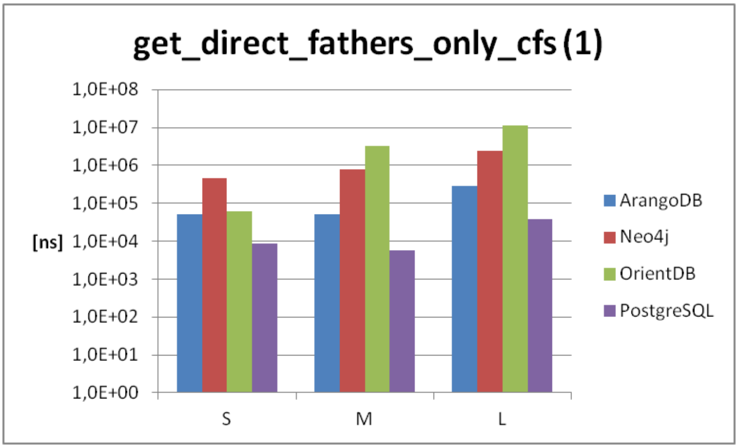
This query is analogous to the previous one, however this time we only return the *codice_fiscale* values of the reached nodes. This is done to study how the lightening of the results affects the execution time.

In ArangoDB the query is implemented as shown below:

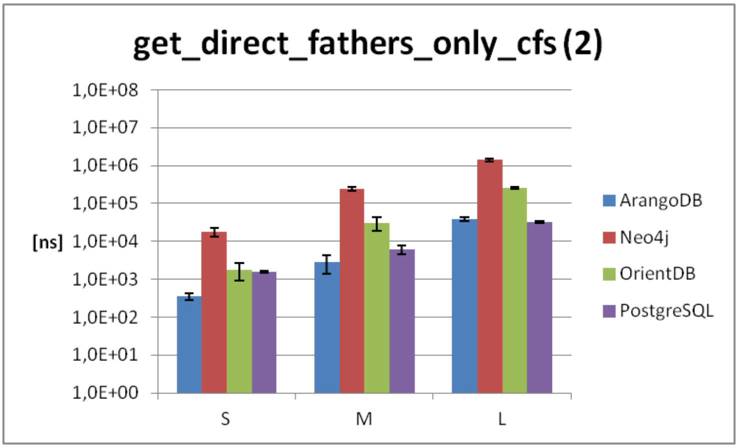
```
LET startVertex = 'Enterprise/00102681419'
LET minLevel = 1
LET maxLevel = 1
FOR v
  IN minLevel..maxLevel
  INBOUND startVertex
  GRAPH 'enterprises_and_members_Graph'
  LIMIT 200000
  RETURN { 'vertex': v.codice_fiscale }
```

The limitation on the property values to return is done within the RETURN statement, as last operation.

In Neo4j the query is implemented as shown below:



(a) First query



(b) Following queries

Figure 5.7: Query get_direct_fathers_only_cfs charts

```
MATCH (startNode:Enterprise {codice_fiscale:
    '00102681419'})<-[m:MEMBER_OF]-(father:Enterprise)
RETURN father.codice_fiscale
LIMIT 200000;
```

Even with Cypher the query remains the same and it changes only within the RETURN section.

In OrientDB the query is written as shown below:

```
SELECT in('member_of').codice_fiscale
FROM Enterprise
WHERE codice_fiscale = '00102681419'
LIMIT 200000
```

The only change with respect to the previous query version is the selection of the property of interest at SELECT time.

In PostgreSQL the query is implemented as shown below:

```
SELECT socio
FROM companies_and_memberships.memberships
WHERE partecipata = '00102681419'
LIMIT 200000;
```

This time we do not need to perform a JOIN with the table containing enterprises information.

The times required to resolve such query for the three enterprises are shown in Figure 5.7. The query has obviously been executed on the same three *codice_fiscale* values of the previous case. It can be seen that this time PostgreSQL works way better than before and always better than the graph DBMSs on the first query execution. We see also that again ArangoDB seems to have a very pushed form of caching, which allows it to have such execution times at operating speed after the first execution.

5.5.4 *get_direct_neighbors*

The query searches for the enterprises directly connected to the enterprise given, regardless of the direction of the participation edge. It thus collects only the nodes reachable by traversing a single edge. The example with the node '00102681419' is shown in Figure 5.8.

With AQL the query is written as shown below:

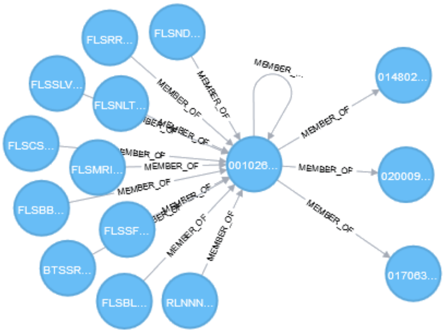


Figure 5.8: Neighbors of the node with *codice_fiscale* = '00102681419'

```
LET startVertex = 'Enterprise/00102681419'
LET minLevel = 1
LET maxLevel = 1
FOR v
   IN minLevel..maxLevel
   ANY startVertex
   GRAPH 'enterprises_and_members_Graph'
   OPTIONS {uniqueVertices : 'global'}
   LIMIT 200000
   RETURN { 'vertex': v }
```

The query remains similar to the previous one, it only changes the INBOUND word with ANY.

With Cypher the query is written as shown below:

```
MATCH (e:Enterprise {codice_fiscale:
     '00102681419'})-[m:MEMBER_OF]-(neighbor:Enterprise)
RETURN DISTINCT neighbor
LIMIT 200000;
```

Again the query is very similar to the previous one. The lack of direction is encoded by the absence of the angular bracket which denotes the edge direction.

In OrientDB the query is implemented as shown below:

```
SELECT expand(both('member_of'))
FROM Enterprise
WHERE codice_fiscale = '00102681419'
LIMIT 200000
```

Here we force to traverse edges in both directions by invoking the function *both()*.

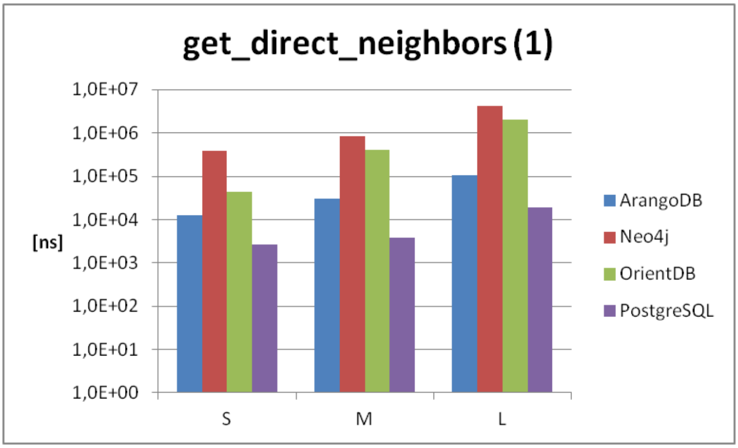
In PostgreSQL the query is written as shown below. Two temporary tables are used for collecting separately the records which have a participation on the given enterprise, and the records which are participated by such enterprise. While for the three graph DBMSs the query realization remained simple even when both directions have to be traversed, here the query became a little bit more complex.

```
CREATE TEMPORARY TABLE IF NOT EXISTS direct_children AS
    SELECT *
    FROM companies_and_memberships.memberships AS memb
        INNER JOIN
        companies_and_memberships.companies AS comp
        ON memb.partecipata = comp.codice_fiscale
    WHERE socio = '00102681419'
    LIMIT 200000;

CREATE TEMPORARY TABLE IF NOT EXISTS direct_fathers AS
    SELECT *
    FROM companies_and_memberships.memberships AS memb
        INNER JOIN
        companies_and_memberships.companies AS comp
        ON memb.socio = comp.codice_fiscale
    WHERE partecipata = '00102681419'
    LIMIT 200000;

SELECT * FROM direct_children
    UNION
SELECT * FROM direct_fathers;
```

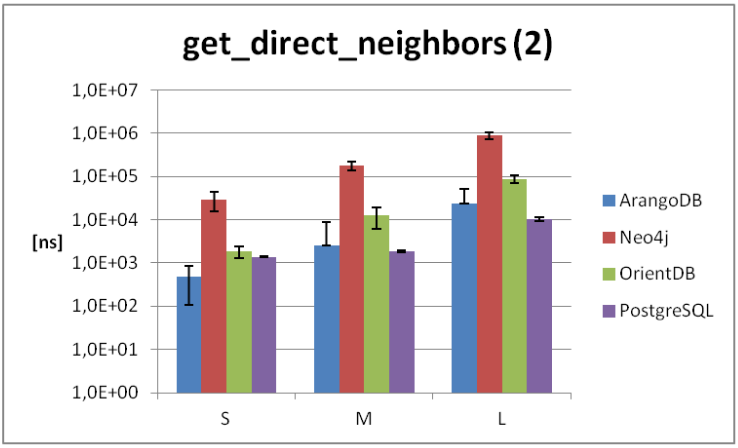
In this query we separately compute the *child* and *father* records, and we perform an union between the two temporary tables used to collect such records.

We used the two temporary tables for readability reasons, however this may implies some additional operations which could slow down the execution speed; so they could be removed and their bodies could be directly used by the UNION operation.

The number of result nodes for the enterprise with low number of direct father (S) is 5; for the intermediate case (M) the number is 120; for the case with a big number of nodes involved (L)



(a) First query



(b) Following queries

Figure 5.9: Query get_direct_neighbors charts

is 1815. The times required to resolve such query for the three situations are shown in Figure 5.9. Even this time, even though we retrieve all fields' values for each record, PostgreSQL responds in less time than the other DBMSs.

5.5.5 *get_distinct_descendants_and_level*

The query searches for the distinct descendant nodes of the node with the given *codice_fiscale* value; it thus corresponds asking for "all direct and indirect participations of the given enterprise". The query then collects all the nodes reachable by traversing outgoing edges, till the deepest level; however, we want to avoid duplicate records on the result set. In addition, we also want to obtain the minimum distance from the anchor node to the each retrieved descendant

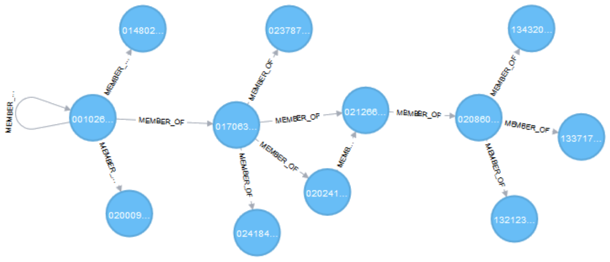


Figure 5.10: Descendants of the node with *codice_fiscale* = '00102681419'

The descendants of the node '00102681419' is shown in Figure 5.10, where the anchor node is the one to the left.

Let's proceed with some steps in order to understand how to face such query. Suppose we want to count the descendants of the node '00102681419'. We may think to obtain the number 12 (or 11 if we want to exclude the initial node); however, if we count all the results obtained, we get the number 33. In Figure 5.10 indeed we can see that there are two source of multiple paths from the anchor node to some of the descendant nodes: one is due to the loop on the initial node; the other is due to the fact that from the node whose *codice_fiscale* is '01706331419', and the node whose *codice_fiscale* is '02126680517', there are two paths. The first situation implies that all nodes can be reached not only by one path, but by two different paths, which are different only for the presence of the initial loop edge at the beginning. The second situation implies that the nodes following '02126680517' are reachable by two paths; but because of the initial loop, such nodes are in the end reachable by 4 different paths. We should then specify a DISTINCT clause on the returned result set. As already said, this can be done by simply writing DISTINCT after the RETURN keyword for both AQL and Cypher. In addition, the same could be done by writing DISTINCT after the SELECT keyword in OrientDB's SQL. For PostgreSQL, instead, it is a bit more complex, because in order to traverse a variable number of depths we made use of a CTE. We do not have a classical SELECT statement, and then we can only impose the DISTINCT clause on a SELECT statement that follows (i.e. embeds) the WITH RECURSIVE statement. We will see the query structure soon when we will show the implemented query.

Anyway, a first problem on such approach is that with AQL, Cypher and OrientDB's SQL, it is not possible to specify on what element impose the DISTINCT if more than one element is returned. So if we would like to return additional information beside the descendant nodes, the DISTINCT would take both the terms as elements over which impose the uniqueness constraint. This generally holds also for SQL, even though with PostgreSQL there is the possibility to specify over which fields impose the constraint by using the DISTINCT ON (...) clause, so that it is possible to not include some of the fields returned on the constraint. A workaround for such problem would be to add a GROUP BY row to the query, so that the desired element is returned only once. However, when such statement is used, an aggregate of the remaining fields values would be created; so if (like in this case) one wants to select only the minimum value for such aggregates, additional filtering operations should be added to the query.

In any case, to leverage on the DISTINCT clause for uniqueness imposition is not the best way to realize queries when we work with complex graphs. Indeed, the DISTINCT filtering works only at the last step of the query execution, i.e. it scans all the obtained results and returns multiple occurrences of the same element only once; the GROUP BY approach, on the other hand, would behave the same way. Going back to the query that searches for all the descendants of a given node, if a lot of nodes are traversed while performing the query, and if a lot of interconnections are present among such nodes, then a lot of multiple paths for the result nodes would be present, and the query would work multiple times on the same nodes and, only at the final step, it would limit the nodes to be returned by filtering them. This is not highlighted on nodes with few descendants; however, when we have a more complex situation, the effects are well visible. The best approach would be to restrict the traversal of the graph so that nodes uniqueness is ensured while traversing and with "global scope". In this way, as soon as a node is detected as duplicate, it would not been added to the result set and the path that was being constructed by the exploration would not be further expanded, rather it would be discarded. In addition, such approach would bring to a temporary data set - populated while executing the query - which is reduced in space.

Such problem arises by the fact that the default traversal behavior of the graph databases given is to ensure edges (or nodes) uniqueness on path scope; while in this query we would like to impose nodes uniqueness at global scope.

The other component of the query is the search for the length of the minimum shortest path from the initial node to each (distinct) descendant returned. The query thus simulates the Dijkstra's algorithm for the computation of the shortest paths from a given source node to all the reachable nodes.

The invocation of the shortest path function (provided by the three graph query languages) from the given node to all its descendants would be impractical with a large number of descendant nodes. Rather, we could achieve our goal in a single query by properly working on the exploration phase. In order to do this, we need to change another thing from the default traversal approach: the exploration order. The simplest way to obtain the computation of the shortest path from a start node to a destination node, indeed, is by exploring the graph in breadth-first order and by returning the length of the first path found during the exploration. If the data domain would have been a tree, no multiple paths for the same node would be present. However, in graphs, there both may be cycles (paths that pass through the same node/edge more than once) and multiple ways for reaching the same node by coming from the same initial node. While the uniqueness constraint on node/edges visits with path scope resolves the cycles avoidance problem, the problem of avoiding multiple visits on a global scope would not be faced efficiently if there are no ways to declare it directly during the graph exploration phase.

The two key ingredients for this query are then: the modification of the exploration order (set to Breadth-First); and the uniqueness constraint (set for nodes with global scope).

In AQL the query is implemented as shown below:

```
LET startVertex = 'Enterprise/00102681419'
LET minLevel = 1
LET maxLevel = 999
FOR v, e, p
  IN minLevel..maxLevel
  OUTBOUND startVertex
  GRAPH 'enterprises_and_members_Graph'
  OPTIONS {bfs: true, uniqueVertices : 'global'}
  LET level = ( LENGTH(p.edges[*]) )
  LIMIT 200000
  RETURN { 'vertex': v, 'level': level }
```

The AQL Traversal syntax expects at most three parameters after the FOR keyword, which are variable names for - respectively - vertex, edge, and path elements. Such variables can be used within the query, as shown for example by the row where the computation of the path length is performed. In addition, the *bfs* OPTION set to true is the one which changes the exploration approach. A very high max depth level has been specified so that no possible paths are cut away while exploring the graph. Anyway, by the characteristics of the dataset, such value could be set way more smaller (like 50 or 100) and may be used like a safety limit in those cases where the query seems to have started working too much,maybe because of some unexpected data situations.

In Cypher the query would be implemented as shown below:

```
MATCH path = (startNode:Enterprise {codice_fiscale:
    '00102681419'})-[m:MEMBER_OF*1..999]->(descendant:Enterprise)
WITH MIN( LENGTH(path) ) AS level, descendant
RETURN level, descendant
ORDER BY level
LIMIT 200000;
```

The query can be distinct in three components: in the MATCH phase we define the graph patterns to be retrieved, which are variable length paths made of outgoing edges; then it follows a WITH statement, which is used for performing some aggregate functions and where the variables used in successive phases should be reported; and then it follows a RETURN statement. The WITH statement was added only to allow the ordering based on the depth value found. The part of the query where the majority of the amount of work is executed corresponds to the MATCH phase. In Cypher it is not possible to define how the query should be resolved; in particular it is not possible to change both the uniqueness constraint and the exploration order. This implies that the databases works in depth-first order and with edge uniqueness at path scope; and it computes all the possible paths before proceeding to the second phase where the minimum length path is selected for each descendant node. This implies that when a particularly heavy situation is found around a given node - i.e. there are a lot of nodes and multiple paths to some of its internal (and thus also leaf nodes) - the query may not answer in useful time. The only way to force the different traversal constraints we need is by using the Java API provided, which is called Traversal API. The query (which is part of a purposely implemented Java class) is shown below:

```
int resultsCounter = 0;
Map<String, Object> properties;
try ( Transaction tx = graphDb.beginTx() ) // execute within a transaction
{
    // Find the source node
    sourceNode = graphDb.findNode( Labels.Enterprise, "codice_fiscale", c_f );
    // Describe how to retrieve the desired nodes
    resultNodes = graphDb.traversalDescription()
            .breadthFirst()
            .relationships( RelationshipTypes.MEMBER_OF, Direction.OUTGOING )
            .evaluator( Evaluators.excludeStartPosition() )
            .uniqueness(Uniqueness.NODE_GLOBAL);
    traverser = resultNodes.traverse( sourceNode );
    // Extract the result
    for ( Path path : traverser )
    {
        StringBuilder sb = new StringBuilder();
        sb.append( "    level: " + path.length() + ", " );
        properties = path.endNode().getAllProperties();
        int iteratorPosition = 0;
        for ( Map.Entry<String,Object> entry : properties.entrySet() ) {
            // put all properties on the result string
            sb.append( entry.getKey() + ": " + entry.getValue() );
            if ( iteratorPosition == properties.entrySet().size() - 1 ) {
                sb.append( ";\n" );
            }
            else {
                sb.append( ", " );
            }
            ++iteratorPosition;
        }
        ++resultsCounter;
        // print single node properties
        System.out.print( sb.toString() );
    }
}
```

The query must be executed within a transaction and is composed of: an initial section where the starting node is searched; a section where it is described how the traversal must be performed; and a following section where the query is executed with a lazy-evaluation approach, i.e. the actual traversal is performed lazily each time the *next()* method of the iterator of the Traverser is invoked. This also means that the full traversal is not fully performed if the *next()* method is not invoked for all the possible times [79]. Then, for each possible path, the values of the descendants' properties (and the length of the path) are retrieved, collected, and at the end printed. The core of the query is the query description section: there the breadth-first approach can be specified, as long as the uniqueness constraint.

Note that all implemented Neo4j's queries with the Traversal API are preceded by a call to the query that performs the warm-up of the cache.

In OrientDB the query may be based on a query structure made like:

```
SELECT expand(descendant)
FROM (
  MATCH {class: Enterprise, as: start_node, where: (codice_fiscale =
      '00102681419')}.out('member_of'){as: descendant, while: ($depth + 1
      <= 999)}
  RETURN descendant
  LIMIT 200000
)
```

Such query exploits the suggested command for the implementation of graph traversals, which is MATCH. However, it works in depth-first order, so the global uniqueness constraint may return a wrong depth value. Another formulation should then be used, which is the one that follows:

```
SELECT $depth AS level, *
FROM (
  TRAVERSE out('member_of')
  FROM (
    SELECT *
    FROM Enterprise
    WHERE codice_fiscale = '00102681419'
  )
  MAXDEPTH 999
  LIMIT 200000
  STRATEGY BREADTH_FIRST
)
ORDER BY level
```

The query is made of a SELECT statement with a nested TRAVERSE statement. The TRAVERSE statement is the one which describes how to retrieve data, while the outer SELECT statement only selects the data to be returned. Within the TRAVERSE statement we define the traversing direction by using the *out()* method; the starting node (which is retrieved by its *codice_fiscale* value); a maximum exploration depth; a maximum number of nodes to limit eventual query's perpetual explorations; and the exploration order. At the end we exploit the $depth system variable for the length of the path, and we return the results ordered by it. Within the query there misses the definition of the uniqueness constraint; this is due to the fact that OrientDB natively imposes vertices uniqueness at global scope when the TRAVERSE statement is used.

In PostgreSQL the query is written as shown below. The query uses a CTE for recursively adding rows that represent the direct and indirect participated enterprise records found while traversing outgoing relationships. The WITH RECURSIVE statement is composed of two sections: the first it the statical section, which is executed just one time, as the first operation; the second sections holds the sub-query that has to be executed recursively until some stop conditions are met, or all reachable records have been explored.

The fields of the CTE are:

- *socio*: the leftmost field of the table containing the relationships between companies; it corresponds to the source of the relationship.
- *partecipata*: the rightmost field of the same table; it corresponds to the destination of a relationship.
- *depth*: the number of relationships traversed for the current path.
- *path*: an array of strings containing the *codice_fiscale* values for the records visited during each path traversed.
- *alreadyVisited*: a flag field that signals the fact that the analyzed enterprise record have already been visited within the same path; it highlights then the presence of cycles within it.

In the first section, the CTE is initialized with the records of the table “memberships” where the *socio* field contains the *codice_fiscale* given; in this way, the nodes reachable by means of a single outgoing edge are collected, which are the direct children of the given enterprise “node”. The depth value is initialized to 1 for such new rows, the path is updated with the starting enterprise, and the cycle flag is set to false.

In the second section of the WITH RECURSIVE statement there is specified the sub-query to be executed recursively on data collected within the CTE. In particular, a (hidden) JOIN between the memberships table and the current CTE is performed, where the fields for the JOIN are the *partecipata* field of the CTE and the *socio* field of the membership table. In this way, new records are searched from the memberships table, which are those representing participation edges that start from the nodes that have been collected during the previous phase of the recursive call. An additional edge is then traversed starting from the nodes previously found, and the reached *codice_fiscale* values are added to the CTE itself.

```
WITH RECURSIVE traverse_outgoing_edges(socio, partecipata, depth, path,
    alreadyVisited) AS (
     SELECT socio, partecipata, 1, ARRAY[socio]::character(16)[], false
     FROM companies_and_memberships.memberships
     WHERE socio = '00102681419'
  UNION ALL
     SELECT _current.socio, _current.partecipata, _previous.depth + 1,
       (path || _current.socio)::character(16)[],
       _current.socio = ANY(path)
     FROM companies_and_memberships.memberships AS _current,
        traverse_outgoing_edges AS _previous
     WHERE _current.socio = _previous.partecipata
          AND _previous.depth + 1 <= 30 -- BE CAREFUL: do not use a too
             high depth value!
          AND _current.partecipata != _current.socio -- avoid immediate
             cycles
          AND NOT _previous.alreadyVisited
),
cleaned_descendants AS (
     SELECT DISTINCT ON (partecipata) partecipata, depth -- select the
        first occurrence of the pair (CF,depth)
     FROM traverse_outgoing_edges
     WHERE NOT alreadyVisited
     ORDER BY partecipata, depth -- select the pair with minimum depth
        value
     LIMIT 200000
)

SELECT *
FROM cleaned_descendants AS c_d
     JOIN
     companies_and_memberships.companies AS comp
     ON c_d.partecipata = comp.codice_fiscale
ORDER BY depth -- order the list of descendants according to the depth
    value
```

The new records are composed of the *socio* and *partecipata* values retrieved from the memberships relation; the incremented value for the depth, the concatenation of a new *codice_fiscale* value to the path string, and the detection of a cycle within the path. Such new records will be then used for the next iteration of the sub-query. However, some checks are performed at each iteration with respect to the maximum exploration depth allowed; to the avoidance of immediate cycles within the extracted nodes; and with respect to the absence of cycles during the previous iteration, so that useless work is not computed at all. (Note that the example provided by the official documentation of PostgreSQL instead computes the presence of a cycle only at the current stage, when maybe useless computations could already had been performed).

The WITH RECURSIVE block is followed by a WITH statement, which collects some of the data extracted on a separate CTE, so that it can be used by a successive SELECT query. Within this new CTE are collected distinct records belonging to the previous CTE. The combination of ORDER BY and DISTINCT allows to select the first occurrence of the pairs (*partecipata*, depth) (i.e. the one with minimum depth value) for each *partecipata* value. The next SELECT query then globally orders the pairs by means of the depth value and performs a JOIN with the anagraphical table so that all field values are given for each descendant enterprise record reached.

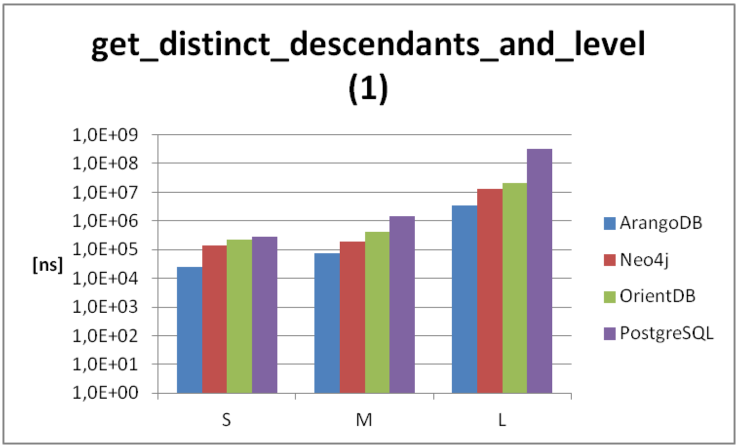
The query thus simulates a traversal which starts from a given node and proceeds in breadth-first order; such exploration order is given not by the filtering and ordering a-posteriori of the results obtained, but properly by the way the WITH RECURSIVE query proceeds during the execution.

The number of result nodes for the enterprise with low number of descendants (S) is 12; for the intermediate case (M) the number is 76; for the case with a big number of nodes involved (L) is 14037. The time values shown in Figure 5.11 are related to the times obtained by using the Java query for Neo4j, rather than the Cypher query. In fact, the times taken by the Cypher query for the same three nodes (S), (M) and (L) are the ones displayed in Figure 5.12. Actually, the end time for the L case with the path uniqueness constraint (Cypher) is undefined because after several minutes we stopped the execution. The comparison between the path uniqueness and the global uniqueness constraints is showed only for Neo4j but holds also for the others.

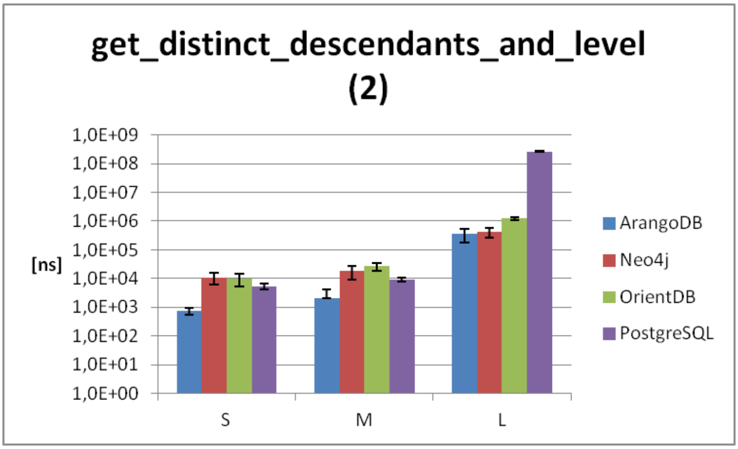
As it can be seen in Figure 5.11, while PostgreSQL answers in times quite similar to the graph databases for the S and M cases, for the L case the difference becomes more evident, in fact there is an order of magnitude of difference between them.

In general, and in particular way for PostgreSQL, the more the levels, the nodes per level, and the interconnections among such nodes; the more the memory and the time required for the execution of the query.

Anyway, the main reason why the relational query takes so much time is due to the fact that the global uniqueness on enterprise records is imposed only at a posteriori, and not while traversing the simulated graph. There is no way to realize a global uniqueness constraint by using the WITH RECURSIVE statement. One approach could be the one that makes use of a temporary table for collecting the *socio* values visited; each time a new record is retrieved by the recursive phase, it would then be assured that it is added to the CTE only if it is not already present within such auxiliary table. However, it is not possible to create or update temporary tables within a WITH RECURSIVE statement, so the proposed approach cannot be implemented. One may think that another solution may be the one that, instead of using a temporary table, uses a sting which collects the *socio* values, as it is done by the path string. However, every

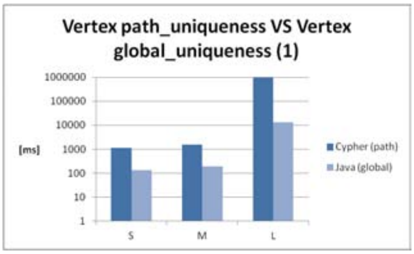


(a) First query

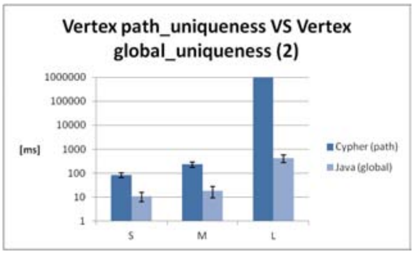


(b) Following queries

Figure 5.11: Query get_distinct_descendants_and_level charts



(a) First query



(b) Other queries

Figure 5.12: Quick study of path uniqueness vs. global uniqueness.

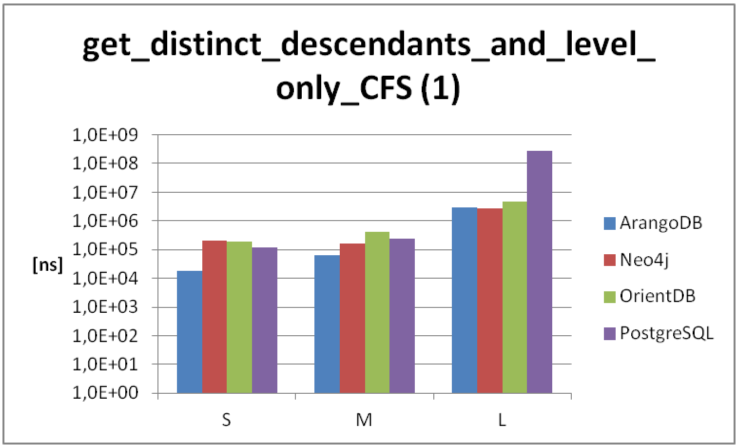
path string is a value bound to each row of the CTE, so it is local to the records and not global for the entire table. So again even this approach is not valid. One possibility could be to analyze the CTE itself for checking that the reached value has not been already inserted in previous rows. However such approach has not been tried.

There are two alternative ways to overcome such limits of the WITH RECURSIVE statement, so that both breadth-first and global uniqueness are ensured, which are the implementation of a quite complex stored procedure, or the implementation of an algorithm developed with an high-level programming language which continuously queries the database. Such algorithm would asks, step by step, which are the edges to traverse at each iteration; and would handle the aspects about result storage, cycles detection, global constraints, exploration order, etc. However, such approaches have not been tested.

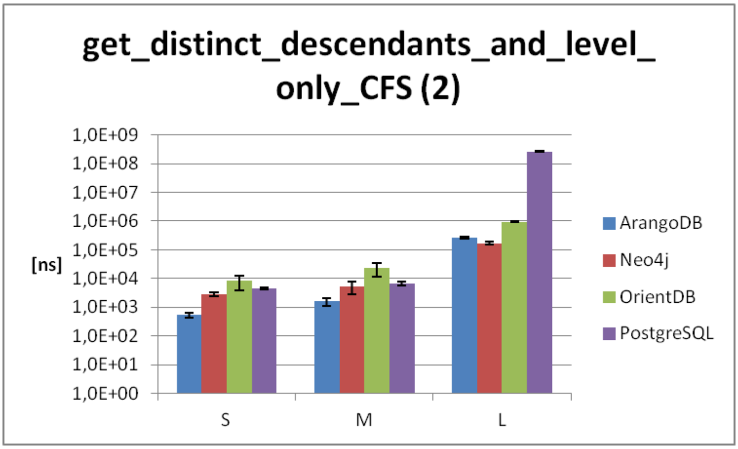
The other component that enhance the time required by PostgreSQL to answer to the query is that all temporary data generated by the execution of the query are placed on disk [133]. In fact, while the amount of RAM remains reduced during the execution (about 180MB), the disk space is heavily used as support for the elaboration. During this query, the peak of disk space used was of about 7GB. The other DBMSs instead managed to keep temporary data on RAM, and - by doing a quick check with the *top* Linux command - we typically have that ArangoDB uses a lot of RAM for keeping the indexes (about 5.4GB), while the other DBMSs require less RAM, and the one which typically needs less of it is OrientDB; in fact Neo4j used 2.4GB of RAM, while OrientDB used 220MB (without cache warm-up).

Note the fact that the global uniqueness constraint not only meets the requirement of returning just once each descendant, but also avoids to explore multiple times some of the graph areas that could be reached several times with a local uniqueness constraint by different paths. We will explain this with more details in Section 6.

We also studied the similar query that retrieves only the *codice_fiscale* values for each node. The query structure is identical to the previous one, the only change is on the results returned at the final step of its execution (just like for the query *get_direct_fathers_only_cfs*). The current query is performed on the same *codice_fiscale* values as previous and the times required are



(a) First query



(b) Following queries

Figure 5.13: Query get_distinct_descendants_and_level_only_cfs charts

shown in Figure 5.13.

As it can be seen, the times required to resolve the query are generally smaller for all the analyzed DBMSs with respect to the previous case.

5.5.6 *count_distinct_ancestors_per_level*

Suppose we want to get some statistic measures about the surrounding area of a node, and that we are interested in obtaining the number of nodes which are one hop distant from a specified node - i.e. those reachable by one (maybe directed) edge - then those two hops distant, etc. The query may seem simple: in fact it would consist on the count of the length values for all the paths obtained; in this case the paths are made of only ingoing edges. The example with the node '13371752902' is shown in Figure 5.14. The starting node is the one to the right, we then want a result which states that at level 1 we have one node, at level 2 again one node, at level 3 we have two nodes, and so on.

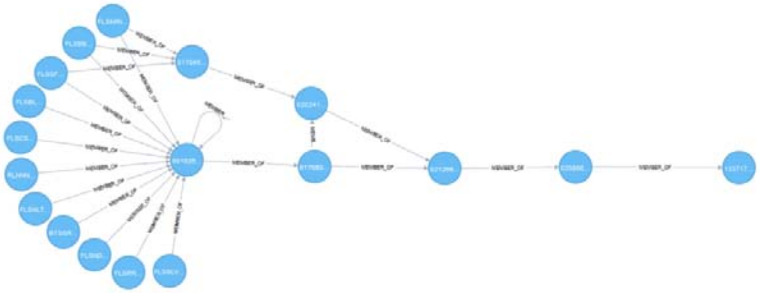


Figure 5.14: Ancestors of the node with *codice_fiscale* = '13371752902'

Such queries would be, for example, written like:

```
LET startVertex = 'Enterprise/00102681419'
LET minLevel = 1
LET maxLevel = 999
FOR v, e, p
  IN minLevel..maxLevel
  INBOUND startVertex
  GRAPH 'enterprises_and_members_Graph'
  COLLECT level = LENGTH(p.edges[*]) INTO vertices_per_level
  LIMIT 200000
  RETURN {'level' : level, 'count' : LENGTH(vertices_per_level)}
```

with AQL, and with Cypher like:

```
MATCH p = (e:Enterprise {codice_fiscale:
    '00102681419'})<-[m:MEMBER_OF*1..999]-(ancestor:Enterprise)
WITH MIN(LENGTH(p)) AS level, ancestor.codice_fiscale AS ancestors
RETURN level, SIZE(COLLECT(ancestors)) AS ancestors_number
ORDER BY level
LIMIT 200000;
```

However, these queries would return the following counting values: one node at level 1; one node at level 2; two nodes at level 3; but three nodes at level 4, fifteen nodes at level 5, and 11 nodes at level 6; while one may expect, by rapidly looking at the figure, to get at most 5 levels and with the sum of the counting numbers to 17. In addition, the given numbers for each level are referred to the case where a uniqueness constraint is defined on vertexes and with path scope; if the constraint would be defined on edges with path scope, the number would be even incremented by the presence of the loop on the '00102681419' node.

The queries respond correctly for the way they are written, but not as one may have thought the initial query. They count, in fact, the different lengths of all the possible paths that can be found. However, if the original question was not simply the one given before, but "get the distribution, on the various depth levels, of the distinct ancestors of the given node", here one wants to know how many of the distinct ancestors are placed only one hop distant, how many are instead placed two levels distant without considering those who were already counted at the first level, and so on.

Again, if this second one is the query of interest (as will be for us) a depth-first approach with "local" uniqueness would not answer to the query; in fact it gives the results previously reported. The query has to be resolved with global uniqueness constraint for nodes - so that nodes are considered only once - and with a breadth-first exploration - so that only the smallest distance is considered. So, for the example node provided, we would like to obtain a result made like this:

level	ancestors_number
1	1
2	1
3	2
4	2
5	11

Table 5.2: Result example for query count_distinct_ancestors_per_level

In AQL, the query is thus written like:

```
LET startVertex = 'Enterprise/00102681419'
LET minLevel = 1
LET maxLevel = 999
FOR v, e, p
  IN minLevel..maxLevel
  INBOUND
  GRAPH 'enterprises_and_members_Graph'
  OPTIONS {bfs: true, uniqueVertices : 'global'}
  COLLECT level = LENGTH(p.edges[*]) INTO vertices_per_level
  LIMIT 200000
  RETURN { 'level' : level, 'count' : LENGTH(vertices_per_level) }
```

In Neo4j the Traversal API should be used, the query structure remains the same of the query about the search of the descendant nodes.

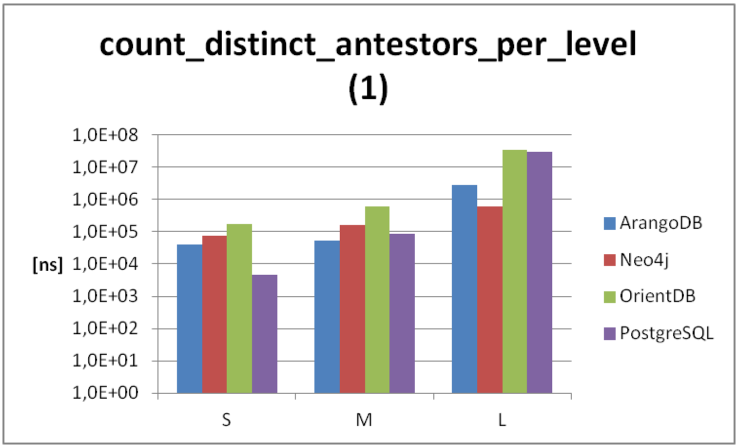
```
HashMap<Integer,Integer> nodPerLev = new HashMap<Integer, Integer>();
StringBuilder sb = new StringBuilder();
try ( Transaction tx = graphDb.beginTx() ) // execute within a transaction
{
    // Find the source node
    sourceNode = graphDb.findNode( Labels.Enterprise, "codice_fiscale", c_f );
    // Describe how to retrieve the desired nodes
    resultNodes = graphDb.traversalDescription()
            .breadthFirst()
            .relationships( RelationshipTypes.MEMBER_OF, Direction.INCOMING )
            .evaluator( Evaluators.excludeStartPosition() )
            .uniqueness(Uniqueness.NODE_GLOBAL);
    traverser = resultNodes.traverse( sourceNode );
    // Extract the result
    for ( Path path : traverser ) {
        ++resultsCounter;
        // for each result node, update the depth counter
        Integer previousCounter = getPreviousCounter( nodPerLev, path.length()
            );
        nodPerLev.put( (Integer)path.length(), previousCounter + 1 );
    }
    // print counting result
    for ( Map.Entry<Integer, Integer> entry : nodPerLev.entrySet() ) {
        sb.append( "    level: " + entry.getKey() + ", count: " +
            entry.getValue() + "\n" );
    }
}
[...]
private static Integer getPreviousCounter(HashMap<Integer, Integer> map,
    Integer key) {
    map.get(key) == null ? return 0 : return (Integer)(map.get(key));
}
```

A HashMap object is used to store the number of descendants found at the different depth values: the key is the depth; the value is the count for such depth level.

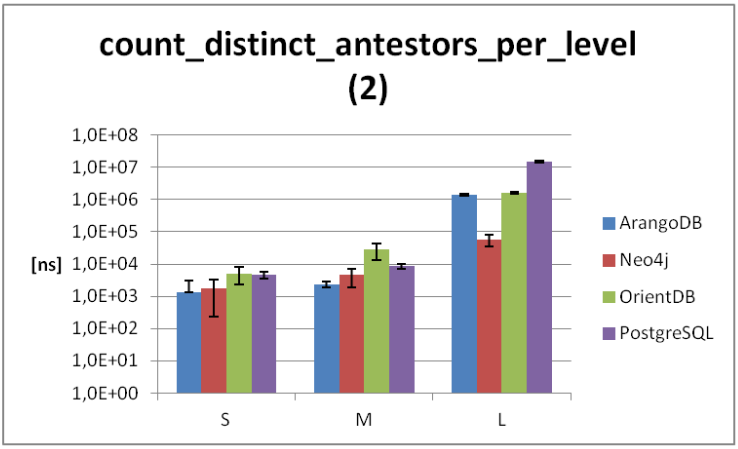
In OrientDB the query is implemented as shown below. We again use the TRAVERSE statement so that the breadth-first strategy can be forced; in addition the counting of the distinct ancestors is performed on an outer SELECT statement.

```
SELECT level, list(codice_fiscale).size() AS ancestors_count
FROM (
  SELECT codice_fiscale, $depth AS level
  FROM (
    TRAVERSE in('member_of')
    FROM (SELECT FROM Enterprise WHERE codice_fiscale = '00102681419')
    MAXDEPTH 999
    LIMIT 200000
    STRATEGY BREADTH_FIRST
  )
  WHERE $depth > 0 )
  GROUP BY level
  ORDER BY level
```

The query for PostgreSQL is written as follow:



(a) First query



(b) Following queries

Figure 5.15: Query count_distinct_ancestors_per_level charts

```
WITH RECURSIVE traverse_ingoing_edges(socio, partecipata, depth, path,
    alreadyVisited) AS (
    SELECT socio, partecipata, 1, ARRAY[partecipata]::character(16)[], false
    FROM companies_and_memberships.memberships
    WHERE partecipata = '00102681419'
  UNION ALL
    SELECT _current.socio, _current.partecipata, _previous.depth + 1,
      (path || _current.partecipata)::character(16)[],
      _current.partecipata = ANY(path)
    FROM companies_and_memberships.memberships AS _current,
        traverse_ingoing_edges AS _previous
    WHERE _current.partecipata = _previous.socio
        AND _previous.depth + 1 <= 30
        AND _current.partecipata != _current.socio
        AND NOT _previous.alreadyVisited
),
cleaned_ancestors AS (
    SELECT DISTINCT ON (socio) socio, depth
    FROM traverse_ingoing_edges
    WHERE NOT alreadyVisited
    ORDER BY socio, depth -- take on top the pairs (socio,depth) with smallest
        depth values
    LIMIT 2000000
)

SELECT depth, count(socio) AS members_count
FROM cleaned_ancestors
GROUP BY depth
ORDER BY depth;
```

The core structure of the query remains the WITH RECURSIVE statement; however some additional operations are performed on the following query statements so that the request is met. Again, the distinct filter is only applied after the full exploration has been performed; so potential useless computations will be elaborated.

Such queries allow then to count the nodes only once, and to place them at the nearest level possible to the initial node.

The queries are performed on a node where the number of levels (with the sense here explained) is 1 and with about 5 ancestors (S); on another node with 3 levels and with about 100 ancestors (M), and on a node with 29 levels and about 20000 ancestors (L). The times required for their executions are shown in Figure 5.15. We see that OrientDB takes more time than the others on the first execution,and the same holds for the next, meaning that probably some optimization may be done on the query implemented or on the counting function. We also see that Neo4j goes way better than the others on the heavy situation (L).

5.5.7 *get_common_descendants*

The query searches for the enterprises that are directly and indirectly participated by two given enterprises. In a single pass, such query answers then to the need of looking at whether there are some enterprises participated by both the two enterprises analyzed; and in such case, to obtain a list of them. The example with the two nodes 'FLSBLN55L65G482J' and 'BTSSRM48K54D321A' is shown in Figure 5.16.

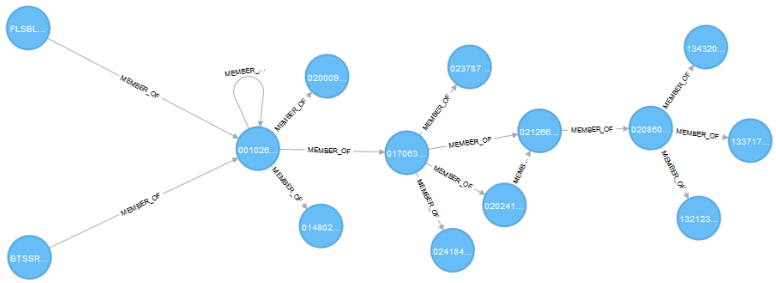
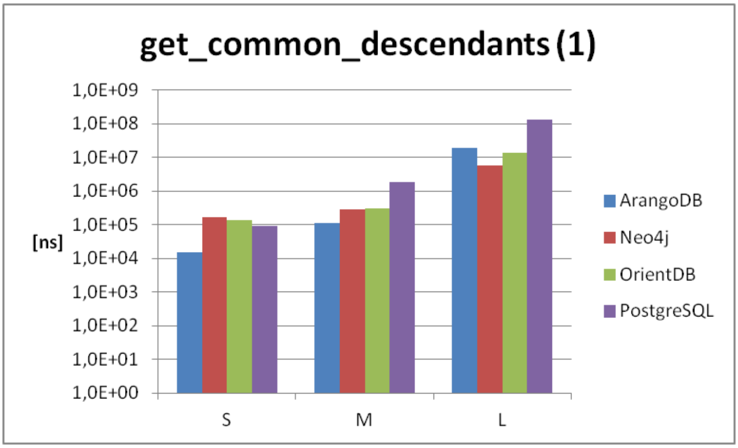


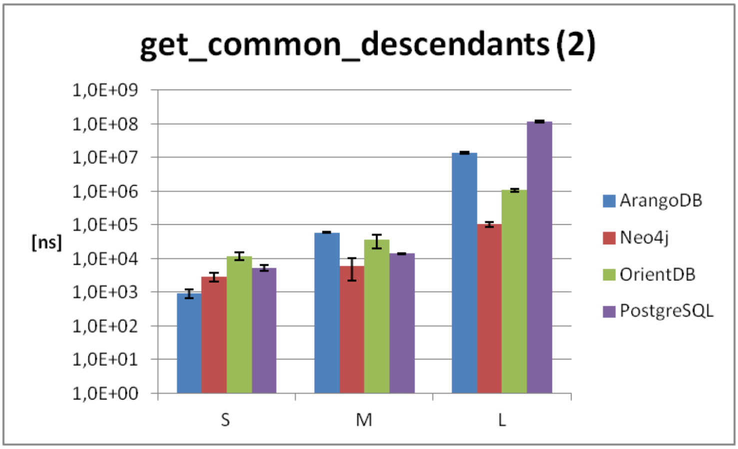
Figure 5.16: Common descendants of the nodes with *codice_fiscale* 'FLSBLN55L65G482J' and 'BTSSRM48K54D321A'.

The first thought approach was based on the intersection of the two sets of descendants. However, a second approach has been thought, whose basic idea is the following: if it was possible to efficiently obtain which is the pivotal node (or nodes) - i.e. the nearest node in common of both the descendants of the two nodes - then we may apply the descendant search for such node and obtain the list of descendants. In the example provided with Figure 5.16, the pivotal node would be '00102681419'.

However, the problem with this approach is that in order to obtain the pivotal node(s), we have to find and intersect the two lists of descendant nodes. So we perform the computations for the entire query only for the computation of the single node, which is a waste if we later use such node for the search of its descendants. The main reason of its infeasibility is due to the fact that such pivotal node may be more than one level distant from the source nodes; and also, the distances from the two nodes would in general be different. The only case where the approach here reported would be efficient is when the pivotal node is one hop distant from both the two nodes, that is it is child of both of them, like in Figure 5.16. In this way, the intersection with a (hopefully) restricted amount of nodes would lead to an efficient individuation of such node. However, the general case do not allow to proceed this way, neither allows to iteratively proceed level by level for both nodes so that the intersection of the two result nodes is not null, because, as already stated, it is not ensured that the node has the same distance from the two initial nodes. So such approach was discarded, and the simple intersection between the two list has been implemented.



(a) First query



(b) Following queries

Figure 5.17: Query get_common_descendants charts

With AQL the query is written like:

```
LET firstVertex = ‘Enterprise/00210880225’
LET secondVertex = ‘Enterprise/00487520223’
FOR common_descendants
IN INTERSECTION (
   (
     FOR vertices1 IN 1..999
     OUTBOUND firstVertex
     GRAPH ‘enterprises_and_members_Graph’
      LIMIT 200000
     RETURN vertices1
   ),
   (
     FOR vertices2 IN 1..999
     OUTBOUND secondVertex
     GRAPH ‘enterprises_and_members_Graph’
      LIMIT 200000
     RETURN vertices2
   )
)
RETURN common_descendants
```

An intersection is performed on the result sets of the two separate descendants collections; in addition all values for nodes properties are printed.

With Neo4j the query is implemented in Java as shown below:

```
Set<String> set1 = new HashSet<String>();
Set<String> set2 = new HashSet<String>();
try ( Transaction tx = graphDb.beginTx() ) {
    // Find the source node 1 and describe how to retrieve the desired nodes
    sourceNode1 = graphDb.findNode( Labels.Enterprise, "codice_fiscale", c_f_1 );
    resultNodes1 = graphDb.traversalDescription()
                          .breadthFirst()
                          .relationships( RelationshipTypes.MEMBER_OF, Direction.OUTGOING )
                          .evaluator( Evaluators.all() )
                          .uniqueness(Uniqueness.NODE_GLOBAL);
    traverser1 = resultNodes1.traverse( sourceNode1 );
    [...] // The same for the second node
    // Extract results for the first node
    for ( Path path : traverser1 ) {
        String cfValue = (String) path.endNode().getProperty( "codice_fiscale");
        ++resultsCounter1;
        set1.add(cfValue);
    }
    [...] // The same for the second node
}
// Intersect sets
set1.retainAll(set2);
// Prepare the result string
int intersectedCount = 0;
StringBuilder sb = new StringBuilder();
sb.append( "   " );
for ( String element : set1 ) {
    sb.append( element );
    ++intersectedCount;
    if(intersectedCount == set1.size()) sb.append( ";\n" );
    else sb.append( ", " );
}
```

Again, the query collects separately two sets of *codice_fiscale* values, related to the descendants of the two nodes; at the end, the intersection of the two sets is performed. However, only the value of the *codice_fiscale* property is returned by such algorithm.

In OrientDB the query is implemented as shown below:

```
SELECT EXPAND( $c )
     let $a = (
       TRAVERSE out('member_of')
       FROM (
         SELECT FROM Enterprise WHERE codice_fiscale = '01855720155'
       )
       MAXDEPTH 999
       LIMIT 200000
       STRATEGY BREADTH_FIRST
     ),
     $b = (
       TRAVERSE out('member_of')
       FROM (
         SELECT FROM Enterprise WHERE codice_fiscale = '02935970984'
       )
       MAXDEPTH 999
       LIMIT 200000
       STRATEGY BREADTH_FIRST
     ),
     $c = INTERSECT( $a, $b )
```

The query thus performs an intersection between the two sets of RIDs for the descendants collected, and then it expands those RIDs so that all properties values for each node are displayed.

In PostgreSQL the query is implemented as shown below.

```
WITH RECURSIVE traverse_outgoing_edges_1(socio, partecipata, depth, path, alreadyVisited) AS (
    SELECT socio, partecipata, 1, ARRAY[socio]::character(16)[], false
    FROM companies_and_memberships.memberships
    WHERE socio = '01706330519'
  UNION ALL
    SELECT _current.socio, _current.partecipata, _previous_1.depth + 1,
      (path || _current.socio)::character(16)[],
      _current.socio = ANY(path)
    FROM companies_and_memberships.memberships AS _current, traverse_outgoing_edges_1 AS
         _previous_1
    WHERE _current.socio = _previous_1.partecipata
        AND _previous_1.depth + 1 <= 30
        AND _current.partecipata != _current.socio
        AND NOT _previous_1.alreadyVisited
),
traverse_outgoing_edges_2(socio, partecipata, depth, path, alreadyVisited) AS (
    SELECT socio, partecipata, 1, ARRAY[socio]::character(16)[], false
    FROM companies_and_memberships.memberships
    WHERE socio = '01706500517'
  UNION ALL
    SELECT _current.socio, _current.partecipata, _previous_2.depth + 1,
      (path || _current.socio)::character(16)[],
      _current.socio = ANY(path)
    FROM companies_and_memberships.memberships AS _current, traverse_outgoing_edges_2 AS
         _previous_2
    WHERE _current.socio = _previous_2.partecipata
        AND _previous_2.depth + 1 <= 30
        AND _current.partecipata != _current.socio
        AND NOT _previous_2.alreadyVisited
),
cleaned_descendants_1 AS (
    SELECT DISTINCT ON (partecipata) partecipata, depth
    FROM traverse_outgoing_edges_1
    WHERE NOT alreadyVisited
    ORDER BY partecipata, depth
    LIMIT 200000
),
final_descendants_list_1 AS (
SELECT depth, partecipata AS subsidiaries
FROM cleaned_descendants_1
ORDER BY depth
),
cleaned_descendants_2 AS (
    SELECT DISTINCT ON (partecipata) partecipata, depth
    FROM traverse_outgoing_edges_2
    WHERE NOT alreadyVisited
    ORDER BY partecipata, depth
    LIMIT 200000
),
final_descendants_list_2 AS (
    SELECT depth, partecipata AS subsidiaries
    FROM cleaned_descendants_2
    ORDER BY depth
),
intersected_descendants AS (
    SELECT subsidiaries FROM final_descendants_list_1
    INTERSECT
    SELECT subsidiaries FROM final_descendants_list_2
)
SELECT *
FROM intersected_descendants AS i_d
    INNER JOIN
    companies_and_memberships.companies AS comp
    ON i_d.subsidiaries = comp.codice_fiscale;
```

We followed the approach used in previous query for the exploration of the graph; we save the two descendants lists in two CTEs and, at the end, we perform the intersection of them.

The number of common nodes for the "small" case (S) is 6; for the intermediate case (M) the number is 84; for the "large" case (L) the number is 8728. The times required to resolve such query for the three situations are shown in Figure 5.17. We see that Neo4j and OrientDB are aligned on the first execution, and a pattern with Neo4j which performs better than OrientDB can be seen at operating speed. ArangoDB on the other hand seems to show some difficulties on the heavy node. PostgreSQL shows good performance at operating speed for the cases S and M; however for the L case it responds an order of magnitude later than the worst graph database, and three orders of magnitude later than the best.

We did a quick study on the memory consumption also for this query. Even in this case, PostgreSQL uses few RAM (about 70MB) and places its temporary data on disk, consuming about 3GB of space, which is clearly released at the end of the execution. This time we have that both Neo4j and OrientDB use about 2.5GB of RAM, while ArangoDB remains at its 5.4GB.

5.5.8 *get_common_ancestors*

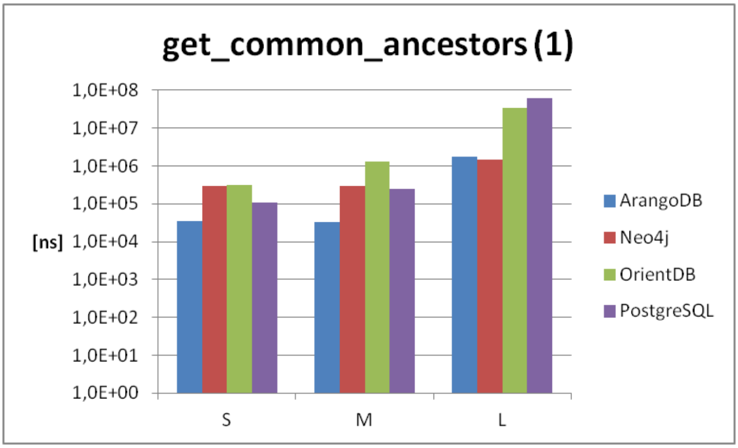
Here the queries are identical to the previous ones, it only changes the direction of the exploration. Note, however, that while for graph databases such thing is achieved by simply changing the direction keyword within the query, for SQL it have to be changed the fields for the JOINs and the field over which search the given *codice fiscale* value at the initial step of the recursive statement (i.e. it must be searched on the *partecipata* field instead of the *socio* field). In addition, also the policies for path and alreadyVisited fields should be "inverted".

The number of common nodes for the "small" case (S) is 26; for the intermediate case (M) the number is 140; for the "large" case (L) the number is 19548. The times required to resolve such query for the three situations are shown in Figure 5.18. Again, Neo4j is the one which performs better on the heavy case; on the other hand ArangoDB performs well on the remaining cases.

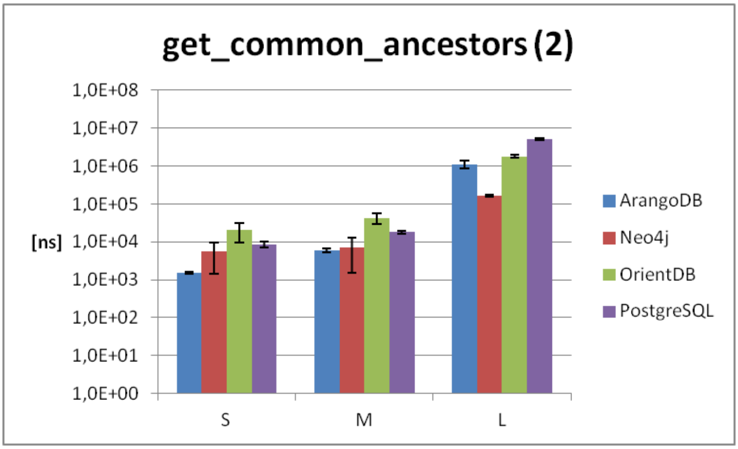
5.5.9 *get_shortest_directed_path*

The query searches for the shortest path, made of only outgoing edges, from a given node to another given node. The example for the pair ('00102681419', '02086091417') is shown in Figure 5.19.

With AQL the query is implemented as shown below:



(a) First query



(b) Following queries

Figure 5.18: Query get_common_ancestors charts

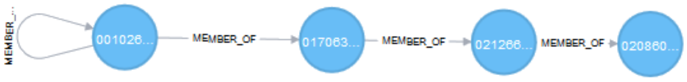


Figure 5.19: Shortest (directed) path from the node ‘00102681419’ to the node ‘02086091417’.

```
LET startVertex = 'Enterprise/80147930150'
LET targetVertex = 'Enterprise/08800160964'
LET path = (
  FOR vertex, edge
  IN OUTBOUND
  SHORTEST_PATH startVertex TO targetVertex
  GRAPH 'enterprises_and_members_Graph'
  RETURN vertex
)
RETURN { 'path': path, 'length': LENGTH(path) - 1 }
```

The syntax is not the same of the AQL Traversal; in fact, it is a new syntax designed for this specific purpose. In this query structure we obviously specify the edges directions, the two extreme nodes of the path, and the named graph. At the end, we also return the length of the path, which is decremented by one because the *LENGTH()* function, applied to a path, substantially counts the number of the vertexes; while we preferred to count the length of the path in terms of edges.

With Cypher the query is written as shown below:

```
OPTIONAL MATCH p = shortestPath((startCompany:Enterprise { codice_fiscale:
    '80147930150' })-[e:MEMBER_OF*..999]->(targetCompany:Enterprise {
    codice_fiscale: '08800160964' }))
RETURN p, LENGTH(p);
```

The OPTIONAL MATCH statement is used so that the result contains a null if there are not paths found. The path to be searched is defined as usual, however the function *shortestPath()*, called on the path pattern specified, is the way by which Neo4j knows that is has to compute and return only the shortest path. Again, together with the path, we also return its length.

In OrientDB the query is implemented as shown below:

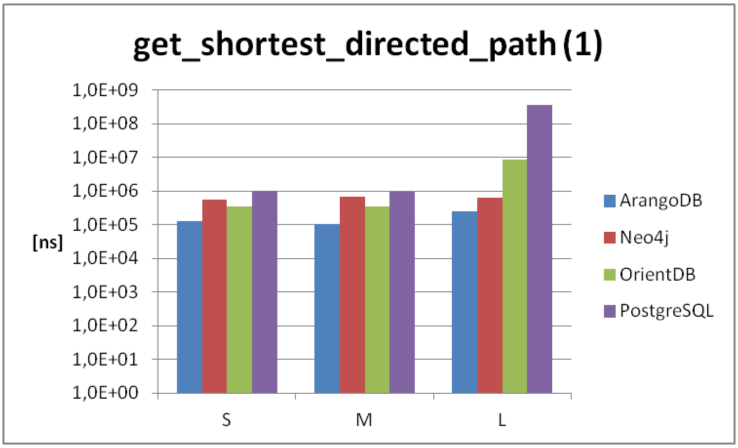
```
SELECT expand(shortest_path)
FROM (
  SELECT shortestPath(
    (SELECT FROM Enterprise WHERE codice_fiscale = '80147930150'),
    (SELECT FROM Enterprise WHERE codice_fiscale = '08800160964'),
    'OUT',
    'member_of',
    {"maxDepth": 30}
  ) AS shortest_path
)
```

Similarly to ArangoDB, also in OrientDB there is a specific syntax for searching the shortest path, which is encoded as a function call with specified arguments. Such arguments are about the two involved nodes (which are searched by their *codice_fiscale* value rather than passing their RIDs); the edge directions, the edge type, and a parameter specifying the max length for the path. Since such functions returns a list of RIDs, the *expand()* function is called on it by an outer SELECT statement.

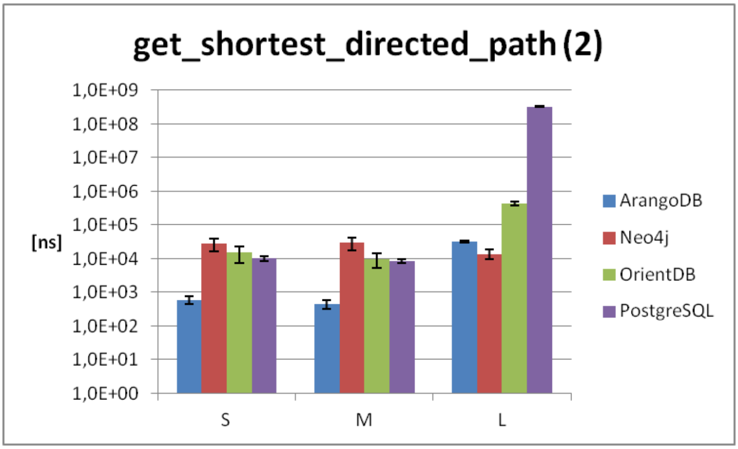
In PostgreSQL the query is written as shown below.

```
WITH RECURSIVE search_path (src_company_denom, dst_company_denom, path_CFs,
    hops, cycle_detected) AS
(
    SELECT
        src_company_denom,
        'not_already_retrieved'::varchar(305),
        ARRAY[src_company_cf, dst_company_cf]::character(16)[],
        1,
        src_company_cf = dst_company_cf
    FROM
        companies_and_memberships.company_memberships_view
        WHERE
        src_company_cf = '08800160964'
  UNION ALL
    SELECT
        f.src_company_denom,
        d.dst_company_denom,
        (path_CFs || d.dst_company_cf)::character(16)[],
        f.hops + 1,
        d.dst_company_cf = ANY(f.path_CFs)
    FROM
        companies_and_memberships.company_memberships_view d,
        search_path f
    WHERE
     f.path_CFs[array_length(path_CFs, 1)] = d.src_company_cf
    AND NOT f.cycle_detected
)

SELECT *
FROM search_path
WHERE path_CFs[1] = '06194870017' AND path_CFs[array_length(path_CFs, 1)] =
    '03747000408'
ORDER BY hops -- this orders paths found by number of hops
LIMIT 1; -- this selects only the first path
```

The query uses a view defined so that, together with the *codice_fiscale*, also the *denominazione* value is collected. It uses the WITH RECURSIVE statement and expands all the paths that start from the given node, checking that no cycles are present. At the end, it selects the path which reaches the desired node. It clearly is sub-optimal, because it does not stop if one of the



(a) First query



(b) Following queries

Figure 5.20: Query get_shortest_directed_path charts.

explored path already computed reaches the desired node; so one possible optimization may be this one. Such optimization may be done by initializing an additional field, called *destination*, to the *codice_fiscale* value of the destination node; and by initializing another new field *reached* to 'false'. At each recursive step, thus for each new *codice_fiscale* retrieved, if it is equal to the value on the *destination* field (which remains the same for each iteration) then the *reached* field may be set to 'true' and then a check on such field would stop the exploration. However, it would only stop the exploration on the current path found, and the expansion of other paths would be performed. Again the lack of global information does not allow to stop the exploration based on run-time results, and thus does not allow to implement the algorithms of graph theory that face such problem.

Obviously, in alternative to the view, we could have worked only on the relationships table, as done in previous queries, and thus the path would have been made of *codice_fiscale* values.

```
CREATE VIEW companies_and_memberships.company_memberships_view AS
SELECT
    company_1.codice_fiscale AS src_company_cf,
    company_1.denominazione AS src_company_denom,
    company_2.codice_fiscale AS dst_company_cf,
    company_2.denominazione AS dst_company_denom
FROM
    companies_and_memberships.companies AS company_1
    JOIN companies_and_memberships.memberships AS memb ON
        (company_1.codice_fiscale = memb.socio)
    JOIN companies_and_memberships.companies AS company_2 ON
        (memb.partecipata = company_2.codice_fiscale);
```

The length of the shortest path in the lightweight case (S) is 5; in the intermediate case (M) the number is 120; in the heavy case (L) is 1815. The times required to resolve such query for the three situations are shown in Figure 5.20. It emerges that PostgreSQL, even though the query is suboptimal, performs well on both S and M cases and on both first execution and following executions; however, for the heavy case (L), the query displays its inefficiency. On the other hand, ArangoDB performs better than the others on all cases, apart from the L case at operating speed, given that Neo4j goes faster.

Even here, PostgreSQL do not uses big amounts of RAM (about 180MB), rather uses a quite big amount of disk space (about 4.5GB). ArangoDB as usual uses 5.4GB of RAM due to the indexes kept in memory; Neo4j uses about 2.8GB of RAM and OrientDB about 2.2GB.

5.5.10 Summary

Warm-up

The first thing that stands out is that, despite we performed a preliminary warm-up of the cache, for all the databases we see a great difference between the times of the first execution of a query, and the times of the following executions. This means that an additional form of "cache level" should be present, which goes beyond the caching of the full data collection. We suppose this is given by both the fact that data of interest is brought to a "higher level" of cache (in Java it may means that data is considered fresh by the garbage collector), and a query plan caching.

While it is unclear whether ArangoDB and OrientDB implement a query plan caching mechanism (we instead do know they implement a result cache mechanism [30, 86], which we have disabled); we know that Neo4j implements it, and it can be disabled. On the other hand, we know that PostgreSQL implements a query plan caching mechanism, but it is only applied when queries are written with prepared statements or procedures, as explained in Section 3.2.4.

Query times

By looking at the charts, it emerges that ArangoDB generally performs better then the others, especially for the (S) and (M) cases.For the (L) cases, instead, Neo4j seems to work better. However we are talking about a graph database (ArangoDB) which slightly goes far from the philosophy of "durability-for-everything" of relational databases and also other graph databases; rather it goes to the direction of an "in-memory" database. The main reasons for its better performance for the (S) and (M) cases is probably due to the fact that, being written in C++, it allocates main memory so that data is effectively always kept available. With Neo4j and OrientDB, instead, being written in Java, the garbage collection may consider such data "old" at a certain moment after the warm-up; and thus such data may not be "ready" when the query asks for them. So one critical configuration parameter for further tuning Neo4j and OrientDB would regards the JVM configurations for the Garbage Collector. The main reason for the better performance of Neo4j in the heavy case (L) is probably due to the fact that index (or similar forms of) lookups are completely avoided due to the presence of the pointers to the linked data of interest.

In addition, it can be seen that a typical pattern can be recognized, which is the fact that often happens that Neo4j and OrientDB are quite "paired", especially for (S) and (M) cases. For the (L) cases, instead, sometimes goes better Neo4j and other times goes better OrientDB.

One important aspect is that PostgreSQL typically performs well when we talk about (S) and (M) cases, or when we talk about the basic queries of searching nodes in few levels of distance. On the other hand, it takes more time than the others when it has to work with a big amount of connected data (L).

Query writing

In our opinion, Cypher is the most intuitive language for graph queries development; its graph pattern matching syntax is the easiest way to write a query on a graph.

AQL, even though it also is a declarative language, it displays a syntax which let it seems near to a procedural language. In particular, it does not work on pattern matching like Cypher, rather it is focused on handling data collections. This is surely due to its multi-model nature, where documents and collections represent the way by which data is conceptually organized. However, AQL was the only query language that allowed to formulate all queries chosen; so it seemed to be the one with greater power of expressiveness.

On the other hand, the extended SQL of OrientDB clearly has the benefit of being near to the SQL language we are all familiar with; however the possibility to express graph queries by means of three different statements can cause some confusion. However, apart from this aspect, another weak point is that it often exposes on results the internal identifiers of the records, while one typically prefers a result made of valuable data. In addition, for what I saw, there is no way to perform an interrogation (with the MATCH or TRAVERSAL statements) which is targeted to only a specific level of depth. The suggested method for MATCH provided in documentation (that is by imposing a couple of *where* and *while* clauses which restrict the valid depth values) seemed not to work, because it returned all the results till the max depth level specified.

For what regards SQL, it is not a graph query language, and in this domain such aspect is felt very well. It provides a way for working with tree-structured and graph-structured data, however such method is suboptimal due to the difficulties on defining traversal conditions. We think that there are limited possibilities for writing better graph queries with SQL than the way provided by our queries. The enhancement would be brought by implementing them via (quite complex) PL/pgSQL stored procedures, or by high-level languages which would have to manage all the query logic.

We also highlight that, as soon as we start to perform traversal queries on a relational table - both with a fixed number of edges traversed, and with a parametric number - the queries' bodies become very large. So while the same operations can be requested by a graph query language, for a relational database several rows have to be written so that a similar operation is described. This is then one of the reasons why graph databases could be taken in consideration when an application works on graph data: the simplified implementation of queries can bring reduced development times; reduced probability of mistakes; and also leave more room for reasoning about other and more complex queries. In this way, the graph query language could become a starting point for the exploration of new query possibilities, which with a relational database may have been less visible.

For what concerns the implementation given for some of the Neo4j's queries, we recognize that the java code provided by us can be significantly improved. One particularly costly operation is the concatenation of strings made within some appending operations done on the StringBuilder object. The highlighting of such bottleneck emerged only on a later moment. Other improvements could be done on the choice of the data-structures used for storing other kind of data results (like the counting of different ancestors per level). Such improvements are suggested based on a particular attention on the dynamics of the Java garbage collection.

We however conclude by stating that APIs are generally more powerful and versatile than the provided query languages. This seems to be trivial, but it is not: only by testing them it can be discovered the presence of eventual pitfalls or limits, or again the complexity degree for their usage. The provided query languages, however, are the most rapid way for performing queries, most of all for those which display a restrained level of complexity.

Chapter 6

Displaying the resulting subgraph of a query

In this chapter we will discuss about how to handle the problem of being able to reconstruct the graph structure of the result set of a query.

Queries shown in the Chapter 5 lose the graph structure of the result. Consider the query where the descendants of a node are searched: the result is made of a flat list of vertexes; the information about involved edges (or paths), and thus the structure of the graph, is not present. In order to allow an application - which receives data from the database - to be able to render the resulting subgraph of a query, some additional information should be returned. The fact of being able to return enough data so that it is possible to draw the graph structure among the result nodes obtained, in fact, would be of interest for some InfoCamere applications like *BRACCO* and *ri.visual*.

The first approach that could come in mind is to return, together with the descendant nodes found, the paths which reach them. Note that, in a general situation, for the same descendant node there may be multiple paths that end on it; thus for each node we need to return every existing path so that we do not lose some of the edges. This time we are then in a situation that is different from the one of *get_distinct_descendants_and_level* query, because multiple paths to the same node have to be detected and returned, and not discarded. Because of this, one may think that, this time, there is no need to expressly force a breadth-first exploration order, and that the default behavior (depth-first order with unique edge/vertex per path) may work well. This is generally not true because of the same reasons of the query *get_distinct_descendants_and_level*: in graph portions where the situation is quite complex - both in terms of nodes number, and in terms of interconnections among them - the approach may again become impractical; in particular, the query may not end in useful time. So the following Cypher statement:

```
MATCH path = (startNode:Enterprise {codice_fiscale:
    '00102681419'})-[m:MEMBER_OF*1..999]->(descendant:Enterprise)
RETURN descendant, path
LIMIT 200000;
```

will response in acceptable time for those nodes that do not have so much interconnected nodes; however, for the heavy situation (L) of the query *get_distinct_descendants_and_level*, such Cypher statement launches a query that does not end even after minutes.

The approach provided by the *get_distinct_descendants_and_level* query was to force a global uniqueness constraint for vertices; however, such solution would not be valid for the current query. In fact, such constraint implies that, if a second path is found to pass through a node already seen by a previously computed path, the second path would not be returned. Please take in mind that, in simple terms, paths are iteratively constructed by adding pieces (edges/nodes) to the path found in the previous iteration; and are not rebuilt from the initial vertex each time, even if they are returned on the result like that. However, by using such constraint we lose some of the paths present within the source graph; so another approach has to be followed.

Let's see it with the help of some figures. Suppose we have a situation like the one showed in Figure 6.1, which can be seen as the resulting subgraph made of the descendants of the *A* node.

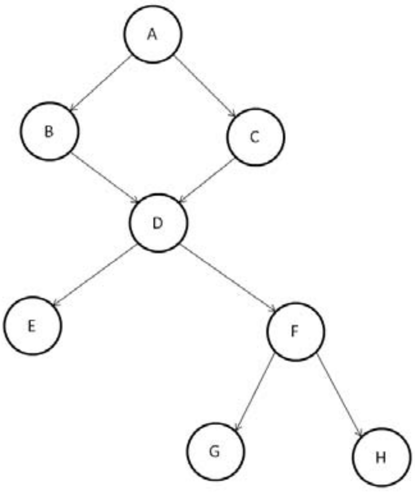


Figure 6.1: Example figure for a subgraph of descendants.

Suppose we do not have any kind of constraints defined for the exploration of the graph; the paths obtained by the two traversal strategies would be like the ones showed in Figures 6.2

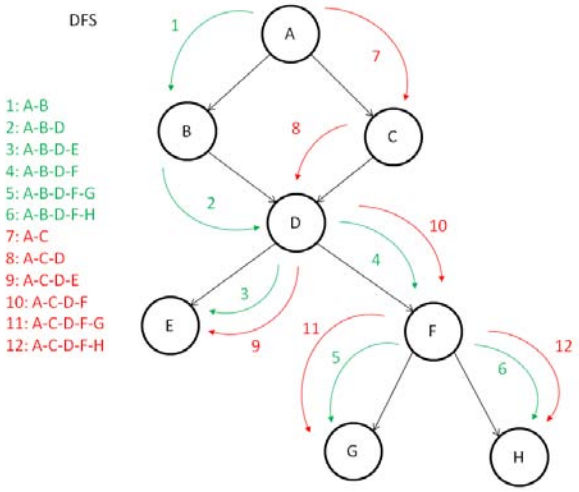


Figure 6.2: DFS order on the example subgraph.

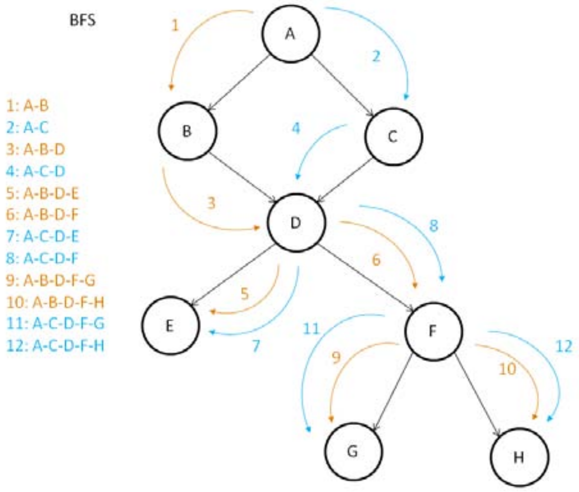


Figure 6.3: BFS order on the example subgraph.

and 6.3. Because of the fork, two paths reach the each node from D to H; the only thing that changes between the two approaches is the order by which paths are found.

Now suppose there is a further edge that goes from the E node to the F node. Because of it, the subgraph made of the nodes F, G and H (and related edges) would be explored a third time. And now suppose we have another edge on A, which goes to itself; then the number of the overall possible paths would become twice the previous number, and the subgraph (F,G,H) would be explored a fourth time. Without constraints, we soon end up in a situation where a lot of graph areas are traversed multiple times, even though only the initial part of the paths changes. There are then some computations that could be reduced. Furthermore, note that the complete absence of constraints would bring the query to never end, because with the hypothetical initial cycle on the A node would generate an infinite number of paths, because multiple loops on such edge may be done.

By forcing the vertexes uniqueness with *path* scope, the situations with cycles (detected by the presence of the same node more then a time on the path retrieved) are avoided. So, supposing to only have the loop on the A node as a modification of the situation showed in Figure 6.1, such constraint would generate, instead of the infinite number of the previous case, the same number of results as the one showed by the two Figures 6.2 and 6.3. In this way, however, we again explore the subgraph (D...H) as many times as the number of different paths that reach D.

On the other hand, a global uniqueness constraint for vertexes would have avoided the exploration of the edges from number 8 to 12 (both included) for the DFS case; and the edges [4,7,8,11,12] for the BFS case. This because D has already been visited during a previous phase of the exploration. Such thing implies that, for node D, only one path would be returned; and then the edge from C to D would be lost, together with the path that goes from A to D. In addition, also for nodes "under" D will be returned with a single path.

So we are in the situation where:

- the lack of constraints brings endless query execution if cycles are present;
- the uniqueness constraint on vertexes at "path level" allows to retrieve all the paths to each single descendant node, avoids cycles, but it performs computation of paths also on already explored sub-graphs, which cause huge workload in complex situations;
- the uniqueness constraint on vertexes at "global level" avoids the exploration of already visited graph regions; however only the first path found is returned for each descendant.

None of the previous constraints allows to also maintain the graph structure on the result set while answering in reasonable time.

The solution is then to reason about the adoption of a constraint on edges.

By using an edge uniqueness constraint at "path level", we would avoid all paths that are made of cycles. However, these cycles are a subclass of the cycles avoided by the constraint on vertexes; in fact, it avoids only cycles where the same edge is found twice within the path, which basically consists on finding two times the two extremal vertexes of such edge. With reference to Figure 6.1, we than have that the number of paths found is the same shown by the two Figures 6.2 and 6.3. If the hypothetical looping edge would be present on the A node, it would only double the number of total paths, but the query would end its execution. So again, this constraint ensures the query to not cycle indefinitely; however it explores multiple times all subgraph areas reachable by multiple paths.

In order to avoid multiple visits on the subgraph made of nodes D,...,H, the edge constraint should be imposed at "global level". Indeed, only the "edge visits" from 1 to 8 would be explored in Figure 6.2; and similarly, edges 7,8,11,12 would not be visited in Figure 6.3. What changes, with respect to the case of global uniqueness on vertexes, is that the edge from C to D is, this time, not only traversed, but also returned.

Since this approach also returns the "missing edge", which was the one from C to D, we now have that we return not all paths, but all edges; and by having all edges we are then able to reconstruct the graph structure of the result. In fact, if for each descendant node we have the information about what are the father nodes from which paths come to it, we can draw the edges from such father nodes to the descendant, and by iteratively doing the same on its fathers, we will end up with the reconstruction of the underlying linking structure. Furthermore, in order to not lose the properties defined on edges, rather then returning what we call "the previous node", we may return the last edge found, so that both nodes and edge properties are present in the result set.

Summarizing, we then have two ways to get enough data on the result set so that the linking information is preserved, which are:

- returning all paths, for each descendant node, by forcing the uniqueness constraint for nodes at "path level";
- returning all last edges, for each descendant node, by forcing the uniqueness constraint for edges at "global level".

The advantage of returning the entire paths is that the single descendant node already has the information about all the paths that reach it, and thus eventual property values defined on the edges of the path would be already present. In this way, successive elaborations would cherish the fact that such data do not have to be rebuilt, so computations on entire paths would be quick.

In contrast, the advantages of returning the second-last node (or the last edge, so that also edge properties are given) are about a great speedup of the query's execution time and a great lightening of the resulting data. However, the disadvantage of this approach is that, if the reconstruction of the paths to the source node is of interest, it would require additional work before such information is available.

An additional observation is the following: if it could be possible to express more than one uniqueness constraint, we could then restrict even more the exploration. The main interest for this would be about using the *global* constraint for edges, and the *path* constraint for nodes.

This would even more reduce the result set by discarding those paths that present some cycles (if the purpose of the query desires to avoid them by the result set).

Let's now see what happens by combining the different constraints on the subgraph made of the descendants of the node '00102681419', showed in Figure 5.10. By supposing it is possible to define more than one constraint, and since there are three definable constraints for nodes, three for edges, and two exploration strategies, the number of combinations is $3 \times 3 \times 2 = 18$.

Strategy	**Uniqueness on edges**	**Uniqueness on nodes**	**Number of results**
BFS | DFS	none	none	infinite
BFS | DFS	path	none	33
BFS | DFS	global	none	13
BFS | DFS	path | none	path	16
BFS | DFS	global	path	12
BFS | DFS	global | path | none	global	11

Table 6.1: Constraints combinations on node '00102681419'

Let's now see how the query for returning the "previous node" can be implemented.

The default exploration approach for ArangoDB is by using a depth-first order, the uniqueness constraint with "path scope" for edges, and none uniqueness constraint for vertexes. AQL allows to define the three kinds of constraint presented so far for both edges and nodes, the exploration order, and also allows to define both uniqueness constraints on edges and vertexes together.

```
LET startVertex = 'Enterprise/00102681419'
LET minLevel = 1
LET maxLevel = 999
FOR v, e, p
    IN minLevel..maxLevel
    OUTBOUND startVertex
    GRAPH 'enterprises_and_members_Graph'
    OPTIONS {bfs: true, uniqueEdges : 'global'}
    //OPTIONS {bfs: true, uniqueEdges : 'global', uniqueVertices :
        'path'}
    LIMIT 200000
RETURN {'descendant': v, 'previous_node': p.vertices[-2]}
//RETURN {'descendant': v, 'previous_node': p.edges[-1]}
```

The commented rows show that it is possible to force more than one constraint, and also show how to return the last edge instead of the previous node.

Talking about Neo4j, its default exploration strategy, as already explained, is in depth-first and with path uniqueness constraint for nodes. Cypher only allows to define queries which work like this; by using the Java Traversal API however the following constraints can be set:

- NODE_GLOBAL: A node cannot be traversed more than once.
- NODE_LEVEL: Entities on the same level are guaranteed to be unique.
- NODE_PATH: For each returned node there's a unique path from the start node to it.
- NODE_RECENT: This is like NODE_GLOBAL, but only guarantees uniqueness among the most recent visited nodes, with a configurable count.
- NONE: No restrictions.
- RELATIONSHIP_GLOBAL: A relationship cannot be traversed more than once, whereas nodes can.
- RELATIONSHIP_LEVEL: Entities on the same level are guaranteed to be unique.
- RELATIONSHIP_PATH: For each returned node there's a (relationship wise) unique path from the start node to it.
- RELATIONSHIP_RECENT: Same as for NODE_RECENT, but for relationships [78].

It seems that only one constraint can be used at a time.

The Cypher code snippet reported at the beginning of this section shows how to retrieve all paths. For the retrieval of descendant nodes and their previous nodes, the query is written as showed below.

```
try ( Transaction tx = graphDb.beginTx() ) // execute within a transaction
{
    // Find the source node
    sourceNode = graphDb.findNode( Labels.Enterprise , "codice_fiscale", c_f );

    // Describe how to retrieve the desired nodes
    resultNodes = graphDb.traversalDescription()
                .breadthFirst()
                .relationships( RelationshipTypes.MEMBER_OF, Direction.OUTGOING
                    )
                .evaluator( Evaluators.excludeStartPosition() ) // DO NOT TURN
                    it to .all(), because you will get a NullPointerException!
                .uniqueness(Uniqueness.RELATIONSHIP_GLOBAL);
     traverser = resultNodes.traverse( sourceNode );

     // Extract the result
     for ( Path path : traverser )
     {
         StringBuilder sb = new StringBuilder();
         sb.append("    path_length: " + path.length() + ", codice_fiscale: " +
             path.endNode().getProperty( "codice_fiscale") + ", previous_node: "
             + path.lastRelationship().getStartNode().getProperty(
             "codice_fiscale") + "\n");
         ++resultsCounter;
         System.out.print(sb.toString());
     }
}
```

Here it is also showed how to eventually access properties defined on the last edge.

OrientDB, when using MATCH or TRAVERSE statements, imposes global uniqueness to vertexes; so it is not possible to collect all previous nodes for each descendant, because only the first retrieved one is returned. The other query way, that is by using the SELECT statement, would return nodes without checking their uniqueness; however, it presents all the drawbacks already explained for its usage in graph traversal queries.

A quick search about the topic has been done also on Java Graph and Traversal API documentation pages [89, 93] and also on Gremlin documentation [129, 130]. However, at the time of writing, its seems there are not methods that allow to specify, while performing the traversal, different kinds of constraint (as done by Noe4j). So the implementation of this kind of query would require some efforts for the developer.

In PostgreSQL, since we did not found a way to express global constraints while writing SQL queries, the implemented way for returning the pairs $(descendant, previous_node)$ is by exploiting the traversal part of the queries explained before, and to collect, during the exploration, the previous node for each node found. However, such information was already collected by the *socio* field, so we only had to return its value together with the *partecipata* value representing the descendant node reached. However such query implements cycles avoidance, which corresponds to unique vertex per path as already explained, so some of the results would not be returned. The strategy for cycles avoidance, on the other hand, cannot be removed because the execution would loop indefinitely.

Resuming, two ways for returning the graph structure in a query result are possible. The first is the one that returns all the paths from the initial node to the reached node; this can be done with both DFS or BFS approach, and with 'none' or 'path' uniqueness constraint defined on vertexes. The second is the one that returns only the last edge of the path that reaches such node; this can be done by both DFS or BFS approach, and with 'global' uniqueness constraint defined on edges, with the eventual 'path' constraint on vertexes.

If path objects are returned by the query languages, they typically already contain, structured in a JSON format, all data about traversed nodes and edges. Actually, OrientDB returns its internal identifiers (RIDs) when the $path system variable is returned, so one has to expand such list, which in turn will returns only vertexes, so a more complex query should be implemented in order to have both vertexes and edges data.

As obvious, when we use a different approach for the implementation of the query (such as a Java native API or stored procedures), the developer has to give a JSON format to the obtained results, so that its transport can be done effectively and the application can again exploit such result structure for extracting data.

Chapter 7

Conclusions

The main goals of this work where to investigate the field of graph databases, to understand their peculiarities and to compare them with the well established relational database. In particular, we were interested in evaluating how they would perform on a dataset that represents relationships among companies. The attention was focused on gaining fast query responses; on the ease of development of queries; and on looking for new opportunities that such technology may bring.

For these purposes, we started by giving some notions about graph databases' characteristics, we analyzed their data models, and we looked at the differences between them and graph computing engines. We also analyzed, with a good level of detail, what are the fundamental characteristics of three of the currently available graph database products. We then tested them on a real dataset, with the additional comparison with a relational database, so that we have an overview of their performance differences. Now, we gather the conclusions of our work.

We start this chapter by giving a quick assessment of the main topics of interest for the four DBMSs compared; we then report the main conclusions of the work; we give some final considerations on the performance results obtained; and at the end we give some hints about future developments.

7.1 DBMSs assessment

We will assign some grades for each DBMS compared, with regards on the following topics: *performance*, *expressiveness of query languages*, *provided APIs*, *resources consumption*, *licensing*, *data import time*, and *documentation quality*. At the end, we will resume the assessment by summing such values, with double weight for the grade related to the performance obtained at query time. Grades will range from 1 to 10, were 10 is better. The value which represents the "sufficiency" is 6.

By looking at the *performance* showed by the three graph databases, we could state that ArangoDB typically answers to the query in less time than the others, especially for the (S) and (M) cases. Sometimes the difference is of an order of magnitude; other times (like for the *get_common_descendants* query) we have seen that on the "heavy node" (L) it performs worse than the others. In any case, we could say that a pattern made like this can be seen on most of the queries: ArangoDB performs better; Noe4j and OrientDB are quite aligned, even though OrientDB typically performs better than Neo4j on simple queries, while Neo4j typically performs better on complex queries and on queries that follow the first one; and PostgreSQL varies its performance accordingly to the query typology, but for the heavy cases (L) it typically requires more time than the others. For these reasons, we assign the grade 9 to ArangoDB, 8 to Neo4j, 7 to OrientDB, and 5 to PostgreSQL.

However, the results obtained should be taken with a good measure of care. In fact, the times collected are dependant of first of all the implementation of the query; then on the dataset; then on the configuration of each database server; and then on the hardware architecture.

Talking about *query languages*, in all three, with different weights, can be seen the influence of SQL; however they are quite different from each other and this aspect made the formulation of the same query for each of them a bit hard.

We saw that AQL is the most complete graph query language; Cypher is the most intuitive; OrientDB'SQL is near to the standard SQL and adds some traversal statements (even though the development of queries is sometimes not immediate); and PostgreSQL's SQL needs very big query bodies when a traversal is specified.

For these reasons, we assign the grade 9 to AQL, 8 to Cypher, 7 to OrientDB's SQL and 5 to PostgreSQL' SQL.

For what regards the *APIs* provided by the DBMSs, while both Neo4j and OrientDB can run embedded in a JVM instance and accessed by their native Java API, ArangoDB (being written in C++ and JavaScript) cannot. However, with ArangoDB Foxx and the possibility to define JavaScript user functions which extends AQL, it can be implemented procedures performed directly within the database, as provided by the other two graph databases. However while AQL is more about database access and queries management, user defined functions are the provided way for implementing more complex queries. Also Noe4j and OrientDB allow to define stored procedures and functions.

The only API which we have seen quite in depth is the one provided by Neo4j, which is simple and well structured; it also provides methods for the definition of graph traversal with different constraints. The API provided by OrientDB is heavily based on TinkerPop Gremlin, which seems not to provide a way for quickly forcing the uniqueness constraints we have seen on our queries. As seen, ArangoDB, Neo4j and OrientDB all provide also a HTTP API, so that the database can be accessed and queried remotely.

For PostgreSQL, even here there is not full access to the native library of the DBMS. However, it provides the possibility of implementing stored procedures or to access the database by several

API drivers.

For the reasons given above, we assess with the grade 7 ArangoDB, with the grade 9 Noe4j, with the grade 8 OrientDB and with the grade 7 PostgreSQL.

As for as the *disk space usage* is concerned, we have seen that Neo4j and PostgreSQL are the ones that need less disk space for storing data. However, PostgreSQL needs quite big amount of disk space when solving complex queries on heavy data situations. ArangoDB is the third classified for disk space usage, and OrientDB is the fourth.

For these reasons, we assign the grade 9 to ArangoDB, the grade 10 to Neo4j, and the grade 8 to both OrientDB and PostgreSQL.

For what concerns the *RAM usage*, we have seen that ArangoDB is the one that most heavily needs enough RAM for being well operative; then we have Neo4j and OrientDB; and at the end we have PostgreSQL, which typically requires less RAM than the others.

For these reasons, ArangoDB is assessed with 7, Noe4j and OrientDB with 9, and PostgreSQL with 10.

Another important aspect is *licensing*. We did not reported licensing prices, however we have seen what are the licenses for the products' usage. All three graph databases provide a community edition of their product which allow free use, however ArangoDB and OrientDB license such edition with the Apache v2 license (so that it can be used also for commercial purposes); while Neo4j releases it under the GPL v3 license, which blocks the usage for commercial purposes. We have seen that they also come with an Enterprise edition with additional features, commercial license and comes with a subscription fee. Within this field, we can say that ArangoDB and OrientDB have pricing quite aligned and below the price required by Neo4j. PostgreSQL instead is released under a BSD-like license, which allows free use of it for both non-commercial and commercial purposes.

For these reasons, the grades are: 9 for ArangoDB and OrientDB, 7 for Neo4j and 10 to PostgreSQL.

We now assess the time required for the *import of data*. We have seen that ArangoDB, Neo4j and PostgreSQL required comparable time and under the 10 minutes threshold; while OrientDB required a lot more. However, not sure if this may be due to our mistakes, we do not assign a too low grade to OrientDB.

The grades are then 10 to ArangoDB, Neo4j and PostgreSQL, and 7 to OrientDB.

The last relevant aspect here considered is the *documentation quality*. We perceived a good quality for both the documentations of Neo4j and PostgreSQL; even though they could be even enriched by more examples. A good documentation is also the one provided by ArangoDB, it gives less material than the others, but it is clear and well-structured anyway. For the documentation of OrientDB we perceived a bit of confusion, it could be structured better and some mistakes on the example code snippets reported would help in reducing confusion.

Because of the reasons provided, we assess ArangoDB's documentation with the grade 9; Neo4j's and PostgreSQL's documentation with 10; and OrientDB's documentation with 8.

	ArangoDB	**Neo4j**	**OrientDB**	**PostgreSQL**
Query performance (2x)	9	8	7	5
Query language	9	8	7	5
API	7	9	8	7
Disk space usage	9	10	8	8
RAM usage	7	9	9	10
Licensing	9	7	9	10
Import time	10	10	7	10
Documentation	9	10	8	10
TOTAL	**78**	**79**	**70**	**70**

Table 7.1: Assessment table of the compared DBMSs.

We see that Neo4j reached the highest score, even though ArangoDB is under it for a single point. ArangoDB wins in performance, while Neo4j generally wins on the majority of the remaining aspects. OrientDB is ranked third among the graph databases; and it is penalized for the reduced expressiveness and ease-of-use of its query language, for the documentation and the import time, which lead to be ranked equally with PostgreSQL.

Anyway, these assessments may be considered quite subjective, so again such values should be taken critically.

7.2 Main conclusions

Rather than simply competing for what is the DBMS better ranked, we would like to highlight some other points. The most important thing that should emerge by this work is that graph databases effectively help when working with graph data, and the main contributes to this are the purposely designed storage techniques, and the functionality given by the query languages (or the APIs) provided. The storage techniques designed for working with graph data, in fact, bring good performance on such domain; the functionality provided by the graph query languages (or the APIs) bring a better management of the typical graph exploration problems. In particular, we saw that the possibility to define the order and the constraints for the traversal operation brings significant added value to the DBMS, because it makes possible to explore the graph in an efficient way and by avoiding those problematic situations that may arise on a graph domain. This aspect becomes even more important when, as seen in Chapter 6, a query that returns a big portion of the graph is performed.

Talking about the results obtained, we state that: based on the dataset given, the kinds of query chosen, the ways by which queries have been implemented, the configuration of the servers given, the configuration of the machine given, and the testing policy, graph databases generally perform better than the relational database, especially for complex queries on heavy data situations.

Resuming, we conclude that:

- since graph databases directly store relationships among entities with a dedicated construct, and since they provide an efficient way to link them to the entities, they avoid those costly operations (JOINs) that relational databases have to perform at query-time to reconstruct them.
- graph databases show to be useful in terms of performance, especially for complex traversal queries on complex graphs; i.e. when a big amount of nodes are heavily connected.
- graph databases show to be useful in queries development: the provide query languages facilitate the implementation of the queries on graph data and are generally thought to handle graph-related problems.
- for simple queries, a relational database is good enough: both in performance and in queries simplicity.
- graph databases are not the right choice for queries that consider the whole graph; graph computing engines are the appropriate solution for it.

7.3 Final considerations

The query times obtained show good performance results by a high point of view. However, they are dependant on the configuration parameters and the hardware specifics. So relative improvements within the performance of the DBMSs may be possible with different configurations. The first two aspects where to search for better performance should be on the DBMSs' configuration and on sizing the available RAM.

Better configurations may be provided for the servers; our not deep knowledge on the field may have brought to not the best configurations possible. Some test with different configurations would be useful for understanding the best configuration possible for each product with the dataset provided.

Talking about RAM sizing, we saw that the most heavy database requires about 6 GB of space for data, indexes and other potentially useful files. Given that current costs for RAM are not so high and also continuously lowering, the deployment of a machine with 8GB of RAM could entirely contain "all graph" data without prohibitive costs. About 1GB would be left to the OS, and the other 1GB would be used for additional indexes on other attributes often used (like the *denominazione* field, over which a full-text index or similar should be created).

An alternative solution to the employment of a lower amount of RAM would be the one that uses a SSD disk for the persistent storage. This way, even if the graph is not entirely present in cache at query moment, it will be retrieved with short delays. However, given the fact that also queries are not simple first-level neighbours investigation, and that they may perform the retrieval of a big amount of nodes - also involving filtering and sorting - the adoption of the solution with a bigger RAM size should help in reaching the interrogation result more quickly.

The last operation that could be done for speeding up queries' execution is the employment of a CPU with a bigger operating frequency. While having a bigger number of threads would help in speeding up simultaneous queries, the enhancement on the CPU frequency would speed-up the execution of each single query.

In any case, the small growth rate of the dataset shown so far (and the nature of the data treated) permits the solution with a big enough RAM to be applied without those worries that are typical of those scenarios where the dataset may be subjected to a huge and improvise growth. So the solution adopted for the deployment of an application based on graph databases would quite surely follow this approach.

However, we are confident that the obtained results already show what is the order of magnitude of the queries' execution times presented for such dataset.

7.4 Future developments

The future developments of this work will be about studying and sizing the best architecture for the deployment of a graph database at the basis of some existing applications. Supposing the deployment for BRACCO - that is the application we currently have more information about the dataset, its behaviour and the kind of users that access it - we propose how its architecture could be changed so that a graph database may be employed.

The current approach followed by the application expects that data come from XML files generated by insert/update/delete operations performed on the Italian Business Register. Such files are then elaborated and data are extracted and stored on a relational database. Our proposal is to use the same architecture but by substituting the relational database with a graph database. Another approach would be to keep the relational database and use a wrapper for the synchronization of its data to the graph database; this in fact would probably be the simplest solution, given the fact that there already exist some wrapping techniques for relational databases which allow the synchronization to other databases. In addition, for example, OrientDB already provides such tool with the name of Teleporter [97]. However, we are confident that the same import operations performed with the relational database can be applied to a graph database, so we would save a machine instance dedicated to the relational database.

We have seen that the complete import of the entire dataset requires some minutes for ArangoDB and Neo4j; while for OrientDB required a lot more. Since the main operations would influence relationships updates, rather than nodes updates, the proposed solution is by updating only the new data without the need of a complete import. However, this approach have to be tested before its deployment.

Quick tests on full-text indexes on the *denominazione* property have been executed, and their usage implies the need of more disk space for Neo4j and OrientDB, while more RAM space for ArangoDB, because even this kind of index is not persistent, but kept in main memory.

On the other hand, the data growth rate of the dataset is quite limited, so the proposed solution is the one that uses a machine with about 8GB-10GB of RAM, so that there is room for additional data to be cached, and also for letting space for data growth.

Talking about the disk space required, we propose a solution with at least 10GB of space, so that again there is room for the additional indexes created (if persistent) and the future data growth.

However, it is also typically recommended that WAL records are stored on a different disk than the disk used to store the DB content. In this way data I/O operations would not be interrupted by WAL I/O operations [96]. Because of this, another disk should be provided. However, due to the amount of RAM and the solution with two separate disks, we do not consider necessary to have them made with SSD technology.

The number of users allowed to perform the access to the database and the number of queries per second are quite limited, so one single instance server should probably fits the requirements. However, some information about the existing relational architecture would help in sizing such aspect.

In addition, since such application takes data from an external source, that the queries are only read operations, and that the number of queries per second is quite limited, we do not feel the need of a complex distributed architecture. For high availability purposes, we could however deploy a master-slave architecture made of one master and one (or two) slaves.

The application would then benefits from the performance provided by a graph database, and by the additional potentialities for queries development provided by their query languages.

The other immediate future development would be the inclusion of the equity participation shares on the edges that connect relationships, so that new queries working on such values would be implemented.

Another interesting aspect correlated to graph data that may became a future development is the usage of a graph computing engine for the implementation of queries which collect some statistics measure related to the entire graph, like the average number of outgoing edges per node; or the length of the longest path; or the number of root nodes; etc. Such information would represent an added value to the application that would be working with such graph data.

Another important aspect is the one related to RDF. Indeed, since InfoCamere releases every day a lot of title searches about enterprises, the release of some of their basic information (like *codice_fiscale*, *denominazione*, *cittadinanza*, and whether or not it is *out-of-business*) in the RDF format would bring enhanced value to the information it holds and to the information it gives available to the public. The release of such basic information would bring a step forward to the enhancement of the Linked Data movement, and would also allow the implementation of new web-sites or application products that employ such data and gives them an additional dimension to the tabular and flat structure of a classical document. In this way, it would be made quicker and enriched the reading of a title search, because links to other cited enterprises would lead to web-pages (or other kind of documents) holding the basic information about them; so that data would easily be explorable, and would be kept fresh and updated.

In addition, such technology may be useful in the context of the realization of a European Business Register, where basic information about enterprises' relationships would be shared and linked by the business registers of each country; so that it would be possible to realize a web of connections that goes beyond the country borders.

There already exists an effort for providing a sort of European Business Register, which is EBR.org. However, such web-site is a portal that simply addresses each query to the business register of the target country, so there is not a unified database, but a unified access method for more databases. The RDF solution would instead bring a sort of database for those basic information that could be freely provided so that the exploration of the data, belonging to different countries, would be improved.

Bibliography

[1] *"DB-ENGINES.com"*. http://db-engines.com/en/. [Online; accessed 29-March-2017].

[2] *"Discussion on ArangoDB and index-free adjacency"*. https://groups.google.com/forum/#!topic/arangodb/xO0qIcZ6h60. [Online; accessed 29-March-2017].

[3] Alice Hertel, J. B., and Stuckenschmidt, H. *"RDF Storage and Retrieval Systems"*. http://publications.wim.uni-mannheim.de/informatik/lski/Hertel08RDFStorage.pdf. [Online; accessed 29-March-2017].

[4] Amazon. *"What is NoSQL?"*. https://aws.amazon.com/it/nosql/graph/. [Online; accessed 5-February-2017].

[5] Amgad Agoub, Felix Kunde, M. K. *"Potential of Graph Databases in Representing and Enriching Standardized Geodata"*. https://www.researchgate.net/publication/305701542_Potential_of_Graph_Databases_in_Representing_and_Enriching_Standardized_Geodata. [Online; accessed 29-March-2017].

[6] Andy Seaborne, P. C. *"Two graph data models : RDF and Property Graphs"*. http://www.slideshare.net/andyseaborne/two-graph-data-models-rdf-and-property-graphs. [Online; accessed 29-March-2017].

[7] Angles, R., and Gutierrez, C. *"An introduction to Graph Data Management"*. 2015.

[8] APOC. *"APOC user guide"*. https://neo4j-contrib.github.io/neo4j-apoc-procedures/. [Online; accessed 29-March-2017].

[9] ArangoDB. *"AQL documentation"*. https://docs.arangodb.com/3.0/AQL/index.html. [Online; accessed 21-February-2017].

[10] ArangoDB. *"ArangoDB 3.1 documentation - Indexing"*. https://docs.arangodb.com/3.1/Manual/Indexing/VertexCentric.html. [Online; accessed 7-February-2017].

[11] ArangoDB. *"ArangoDB Cookbook - Importing Data"*. https://docs.arangodb.com/cookbook/Administration/ImportingData.html. [Online; accessed 12-December-2016].

[12] ArangoDB. *"ArangoDB documentation - 3.0 Release Notes"*. `https://docs.arangodb.com/3.0/Manual/ReleaseNotes/NewFeatures30.html`. [Online; accessed 17-December-2016].

[13] ArangoDB. *"ArangoDB documentation - Arangoimp"*. `https://docs.arangodb.com/2.8/HttpBulkImports/Arangoimp.html`. [Online; accessed 7-March-2017].

[14] ArangoDB. *"ArangoDB documentation - Architecture"*. `https://docs.arangodb.com/3.0/Manual/Architecture/`. [Online; accessed 25-February-2017].

[15] ArangoDB. *"ArangoDB documentation - Data modeling"*. `https://docs.arangodb.com/3.0/Manual/DataModeling/index.html`. [Online; accessed 17-November-2016].

[16] ArangoDB. *"ArangoDB documentation - Getting started"*. `https://docs.arangodb.com/3.0/Manual/GettingStarted/index.html`. [Online; accessed 17-November-2016].

[17] ArangoDB. *"ArangoDB documentation - Graphs"*. `https://docs.arangodb.com/3.0/Manual/Graphs/index.html`. [Online; accessed 12-December-2016].

[18] ArangoDB. *"ArangoDB documentation - Joins"*. `https://docs.arangodb.com/3.0/AQL/Examples/Join.html`. [Online; accessed 22-February-2017].

[19] ArangoDB. *"ArangoDB documentation - Naming conventions"*. `https://docs.arangodb.com/3.0/Manual/DataModeling/NamingConventions/`. [Online; accessed 7-March-2017].

[20] ArangoDB. *"ArangoDB documentation - Transactions"*. `https://docs.arangodb.com/3.0/Manual/Transactions/index.html`. [Online; accessed 05-December-2016].

[21] ArangoDB. *"ArangoDB documentation - Traversal"*. `https://docs.arangodb.com/3.0/AQL/Graphs/Traversals.html`. [Online; accessed 7-March-2017].

[22] ArangoDB. *"ArangoDB FAQ"*. `https://www.arangodb.com/documentation/faq/`. [Online; accessed 19-December-2016].

[23] ArangoDB. *"ArangoDB GitHub - 3.0 VelocyPack"*. `https://github.com/arangodb/velocypack`. [Online; accessed 17-December-2016].

[24] ArangoDB. *"ArangoDB GitHub - 3.0 VelocyPack Readme"*. `https://github.com/arangodb/velocypack/blob/master/VelocyPack.md`. [Online; accessed 17-December-2016].

[25] ArangoDB. *"ArangoDB subscription page"*. `https://www.arangodb.com/subscriptions/`. [Online; accessed 15-November-2016].

[26] ArangoDB. *"ArangoDB website - Why ArangoDB"*. `https://www.arangodb.com/why-arangodb/multi-model/`. [Online; accessed 12-December-2016].

[27] ArangoDB. *"GitHub project - ArangoDB & Blueprints"*. https://github.com/arangodb/blueprints-arangodb-graph. [Online; accessed 6-March-2017].

[28] ArangoDB. *"Graph Functions"*. https://docs.arangodb.com/3.0/Manual/Graphs/GeneralGraphs/Functions.html. [Online; accessed 26-February-2017].

[29] ArangoDB. *"Issue #392 - Gremlin graph queries for REST"*. https://github.com/arangodb/arangodb/issues/392. [Online; accessed 6-March-2017].

[30] ArangoDB. *"The AQL query result cache"*. https://docs.arangodb.com/3.0/AQL/ExecutionAndPerformance/QueryCache.html. [Online; accessed 6-March-2017].

[31] Babcock, D. *"CS 360 - Analysis of Algorithms - Graph Theory"*. http://ycpcs.github.io/cs360-spring2015/lectures/lecture15.html. [Online; accessed 22-January-2017].

[32] Beckett, D. *"What does SPARQL stand for?"*. http://lists.w3.org/Archives/Public/semantic-web/2011Oct/0041.html, 2011. [Online; accessed 2-November-2016].

[33] Belaid, R. *"Introduction to PostgreSQL physical storage"*. http://rachbelaid.com/introduction-to-postgres-physical-storage/. [Online; accessed 17-February-2017].

[34] Bloor, R. *"The Graph Database and the RDF Database"*. http://insideanalysis.com/2015/01/the-graph-database-and-the-rdf-database/, 2015. [Online; accessed 2-November-2016].

[35] B.N., M. *"Understanding caching in Postgres - An in-depth guide"*. https://madusudanan.com/blog/understanding-postgres-caching-in-depth/. [Online; accessed 22-February-2017].

[36] Brandt, A. *"GitHub project - ArangoDB & Gremlin"*. https://github.com/arangodb/blueprints-arangodb-graph/wiki/Gremlin. [Online; accessed 6-March-2017].

[37] DataStax. *"DataStax Acquires Aurelius, The Experts Behind TitanDB"*. http://www.datastax.com/2015/02/datastax-acquires-aurelius-the-experts-behind-titandb. [Online; accessed 27-February-2017].

[38] David C. Faye, Olivier Curé, G. B. *"A survey of RDF storage approaches"*. Revue Africaine de la Recherche en Informatique et Math´ematiques Appliquées, INRIA, 2012, 15, pp.11-35, hal-01299496.

[39] Donald Feinberg, N. H. *"IT Market Clock for Database Management Systems, 2014"*. https://www.gartner.com/doc/3100219/making-big-data-normal-graph. [Online; accessed 1-March-2017].

[40] Everett, N. *"Investigate ArangoDB for Wikidata Query"*. https://phabricator.wikimedia.org/T88549. [Online; accessed 28-March-2017].

[41] Ferro, N. *Databases course lectures.* Department of Information Engineering - University of Padua, 2014/2015.

[42] G, I. S. *"Introduction of Graph Database"*. http://systemg.research.ibm.com/database.html. [Online; accessed 20-February-2017].

[43] Gordon, D. *"Warm the cache to improve performance from cold start"*. https://neo4j.com/developer/kb/warm-the-cache-to-improve-performance-from-cold-start/. [Online; accessed 10-March-2017].

[44] Heath, T. *"Linked Data - Connect Distributed Data across the Web"*. http://linkeddata.org/. [Online; accessed 17-January-2017].

[45] Herman, I. *"W3C Semantic Web FAQ"*. https://www.w3.org/RDF/FAQ. [Online; accessed 28-January-2017].

[46] Hironobu, S. *"The Internals of PostgreSQL - for database administrators and system developers"*. http://www.interdb.jp/pg/pgsql01.html. [Online; accessed 26-February-2017].

[47] HypergraphDB. *"HypergraphDB Web Site"*. http://hypergraphdb.org/. [Online; accessed 10-January-2017].

[48] Ian Robinson, J. W., and Eifrem, E. *"Graph Databases"*, 2nd ed. O'Reilly, 2015.

[49] InfoCamere. *"Cos'è il Registro Imprese"*. http://www.registroimprese.it/en/web/guest/il-registro-imprese-e-altre-banche-dati#page=registro-imprese. [Online; accessed 14-January-2017].

[50] InfoCamere. *"InfoCamere - Sistema Camerale"*. http://www.infocamere.it/sistema-camerale. [Online; accessed 14-January-2017].

[51] Javastaff.com. *"Intervista a Luca Garulli"*. http://www.javastaff.com/2007/12/11/intervista-a-luca-garulli/. [Online; accessed 24-February-2017].

[52] Jeffrey Dean, S. G. *"MapReduce: Simplified Data Processing on Large Clusters"*. https://research.google.com/archive/mapreduce-osdi04.pdf. [Online; accessed 3-March-2017].

[53] John E. Savage, M. G. W. *"Heuristics for Parallel Graph-Partitioning"*. http://dl.acm.org/citation.cfm?id=864816. [Online; accessed 23-February-2017].

[54] Lovinger, R. *"RDF and OWL"*. https://www.slideshare.net/rlovinger/rdf-and-owl. [Online; accessed 27-March-2017].

[55] Marcus Cobden, Jennifer Black, N. G. L. C., and Shadbolt, N. *"A Research Agenda for Linked Closed Data"*. http://eprints.soton.ac.uk/272711/. [Online; accessed 17-January-2017].

[56] Marek Ciglan, A. A., and Hluchy, L. *"Benchmarking traversal operations over graph databases"*. http://ieeexplore.ieee.org/document/6313678/. [Online; accessed 15-January-2017].

[57] Mark A. Beyer, N. H. *"Making Big Data Normal With Graph Analysis for the Masses"*. https://www.gartner.com/doc/2852717/it-market-clock-database-management. [Online; accessed 1-March-2017].

[58] Max Schmachtenberg, Christian Bizer, A. J., and Cyganiak, R. *"Linking Open Data cloud diagram 2014"*. http://lod-cloud.net/. [Online; accessed 23-January-2017].

[59] Momjian, B. *"PostgreSQL: Introduction and Concepts"*. Addison Wesley, 2001.

[60] Mullane, G. S. *"PostgreSQL mailing list - Database Caching"*. https://www.postgresql.org/message-id/E16gYpD-0007KY-00@mclean.mail.mindspring.net. [Online; accessed 22-February-2017].

[61] Neo4j. *"Compare editions"*. https://neo4j.com/editions/. [Online; accessed 2-February-2017].

[62] Neo4j. *"Data modeling"*. https://neo4j.com/developer/guide-data-modeling/. [Online; accessed 2-February-2017].

[63] Neo4j. *"Neo4j - Licensing"*. https://neo4j.com/licensing/. [Online; accessed 27-February-2017].

[64] Neo4j. *"Neo4j documentation - Clustering"*. https://neo4j.com/docs/operations-manual/current/clustering/. [Online; accessed 23-February-2017].

[65] Neo4j. *"Neo4j documentation - Constraints"*. http://neo4j.com/docs/developer-manual/current/cypher/schema/constraints/. [Online; accessed 8-March-2017].

[66] Neo4j. *"Neo4j documentation - Cypher"*. http://neo4j.com/docs/developer-manual/3.0/cypher/. [Online; accessed 11-December-2016].

[67] Neo4j. *"Neo4j documentation - Importing CSV Data into Neo4j"*. https://neo4j.com/developer/guide-import-csv/. [Online; accessed 8-March-2017].

[68] NEO4J. *"Neo4j documentation - Indexes"*. http://neo4j.com/docs/developer-manual/current/cypher/schema/index/. [Online; accessed 8-March-2017].

[69] NEO4J. *"Neo4j documentation - Performance"*. https://neo4j.com/docs/operations-manual/current/performance/. [Online; accessed 21-February-2017].

[70] NEO4J. *"Neo4j documentation - Query Tuning"*. https://neo4j.com/docs/developer-manual/current/cypher/query-tuning/. [Online; accessed 26-February-2017].

[71] NEO4J. *"Neo4j documentation - Transactional Cypher HTTP endpoint"*. https://neo4j.com/docs/developer-manual/current/http-api/#http-api-transactional. [Online; accessed 20-February-2017].

[72] NEO4J. *"Neo4j documentation - Transactions"*. https://neo4j.com/docs/developer-manual/current/cypher/introduction/transactions/. [Online; accessed 20-February-2017].

[73] NEO4J. *"Neo4j documentation - Use the Import tool"*. https://neo4j.com/docs/operations-manual/current/tutorial/import-tool/. [Online; accessed 8-March-2017].

[74] NEO4J. *"Neo4j download page"*. https://neo4j.com/download/. [Online; accessed 15-November-2016].

[75] NEO4J. *"Neo4j Java API documentation - org.neo4j.graphdb.index"*. http://neo4j.com/docs/java-reference/current/javadocs/org/neo4j/graphdb/index/package-summary.html. [Online; accessed 8-March-2017].

[76] NEO4J. *"Neo4j Java API documentation - Relationship"*. http://neo4j.com/docs/java-reference/current/javadocs/org/neo4j/graphdb/Relationship.html. [Online; accessed 20-February-2017].

[77] NEO4J. *"Neo4j Java API documentation - Transaction"*. https://neo4j.com/docs/java-reference/current/javadocs/org/neo4j/graphdb/Transaction.html. [Online; accessed 20-February-2017].

[78] NEO4J. *"Neo4j Traversal API documentation - Uniqueness"*. http://neo4j.com/docs/java-reference/current/javadocs/org/neo4j/graphdb/traversal/Uniqueness.html. [Online; accessed 18-March-2017].

[79] NEO4J. *"Traversal framework Java API"*. https://neo4j.com/docs/java-reference/current/#tutorial-traversal. [Online; accessed 20-January-2017].

[80] Neubauer, P. *"Should I learn Cypher or Gremlin for operating a Neo4j database?"*. https://www.quora.com/Should-I-learn-Cypher-or-Gremlin-for-operating-a-Neo4j-database. [Online; accessed 6-March-2017].

[81] Noel Yuhanna, Boris Evelson, B. H. E. J. *"TechRadar™: Enterprise DBMS, Q1 2014"*. https://www.forrester.com/report/TechRadar+Enterprise+DBMS+Q1+2014/-/E-RES106801. [Online; accessed 1-March-2017].

[82] Novet, J. *"DataStax acquires Aurelius, the startup behind the Titan graph database"*. http://venturebeat.com/2015/02/03/datastax-acquires-aurelius-the-startup-behind-the-titan-graph-database/. [Online; accessed 27-February-2017].

[83] OrientDB. *"Find the right solution for your business needs"*. http://orientdb.com/support/. [Online; accessed 5-February-2017].

[84] OrientDB. *"OrientDB documentation - Caching"*. http://orientdb.com/docs/2.2.x/Caching.html. [Online; accessed 22-February-2017].

[85] OrientDB. *"OrientDB documentation - Clusters"*. http://orientdb.com/docs/2.2.x/Tutorial-Clusters.html. [Online; accessed 08-January-2017].

[86] OrientDB. *"OrientDB documentation - Command cache"*. http://orientdb.com/docs/2.2.x/Command-Cache.html. [Online; accessed 22-February-2017].

[87] OrientDB. *"OrientDB documentation - Data modeling"*. http://orientdb.com/docs/2.2.x/Tutorial-Document-and-graph-model.html. [Online; accessed 09-December-2016].

[88] OrientDB. *"OrientDB documentation - ETL"*. http://orientdb.com/docs/2.2/ETL-Introduction.html. [Online; accessed 8-March-2017].

[89] OrientDB. *"OrientDB documentation - Graph API"*. http://orientdb.com/docs/2.2/Graph-Database-Tinkerpop.html. [Online; accessed 6-March-2017].

[90] OrientDB. *"OrientDB documentation - Graph Batch Insert"*. http://orientdb.com/docs/2.2.x/Graph-Batch-Insert.html. [Online; accessed 27-February-2017].

[91] OrientDB. *"OrientDB documentation - Graph Schema"*. http://orientdb.com/docs/2.2/Graph-Schema.html. [Online; accessed 20-February-2017].

[92] OrientDB. *"OrientDB documentation - Gremlin"*. http://orientdb.com/docs/2.2/Gremlin.html. [Online; accessed 6-March-2017].

[93] OrientDB. *"OrientDB documentation - Java Traverse"*. http://orientdb.com/docs/2.2.x/Java-Traverse.html. [Online; accessed 18-March-2017].

[94] OrientDB. *"OrientDB documentation - Performance Tuning"*. http://orientdb.com/docs/2.2.x/Performance-Tuning.html. [Online; accessed 7-March-2017].

[95] OrientDB. *"OrientDB documentation - SQL reference"*. http://orientdb.com/docs/2.2/SQL.html. [Online; accessed 13-December-2016].

[96] OrientDB. *"OrientDB documentation - Storages"*. http://orientdb.com/docs/2.2.x/Paginated-Local-Storage.html. [Online; accessed 13-January-2017].

[97] OrientDB. *"OrientDB documentation - Teleporter"*. http://orientdb.com/docs/2.2.x/Teleporter-Home.html. [Online; accessed 23-March-2017].

[98] OrientDB. *"OrientDB documentation - Transactions"*. http://orientdb.com/docs/2.2/Transactions.html. [Online; accessed 20-February-2017].

[99] OrientDB. *"OrientDB Enterprise"*. http://orientdb.com/orientdb-enterprise/. [Online; accessed 5-February-2017].

[100] Philip Rathle, K. R. *"Webinar: Large Scale Graph Processing with IBM Power Systems & Neo4j"*. https://www.slideshare.net/neo4j/webinar-large-scale-graph-processing-with-ibm-power-systems-neo4j. [Online; accessed 1-March-2017].

[101] Planetmath.org. *"Hypergraph"*. http://planetmath.org/hypergraph. [Online; accessed 15-December-2016].

[102] PostgreSQL. *"PostgreSQL discussion thread - Monitoring query plan cache"*. https://www.postgresql.org/message-id/549564E4.4060800%40aule.net. [Online; accessed 11-March-2017].

[103] PostgreSQL. *"PostgreSQL documentation - Conformance"*. https://www.postgresql.org/docs/9.6/static/features.html. [Online; accessed 21-February-2017].

[104] PostgreSQL. *"PostgreSQL documentation - Planner/Optimizer"*. https://www.postgresql.org/docs/9.6/static/planner-optimizer.html. [Online; accessed 26-February-2017].

[105] PostgreSQL. *"PostgreSQL documentation - PL/pgSQL Under the Hood"*. https://www.postgresql.org/docs/current/static/plpgsql-implementation.html#PLPGSQL-PLAN-CACHING. [Online; accessed 6-March-2017].

[106] PostgreSQL. *"PostgreSQL documentation - PREPARE"*. https://www.postgresql.org/docs/current/static/sql-prepare.html. [Online; accessed 6-March-2017].

[107] PostgreSQL. *"PostgreSQL documentation - Resource Consumption"*. https://www.postgresql.org/docs/current/static/runtime-config-resource.html. [Online; accessed 22-February-2017].

[108] PostgreSQL. *"PostgreSQL documentation - What is"*. https://www.postgresql.org/docs/9.6/static/intro-whatis.html. [Online; accessed 15-February-2017].

[109] PostgreSQL. *"PostgreSQL documentation - WITH Queries"*. https://www.postgresql.org/docs/9.6/static/queries-with.html. [Online; accessed 21-February-2017].

[110] PostgreSQL. *"PostgreSQL Query Cache released"*. https://www.postgresql.org/about/news/1296/. [Online; accessed 6-March-2017].

[111] PostgreSQL. *"PostgreSQL Wiki - Tuning your PostgreSQL server"*. https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server. [Online; accessed 22-February-2017].

[112] Programmazione.it. *"Guida all'utilizzo di OrientDB"*. http://www.programmazione.it/index.php?entity=eitem&idItem=46036. [Offline; last access 12-December-2016].

[113] Raghavendra. *"Caching in PostgreSQL"*. http://raghavt.blogspot.it/2012/04/caching-in-postgresql.html. [Online; accessed 29-March-2017].

[114] Rodriguez, M. A. *"Graph Databases: Trends in the Web of Data"*. http://www.slideshare.net/slidarko/graph-databases-trends-in-the-web-of-data/. [Online; accessed 10-January-2017].

[115] Sakr, S. *"Processing large-scale graph data: A guide to current technology"*. https://www.ibm.com/developerworks/library/os-giraph/. [Online; accessed 3-March-2017].

[116] Salim Jouili, V. V. *"An empirical comparison of graph databases"*. http://ieeexplore.ieee.org/document/6693403/. [Online; accessed 15-January-2017].

[117] Sasaki, B. M. *"Graph Databases for Beginners: Other Graph Data Technologies"*. https://neo4j.com/blog/other-graph-database-technologies/, 2015. [Online; accessed 2-November-2016].

[118] Smith, G. *"Inside the PostgreSQL Shared Buffer Cache"*. https://2ndquadrant.com/media/pdfs/talks/InsideBufferCache.pdf. [Online; accessed 22-February-2017].

[119] Sotiris Beis, Manos Schinas, S. P. A. P. L. G. e. a. *"Performance benchmark between popular graph databases."*. https://github.com/socialsensor/graphdb-benchmarks. [Online; accessed 15-January-2017].

[120] Spark, A. *"GraphX Programming Guide"*. http://spark.apache.org/docs/latest/graphx-programming-guide.html#pregel-api. [Online; accessed 3-March-2017].

[121] Tesoriero, C. *"Getting started with OrientDB"*. Packt Publishing, 2013.

[122] Thomas H. Cormen, Charles E. Leiserson, R. L. R., and Stein, C. *"Introduction to Algorithms"*, 3rd ed. The MIT Press, 2009.

[123] Tim Baker, Tharindi Hapuarachchi, B. B. *"The future is graph shaped"*. https://blogs.thomsonreuters.com/answerson/future-graph-shaped/. [Online; accessed 1-March-2017].

[124] Tim Berners-Lee, J. H., and Lassila, O. *"The Semantic Web"*. Scientific American, 2001.

[125] TinkerPop. *"Blueprints"*. https://github.com/tinkerpop/blueprints/wiki. [Online; accessed 2-March-2017].

[126] TinkerPop. *"Gremlin"*. http://tinkerpop.apache.org/gremlin.html. [Online; accessed 2-March-2017].

[127] TinkerPop, A. *"Apache TinkerPop website"*. http://tinkerpop.apache.org/. [Online; accessed 14-December-2016].

[128] TinkerPop, A. *"Neo4j-Gremlin"*. http://tinkerpop.apache.org/docs/current/reference/#neo4j-gremlin. [Online; accessed 6-March-2017].

[129] TinkerPop, A. *"TinkerPop documentation"*. http://tinkerpop.apache.org/docs/current/reference/. [Online; accessed 2-March-2017].

[130] TinkerPop, A. *"TinkerPop Recipes"*. http://tinkerpop.apache.org/docs/current/recipes/. [Online; accessed 18-March-2017].

[131] Tom Heath, C. B. *"Linked Data - Evolving the Web into a Global Data Space"*. Morgan & Claypool Publishers, 2013.

[132] Toyotaro Suzumura, M. D. *"XGDBench: A Benchmarking Platform for Graph Stores in Exascale Clouds"*. https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnx0b2t5b3RlY2hzdXp1bXVyYWxhYmVuZ3xneDoyMGRiOGFlM2Y2OGY5Mzhj. [Online; accessed 15-January-2017].

[133] Trylks. *"PostgreSQL temporary table cache in memory?"*. http://stackoverflow.com/questions/14162917/postgresql-temporary-table-cache-in-memory. [Online; accessed 29-March-2017].

[134] Virtuoso. *"Virtuoso FAQ"*. https://virtuoso.openlinksw.com/virt_faq/. [Online; accessed 27-February-2017].

[135] Virtuoso. *"Virtuoso web page"*. https://virtuoso.openlinksw.com/. [Online; accessed 27-February-2017].

[136] W3C. *"Data"*. https://www.w3.org/standards/semanticweb/data. [Online; accessed 28-January-2017].

[137] W3C. *"Inference"*. https://www.w3.org/standards/semanticweb/inference. [Online; accessed 28-January-2017].

[138] W3C. *"LargeTripleStores"*. https://www.w3.org/wiki/LargeTripleStores. [Online; accessed 16-January-2017].

[139] W3C. *"Ontologies"*. https://www.w3.org/standards/semanticweb/ontology.html. [Online; accessed 28-January-2017].

[140] W3C. *"OWL Web Ontology Language"*. https://www.w3.org/TR/owl-features/. [Online; accessed 27-March-2017].

[141] W3C. *"Primer"*. https://www.w3.org/TR/rdf11-primer/. [Online; accessed 2-February-2017].

[142] W3C. *"RDF Store Benchmarking"*. https://www.w3.org/wiki/RdfStoreBenchmarking. [Online; accessed 16-January-2017].

[143] W3C. *"SPARQL Query Language for RDF"*. https://www.w3.org/TR/rdf-sparql-query/. [Online; accessed 15-December-2016].

[144] W3C. *"RDF"*. https://www.w3.org/RDF/, 2014. [Online; accessed 15-December-2016].

[145] W3C. *"RDF concepts"*. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/#data-model, 2014. [Online; accessed 15-December-2016].

[146] W3C. *"RDF syntax"*. https://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/, 2014. [Online; accessed 15-December-2016].

[147] Weinberger, C. *"Benchmark: PostgreSQL, MongoDB, Neo4j, OrientDB and ArangoDB"*. https://www.arangodb.com/2015/10/benchmark-postgresql-mongodb-arangodb/. [Online; accessed 28-February-2017].

[148] Weinberger, C. *"Index Free Adjacency or Hybrid Indexes for Graph Databases"*. https://www.arangodb.com/2016/04/index-free-adjacency-hybrid-indexes-graph-databases/. [Online; accessed 09-January-2017].

[149] Weinberger, C. *"Native multi-model can compete with pure document and graph databases"*. https://www.arangodb.com/2015/06/multi-model-benchmark/. [Online; accessed 10-March-2017].

[150] Weinberger, C. *"Performance comparison between ArangoDB, MongoDB, Neo4j and OrientDB"*. https://www.arangodb.com/2015/06/performance-comparison-between-arangodb-mongodb-neo4j-and-orientdb/. [Online; accessed 28-February-2017].

[151] Wiese, L. *"Advanced Data Management: For SQL, NoSQL, Cloud and Distributed Databases."*. Walter de Gruyter GmbH & Co KG., 2015.

[152] Wikipedia. *"Apache Spark"*. https://en.wikipedia.org/wiki/Apache_Spark. [Online; accessed 3-March-2017].

[153] Wikipedia. *"Breadth-first search"*. https://en.wikipedia.org/wiki/Breadth-first_search. [Online; accessed 1-March-2017].

[154] Wikipedia. *"Depth-first search"*. https://en.wikipedia.org/wiki/Depth-first_search. [Online; accessed 1-March-2017].

[155] Wikipedia. *"Graph partition"*. https://en.wikipedia.org/wiki/Graph_partition. [Online; accessed 23-February-2017].

[156] Wikipedia. *"Hypergraph"*. https://en.wikipedia.org/wiki/Hypergraph. [Online; accessed 15-December-2016].

[157] Wikipedia. *"Triplestore"*. https://en.wikipedia.org/wiki/Triplestore. [Online; accessed 16-January-2017].

[158] Wikipedia. *"Graph database"*. https://en.wikipedia.org/wiki/Graph_database, 2016. [Online; accessed 2-November-2016].

[159] Wikipedia. *"SPARQL"*. https://en.wikipedia.org/wiki/SPARQL, 2016. [Online; accessed 2-November-2016].

[160] Wolpe, T. *"DataStax snaps up Aurelius and its Titan team to build new graph database"*. http://www.zdnet.com/article/datastax-snaps-up-aurelius-and-its-titan-team-to-build-new-graph-database/. [Online; accessed 27-February-2017].

[161] Yalamarthi, V. *"Regarding Multiple collections for ArangoDB Graph Queries"*. https://groups.google.com/forum/#!topic/arangodb/_sHmi6ifMJM. [Online; accessed 7-March-2017].