University of West Bohemia

Faculty of applied sciences

Department of computer science and Engineering

DIPLOMA THESIS

NOSQL DATABASES IN EEG/ERP DOMAIN

Pilsen, 2013

Ladislav Janák

Acknowledgments

First of all, I would like to thank to my family and friends for the support during the study. Furthermore, I would like to thank the tutor of my diploma thesis Václav Papež, and his colleague Roman Mouček. Thanks to their valuable advices and comments it was possible to realize this thesis and brought it into the final form.

Statement

I hereby declare that this master thesis is completely my own work and that I used only the cited sources.

Pilsen, ……………………….

…………………………………………….
Ladislav Janák

Abstract

NoSQL databases in EEG/ERP domain

This thesis deals with current knowledge in NoSQL databases. The main goal was to choose a suitable NoSQL database model and a related NoSQL database system. NoSQL database system could replace the current relational database system used in the EEG/ERP portal application. This request of replacement within the meaning of improvement current EEG/ERP portal's database model has been made by EEG/ERP research group at the University of West Bohemia. This thesis addresses the issue of the suitability of the chosen NoSQL database system for EEG/ERP domain. This suitability is evaluated on the base of working convenience and performance testing of the new database system in comparison with existing solution.

The theoretical part of this work contains general overview and concepts of database models that are currently available. Then relational and NoSQL models are compared. It is followed by the description of the current EEG/ERP portal database layer and the more detailed description of the chosen NoSQL database system. In the practical part the development of the NoSQL database model from the EEG/ERP portal model is described. Then the selection of the part of the EEG/ERP portal model and the development of the corresponding NoSQL database model for testing purposes are presented. Performance analysis of both models using the same database queries and commands over each database model is discussed in the next part. The last part of this work evaluates the suitability of the chosen NoSQL model for the EEG/ERP domain.

CONTENT

1 INTRODUCTION

Database layer of any software application (web, standalone, embedded, etc.) is one of the most important parts of application which deals with efficient data management – important roles play especially performance aspects, good variability of data model, more kinds of supported Application Program Interfaces (*APIs*) and standards, good support for query languages or good scaling options. The importance of database systems has been rapidly increased, especially in recent years when thousands of terabytes of data (social data, business data, etc.) are processed per second. Outside the standard usage there are two relatively new areas of database using – social networking and data mining.

Standard relational databases with their tabular model are used for well-known use cases like banking, education, finance, sales, etc. However, they are not so suitable for more general models, which are used e.g. in social networking, where model forms a network where every node of the network is connected with any other node and the nodes are connected with edges. Model for this network in relational database is often very difficult. Moreover, this model has small capabilities of semantic expression of relationships among records. Now, new generation of database systems is coming – Not only SQL (*NoSQL*). These database systems are very suitable for applications like social networking or data analysis. More types of NoSQL databases exist, which are different in their database model and their query language. It is important to realize that NoSQL is not a replacement for relational databases. Both are suitable for specific application.

This work was created on the base of requirements of an EEG/ERP portal group. Currently, the EEG/ERP portal application uses Oracle relational database as database layer. The main requirement was to explore non – relational database solutions, which may be suitable for replacement or improvement of current database layer and thus to increase the performance and options variability. As the best solution for this purpose I chose the OrientDB database, which is NoSQL document/graph database. I chose OrientDB on the base of previous study. I imported the part of EEG/ERP portal database model into OrientDB database model. In this work I will present the analysis of results of database queries testing and commands on OrientDB NoSQL database system against Oracle relational database system. These tests are performed on the same sub-model of EEG/ERP portal application. Finally, I bring my own point of view on OrientDB solution and I bring the summary of suitability of this solution for the EEG/ERP domain.

2 THEORETICAL PART

2.1 DATABASE MODELS

2.1.1 GENERAL CHARACTERISTIC

Information for this chapter was taken from [1].

Database model allows making abstractions of real-world events or conditions and enable us to store characteristics of entities and relationships between them. It allows collecting data into logical construct which represents the data structure and the data relationships.

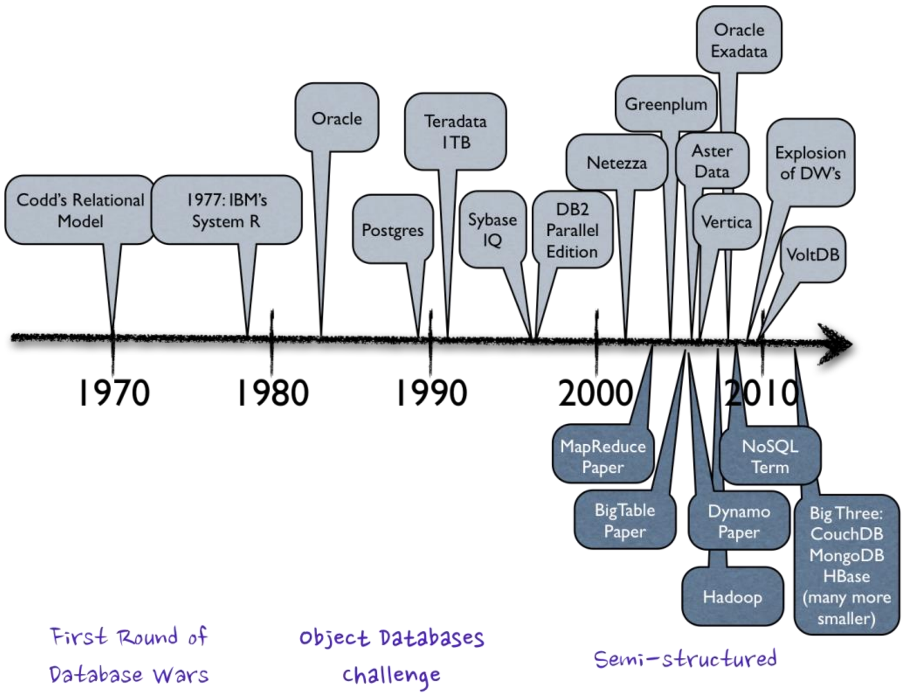


Figure 2-1 *History of database evolution [2]*

Database model may be grouped into two categories:

- **Conceptual model**: focuses on the logical nature of data (what is represented in the database). Conceptual models in data modelling are:
  - Entity relationships (*E-R*) model
  - Domain model

E-R models use three types of relationships to describe associations among data; relationships can be seen on Figure 1-2.

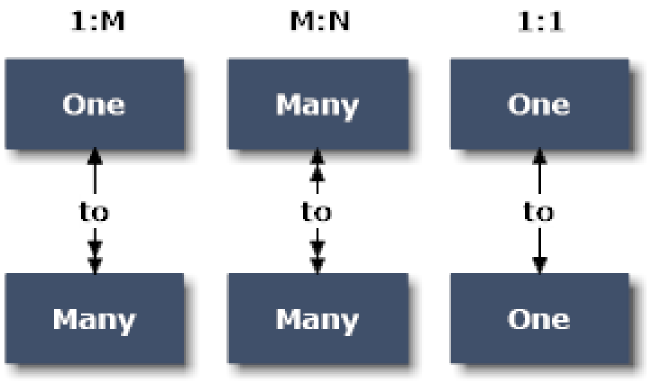


Figure 2-2 *Types of entity relationships in conceptual E-R model*

- **Implementation model**: it explains how information is represented in the database or how the data structures are implemented. Implementation models are:
  - Hierarchical database model (this database model is no longer used)
  - Network database model (this database model is no longer used)
  - Relational database model

There also are other database systems – NoSQL. The well-known NoSQL database models are:

- Document database model
- Column database model
- Key-Value database model
- Graph database model

2.1.2 RELATIONAL DATABASE MODEL – BASIC CONCEPTS

Information for this chapter was taken from [1] [3].

Basically the relational model is a collection of tables in which data are stored. Each of table can be understood like matrix consisting of row/column intersections. Tables are related to each other by sharing entity characteristic. Each table is completely independent of another one, but is easy to connect the data between tables. The relation mathematically is any set R:

$$R \subset A \times A$$

2.1.2.1 LOGICAL VIEW OF RELATIONAL MODEL

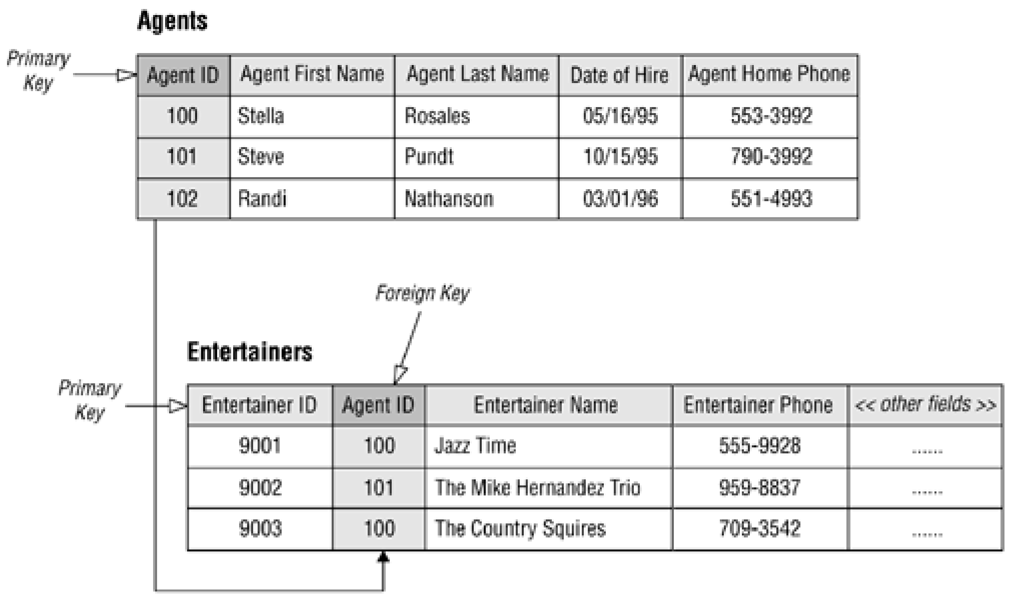
The basic data components are entities and their attributes. These components are stored into logical construct – table. Entities abstract entities from real world e.g. – person, address, room. Each entity has own attributes like – name, address number, room number. Data in relational model must be homogeneous. It means every row has the same format (fixed table schema). Table model with its entities, relationships and attributes is also called E-R-A model.

2.1.2.2 RELATIONS BETWEEN ENTITIES

Relations between tables are realized through keys. Key is an attribute which is shared among tables. The link is created by two tables which share an attribute. There are two basic types of keys:

- Primary key
- Foreign key

The primary key of one table appears again as the foreign key in another table. The foreign key contains value that matches the other table's primary key. This relationship between primary and foreign key can be seen on Figure 2-3.



Agents

Agent ID	Agent First Name	Agent Last Name	Date of Hire	Agent Home Phone
100	Stella	Rosales	05/16/95	553-3992
101	Steve	Pundt	10/15/95	790-3992
102	Randi	Nathanson	03/01/96	551-4993

Entertainers

Entertainer ID	Agent ID	Entertainer Name	Entertainer Phone	<< other fields >>
9001	100	Jazz Time	555-9928	......
9002	101	The Mike Hernandez Trio	959-8837	......
9003	100	The Country Squires	709-3542	......

Figure 2-3 *Example of database table model with primary and foreign key fields [4]*

2.1.2.3 DATA INTEGRITY

It is necessary to avoid data corruption, for this purpose integrity model serves. The three main integrity rules are:

- **Referential integrity**: foreign key must contain values that match the other table's primary key value, or it must contain a *null* value
- **Entity integrity:** no *null* values for a primary key columns and guarantee that each entity will have a unique identity
- **Domain integrity**: it ensures the validity of values for a given column (this is ensured by defining the type, constrains and rules of the column)

2.1.2.4 SQL QUERY LANGUAGE

Structured Query Language (*SQL*) is a standardised query language for relational databases. SQL lets us work with data in relational databases. SQL allows to process sets of data as groups or as individual units. It provides operations like querying data, inserting, updating columns and deleting rows in table. The next provided operations are: creating, replacing, altering, and dropping objects, controlling access to the database and its object. SQL provides resources for ensuring database integrity and consistency.

Among the well-known relational database systems (RDBMS) using relational model belong e.g. Oracle Database Xg, MySQL, MS SQL Server, Firebird or PostgreSQL. Relational model was used very successfully throughout the 80s and 90s and it's still working today. Nowadays, NoSQL databases are increasingly perceived.

2.1.2.5 PROBLEM WITH JOINS

Relational database model retrieves relationships by using foreign key that points to the primary key.

One-to-many relationship – typical query:

```
SELECT TITLE FROM PERSON p
INNER JOIN EDUCATION_LEVEL e ON
p.EDUCATION_LEVEL_ID = e.EDUCATION_LEVEL_ID WHERE p.SURNAME = 'Walker'
```

This query is a *JOIN* operation. JOIN (creation of virtual tables) is executed at run-time. The same problem occurs at one-to-many and at many-to-many relation too. The problem is that

Relational Database Management System (*RDBMS*) does not have concept of *collections*. So, for more complex relations like many-to-many we need a intersect table with keys in all possible combinations – this leads to double JOIN per record.

This approach is very expensive. Indexes speed up the searching but slow down operations like *INSERT*, *UPDATE* and *DELETE*. Moreover, indexes need additional space on disk. If we have e.g. 5 tables with thousands of records - millions of JOINs can be. The solution how to avoid JOIN operations are NoSQL databases.

2.1.3 NOSQL GENERALLY INFORMATION

Information for this chapter was taken from [5] [6] [7] [8].

NoSQL (Not-Only SQL) is not relational database and it is not a replacement for a RDBMS. Instead of tables with columns and rows which we can find in a traditional RDBMS – NoSQL databases have not fixed schema. NoSQL databases have not relationships made by keys too. NoSQL databases can handle hierarchical or unstructured nested data. To handle these types of data in RDBMS, we would need multiple relational tables with all kinds of primary and foreign keys. NoSQL can easily take advantage of horizontally scaling unlike RDBMS (RDBMS require for next scaling faster hardware). They do not use SQL because Structure Query Language was designed for use with relational databases and NoSQL is much closer to object-oriented approach.

The original intention of NoSQL approach has been creation of modern web-scale databases. Primarily, NoSQL is designed for distributed data stores with needs of scaling of the data (e.g. Facebook or Twitter, which accumulates terabits of data every single day). To basic characteristic belong:

- schema-free
- easy replication support
- own API
- consistency (BASE/ *ACID* (Atomicity, Consistency, Isolation, Durability) transactions)
- huge amount of data
- unstructured data
- data on multiple servers in the cloud

Massive scalability, low latency, the ability to grow the capacity of database on demand and an easy programming model is necessary for top-tier web-sites.

Many different NoSQL database systems (NoSQL DBMS) are currently available. These DBMS are different in their way to store data – database model. Next differences are in their programming language (API), query language, transaction management and so on.

Changing needs on storing data are closely related with the arrival of NoSQL databases. Nowadays, it is easier to capture data and access them through third parties such as Facebook, Twitter and others. Personal information, geo data, social graphs, user-generated content, sensor-generated data are a few examples of the ever-expanding array of data being captured. And the usage of the data is rapidly changing the nature of communication, shopping, advertising, entertainment, and relationship management. Developers want very flexible databases that easily accommodate new data types which are not disrupted by content structure changes. Much of the new data is unstructured and semi-structured, so developers also need a database which is capable to efficiently store these types of data. The rigidly defined, schema-based approach used by relational databases makes it difficult to quickly incorporate new data types and it has a poor fit for unstructured and semi-structured data. NoSQL provide data model that can better satisfy these needs.

Semi-structured and unstructured data are generated by applications which have millions users per day. Database must be able to growing up with these needs.

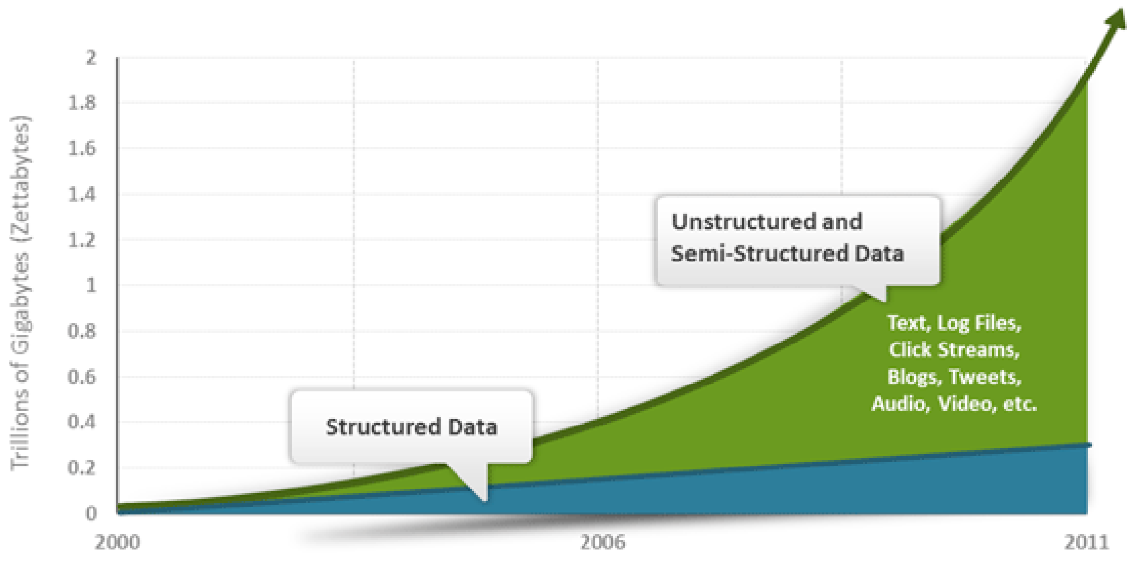


Figure 2-4 *Time trend of changing evolution in terms of needs on data structure and data size [8]*

2.1.4 BASIC DESCRIPTION OF CHOSEN NOSQL DATABASE MODELS

Information for this chapter was taken from [9] [10] [11] [12] [13] [14] [15] [16] [17].

This chapter gives an introduction into chosen NoSQL database models.

2.1.4.1 DOCUMENT-ORIENTED MODEL

A document database is, at its core, a key/value store – the difference is that each record has multiple fields in a document data store (see Figure 2-6) but one key is the only way to access a record in a key/value store (see Figure 2-7). Document is represented by key. Each document has a unique key (often simple string value). Document database requires data which are stored in understandable format. The format can be Extensible Markup Language (*XML*), Javascript Object Notation (*JSON*), *Binary JSON* or anything else the database can understand.

- **Document**: it is the fundamental unit of storage; equivalent to a row in a relational database.
- **Collection**: it is a set of related documents and plays a role similar to table in relational database. Collection may be also likened to a directory in a file system.
- **Associations:** association between documents (collections) are stored as a single document, associations are direct links between documents -no JOINs (see chapter 2.1.2.5) and they can be found by key or by index.

The main difference from RDBMS systems is much more flexible database model called *schema-less* model in document-based style, instead of defining a strict schema. You can see on the Figure 2-6 that the records (documents) of the same entity have not the same set of fields and unused fields might be kept empty. Document is stored by serializing an object (e.g. a Java-based instance of an object class), previously mentioned, by using a recognized data standard such as JSON.

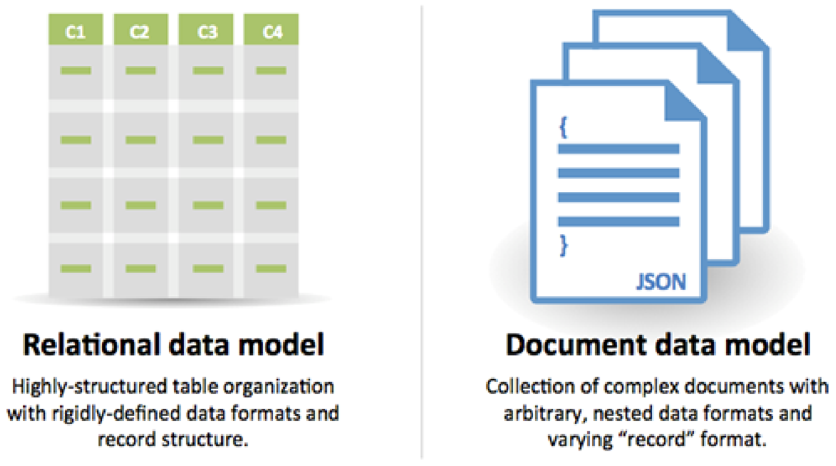

C1
C2
C3
C4
Relational data model
Highly-structured table organization with rigidly-defined data formats and record structure.
{
}
JSON
Document data model
Collection of complex documents with arbitrary, nested data formats and varying "record" format.

Figure 2-5 *Comparison of relational and document data model [18]*

The primary benefit of the document model approach is that the structure does not have to be predefined. The structure of the document allows construction of a document which contains a large amount of information. Moreover, the structure can be change on the fly. This approach gives us a lot of flexibility for storing of composite records and information that would be difficult within RDBMS.

```
{
    "country" : "UK",
    "transaction_country" : "UK",
    "amount" : 23.23,
    "customerid" : "3458983734981294",
    "merchant_country" : "UK",
    "type" : "inperson",
    "merchantid" : "49587",
    "datetime" : "2012-08-23T13:54:00"
}
```

```
{
    "country" : "UK",
    "merchant_country" : "UK",
    "datetime" : "2012-08-23T13:54:00",
    "amount" : 23.23,
    "transaction_country" : "UK",
    "customerid" : "3458983734981294",
    "type" : "inperson",
    "merchantid" : "49587",
    "items" : [
       {
          "amount" : 10.2,
          "description" : "food"
       },
       {
          "amount" : 13.03,
          "descrition" : "DVD"
       }
    ]
}
```

Figure 2-6 *Credit card transaction sample with different information in JSON format [9]*

Important subclasses of document databases are *XML databases,* which can process XML files. They map XML data (elements, attributes, etc.) to instances of their model. There are two types of XML databases:

- **XML-native**: use the XML data model directly – they are designed to hold arbitrary XML documents and XML schemas (elements, attributes, text, etc.). They can store complete documents and can store any document, regardless of schema.
- **XML-enabled**: are useful when publishing existing data as XML or importing data from XML document into an existing database. They are not good way to store complete XML documents. They store data and hierarchy but discard everything else: document identity, sibling order etc.

Document store databases can be especially used when following characteristics are desirable:

- wide variety of access pattern and data types
- to build *CRUD* (Create, Read, Update, Delete) based applications
- easier upgrade path
- programmatically friendly data types (JSON, HTTP, etc.)

Main representatives of document store databases are *CouchDB*, *MongoDB*, *BaseX* or *Redis*.

2.1.4.2 KEY/VALUE MODEL

Key-Value databases use the same pattern, which is used e.g. in accessing memory locations - the memory location's address serving as the key and the value is stored at that memory address. This type of database is especially designed for storing unstructured big amount of data (in these cases are much faster than relational database). These databases use schema-less model too – if we have not some data we do not store fields for this data (see Figure 2-7 where field *access* is empty with *key* 314). Data can be stored in data type of the used programming language. Keys and bins are created to store this kind of data.

- **Keys**: key is created for each record. Arbitrary fields are available as bins. Key can be e.g. string value.

- **Bins**: they could be equated to the columns in relational databases. Each bin consists of a name and a value. Bin can be created for each piece of data.

Each record has a primary key and a collection of bins (values). All data for single record are stored together (similar to rows in relational database).

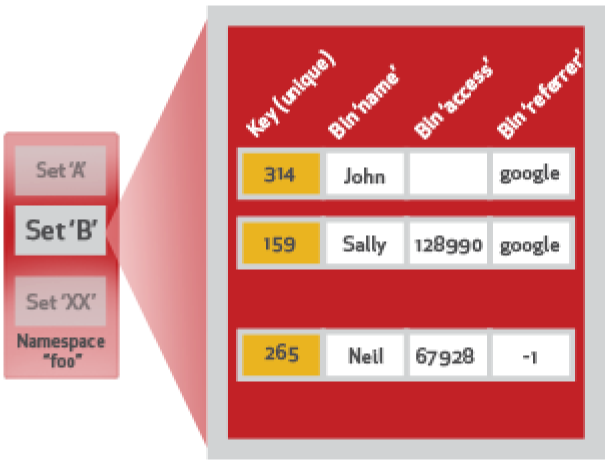


Figure 2-7 *Example of unstructured data stored in bins in one dataset [19]*

Key/Value store databases can be especially used when following characteristics are desirable:

- small continuous read and writes (fast in-memory access)
- programmatically friendly data types
- easier upgrade path
- to store cache or Binary Large Object (*BLOB*) data

Main representatives of Key-Value databases are *Membase*, *Riak* or *Oracle NoSQL* database.

2.1.4.3 GRAPH MODEL

Graph database model can be characterized as those where data structures for the schema and instances are modelled as graphs or generalizations of them. Data manipulation is expressed by graph-oriented operations. Graph model is the most generic data structure, which is capable to represent in a highly accessible way any kind of data. Typical graph is composed from *nodes* and *edges.* Mathematically graph is an ordered pair of nodes and edges:

$$G = (V, E)$$

Most implementations of graph database implements so called property graph, where nodes and edges can have properties.

- **Nodes** represent records that have named values (properties) corresponding to columns or attributes in relational world. The simplest graph consists of one node. Node can have millions of properties, but it is much better to distribute data into multiple nodes, organized with explicit relationships. Properties of node are organised as a collections. Each node has a unique identifier and a set of outgoing and incoming edges. Nodes represents entities of real world (similar to row in relational table)
- **Edges** represent relationships between graph nodes. Edges themselves are records as well. They are labelled for expression of relation between two nodes and can have arbitrary number of properties (it depends on specific implementation). Each edge has a unique identifier. Eventually, edge can have an outgoing tail vertex and incoming tail vertex.

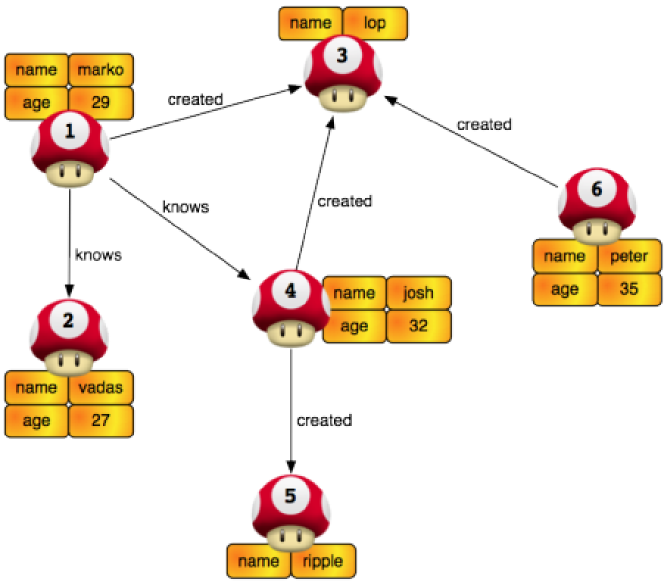


Figure 2-8 *Property graph model [20]*

The biggest advantage of the graph model is its flexibility (unlike RDBMS). They are usually schema-less and they allow a dynamic set of properties (changes on the fly). Storage is optimized to traverse graph and it is optimized for queries which use benefit of the leveraging data proximity - starting from one or several root nodes, rather than global queries. Nodes can be arbitrary linked to other nodes through arbitrary edges. Thanks to all these features, graph databases have great ability to express semantics between records.

The special type of graph databases are Resource Description Framework (*RDF*) *storages* that handle with RDF triples natively. They are designed for building semantic web applications. They can store data and metadata as triples.Graph databases can be especially used when following characteristics are desirable:

- to develop application related with social networking
- to dynamically build relationships between objects that have dynamic properties
- to build database incrementally through programming
- to avoid very nested JOIN operations (thanks to fast navigation between graph entities)

Main representatives of graph databases are *Neo4J*, *AllegroGraph* or *InfiniteGraph*.

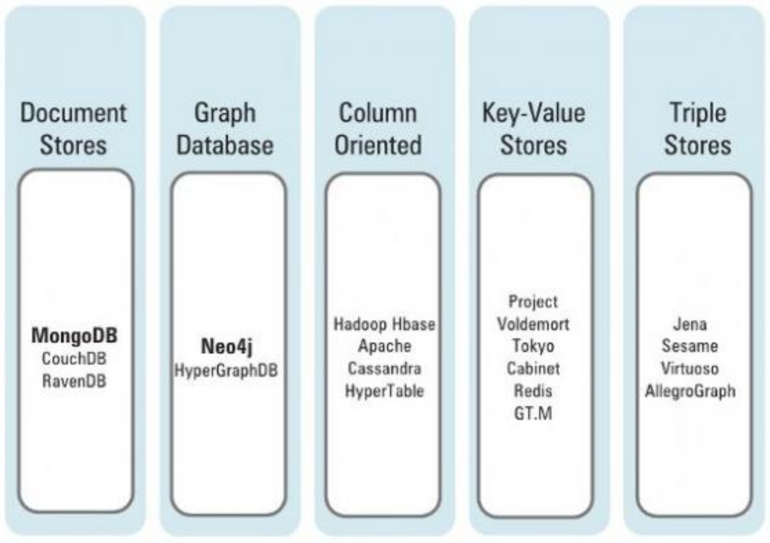


Figure 2-9 *Popular NoSQL databases examples [21]*

2.2 COMPARISON OF RELATIONAL AND NOSQL APPROACH

In this chapter I will describe some advantages and disadvantages of both approaches that have impact on the application design and performance. The right choice of database model for specific use case is very important and also difficult task. We can see comparison between relational and NoSQL databases according to the scaling size and database model complexity on Figure 2-10.

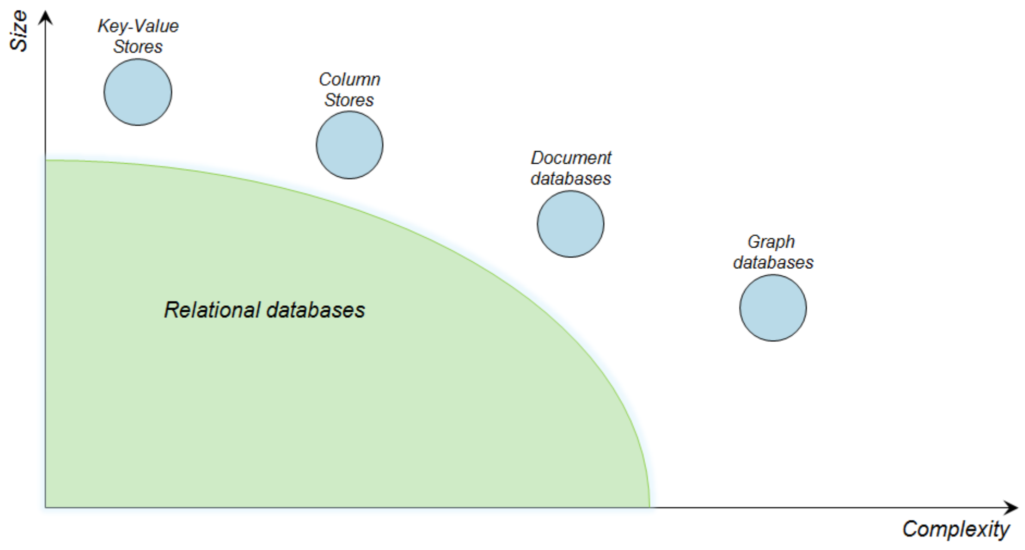


Figure 2-10 *Positions of NoSQL databases (scaling vs. complexity) [16]*

2.2.1 WHEN TO CHOOSE AND NOT TO CHOOSE AN RDBMS

Information for this chapter was taken from [22].

One of the key aspects of RDBMS is its logical model when there is fixed schema in table form, each column could have values with predefined restrictions. Next key aspect is integrity model. Referential integrity ensures logical consistency of the domain model and cross entity consistency. The aspect of consistency is ACID transactions. It ensures that either all changes are consistent in every moment – changes are committed or not committed at all. The next key feature is the ability to execute arbitrary queries within SQL selects.

Pros of RDBMS are:

- suitability for structured data with the ability to ask different questions all the time
- native referential integrity and ACID transactions
- well-known relational model which uses well-known query language (SQL)

Cons of RDBMS are:

- unsuitability for storing application entities in a persistent and consistent way
- unsuitability for hierarchical application objects with query capability into them
- unsuitability for storing large trees or networks
- unsuitability for running in the Cloud and usage as a distributed database
- unsuitability for very fast growing data which is not possible to process on a single machine
- not easy accessible horizontal scaling (without buying more expensive hardware)
- performing JOIN operations

2.2.2 WHEN TO CHOOSE AND NOT TO CHOOSE NOSQL

Information for this chapter was taken from [5] [22] [23].

How it was mentioned before, NoSQL databases were designed for distributed data stores for very large scale of data. Thanks to better options for horizontal scaling, NoSQL databases means an inexpensive solution for large datasets. Another big advantage of NoSQL is its flexible data model (schema-less model). Moreover, we can make changes in model at runtime. There are no restrictions on data unlike RDBMS where every minor change must be carefully managed. The benefit of this approach is that application changes and database schema changes do not have to be managed as one complicated change unit.

Nowadays, the amount of stored data rapidly grows and NoSQL model can handle large datasets very well. Data continues to become more connected (social networks, blogs, etc.) every major system is built to be interconnected. Next challenge of NoSQL databases is that they can easily handle nested data structures. To accomplish the same thing in SQL, we would need multiple relational tables with all kinds of keys – this has the influence on performance, which can degrade in RDBMS as we store massive amount of data required in social networking and semantic web. The last significant advantage of NoSQL approach is the absence of expensive JOINs. In NoSQL databases there are direct links among records (unlike RDBMS - if we make query over multiple tables in RDBMS then JOIN operations among table are used).

The thing which can be a little confused for new users of NoSQL databases is that each NoSQL database has own set of APIs, libraries and query languages. RDBMS have long tradition and SQL knows almost everyone who works with databases. RDBMS are stable, in comparison many NoSQL alternatives are not fully stable versions and their key features are not yet implemented (triggers, ACID transactions). Earlier was the problem that NoSQL did not support ACID transactions, but nowadays these differences are rapidly erased and many of NoSQL databases support ACID natively. Next disadvantage for some users is that even a simple query requires significant programming expertise. NoSQL requires a lot of skills to install and set-up.

Pros of NoSQL are:

- flexible data model without restrictions on data
- suitability for running in the *Cloud*
- good options for horizontal scaling without buying additional expensive hardware
- suitability for storing of rapidly growing data
- suitability for hierarchical, heavily interconnected or unstructured data
- suitability for creation semantic model (*semantic web*)

Cons of NoSQL are:

- unsuitability for users with small programming skills → difficulty to manage database and make database queries
- partial instability of open source projects (the most of NoSQL projects are open source) → on-going development process → some required features could be missing
- bigger difficulty to install and set-up than RDBMS

2.3 EEG/ERP PORTAL APPLICATION

2.3.1 GENERAL CHARACTERISTIC

EEG/ERP portal is a web application that is managed by members of EEG/ERP group at the University of West Bohemia. This group performs electrophysiological experiments in *EEG* (Electroencephalography) laboratory. These experiments are based on the brain activity measurement (EEG). EEG is non-invasive method and EEG/ERP group use it for the measurement of evoked potentials. Following example describes possible course of the experiment: Experimenter prepares subject person and measure required data, then stores information about experiment and measured data into portal application.

The main purpose of the portal is storing and management of EEG data. Other data types managed by portal are:

- articles which are related with EEG/ERP
- information about configuration of EEG/ERP experiments
- information about experimenters
- information about subject persons
- information about used hardware and software

Currently EEG/ERP portal is based on Java Enterprise Edition (*Java EE*) and uses Java Servlet Pages (*JSP*) for presentation layer, Spring Framework for application layer and Oracle Database 11g RDBMS with Hibernate Object Relational Mapping (*ORM*) as database layer.

2.3.2 DATABASE LAYER

The database layer ensures storing all kinds of data which I have previously mentioned and it uses Oracle database 11g Enterprise edition with Hibernate that ensures object relation mapping. This database uses standard relational database model based on tables.

Current database contains about 85 tables. Tables can be divided into three following groups:

1) tables related with experiments and configuration of experiments
2) tables related with research groups and their members
3) tables related with storing of data files, and XML files

Oracle database has not own Java or other API, therefore ORM must be used. ORM ensures binding Plain Old Java Objects (*POJOs*) with records. This approach makes the access to Oracle database easier, but there is still some additional overhead because of using ORM.

As a query language in EEG/ERP portal application Hibernate Query Language (*HQL*) is used. For functions, triggers and procedures are used Oracle *PL/SQL* (Procedural language/Structured Query Language). The weaknesses of current model (except disadvantages mentioned in chapter 2.2.1) are these: Oracle database has not got own Java API. It has the effect need for ORM for binding POJOs with records→ additional overhead in the form of ORM. Finally, current database model is not suitable for semantic web (semantic web principle is one of the possible solutions for EEG/ERP portal database model replacement).

The main goal of this thesis is to find out whether NoSQL solution could avoid above mentioned restrictions and thus improve EEG/ERP portal database layer.

2.4 ORIENTDB NOSQL DATABASE

As a suitable NoSQL database for the replacement of EEG/ERP portal relational model I chose OrientDB. This database was chosen on the grounds of results from study [24]. OrientDB, at its core, is a *document* database written in Java. OrientDB is free to use without limitation – Apache license. The records are documents but relationships between them can be managed like in a *graph* database models.

2.4.1 MAIN CONCEPTS

Information for this chapter was taken from [25] [26]

2.4.1.1 STORAGE TYPES

There are three options how to access the database:

- **Local:** OrientDB runs as *embedded*. Database is open via the local File System (without remote connection). Database cannot be opened by multiple processes. It is the fastest access – it avoids any network connection and transfers.

- **Remote:** The access is made by using network. Database is open via *remote* network connection. Database is shared among multiple clients. It requires OrientDB server running.
- **Memory**: All data remain in *memory* without file system usage

The speed of protocols from the fastest one is: Memory > Local > Remote. Storage is composed of multiple *Cluster* and *Data Segments.*

2.4.1.2 CLASSES, CLUSTERS, DATA SEGMENTS, RECORDID AND RELATIONSHIPS

Classes

The basic element of record is ***document*** (see 2.1.4.1). Document can belong to one ***class***. This approach of classes is well known from object-oriented programming (*OOP*), the same rules are applied in OrientDB. Classes can have properties but they are not mandatory - schema-less model, but class can be schema-full (mandatory properties) or mixed. New class is by default new physical ***cluster*** and cluster has the same name as the class

Clusters

Groups of records are stored in a cluster – equivalent in relational world may be *table.* The main difference is that cluster can record heterogeneous records, e.g. *Customers* and *Providers* all together. One class can be partitioned in multiple clusters – we can store records physically in multiple places, see Figure 2-11.

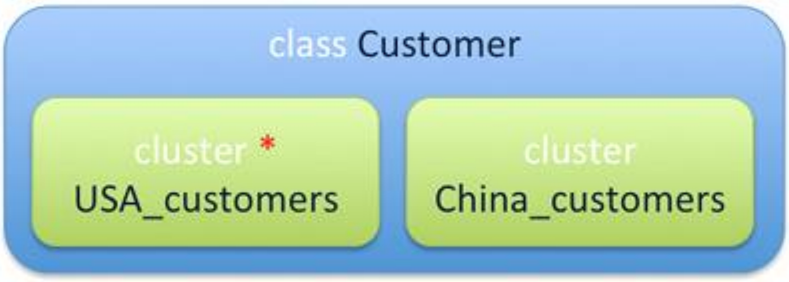


Figure 2-11 *Class Customer partitioned on two clusters (red star marks the default one) [25]*

- The default cluster is cluster USA_customers, it is used by default when the generic class Customer is used: *insert into Customer*
- When we query the Customer class, all clusters are scanned: *select * from Customer*
- If we know the type of Customer, we can query directly the target cluster avoiding to scan all the others: *select * from cluster:China_customers*

Thanks the option to use different physical places to store records we can make faster queries against clusters because we query a sub-set of all class's clusters. We can achieve better partitioning – it reduces usage of indexes. We can make parallel queries on multiple discs.

There are two types of clusters**:**

- **Physical cluster** (default): It is persistent and it is written directly to the file system.
- **Memory cluster:** It is in-memory storage → all data is temporarily in memory.

Data segments

OrientDB uses *data segments* for storing record content. It uses two or more files with extension "oda" (OrientDB Data) and one file with the extension "odh" (OrientDB Data Holes).

RecordID

Each record in OrientDB has its own unique ID - *RecordID*, ID is self-assigned. RecordID has two parts:

```
#<cluster-id>:<cluster-position>
```

- **Cluster-id:** It is the ID of the cluster.
- **Cluster-position:** It is the position of the record inside the cluster.

It means that loading by a recordID has the response time close to O(1).

Relationships (document model)

There are two kinds of relationships:

- **Referenced:** Relationships are natively managed without computing JOINs. OrientDB stores direct links between objects.

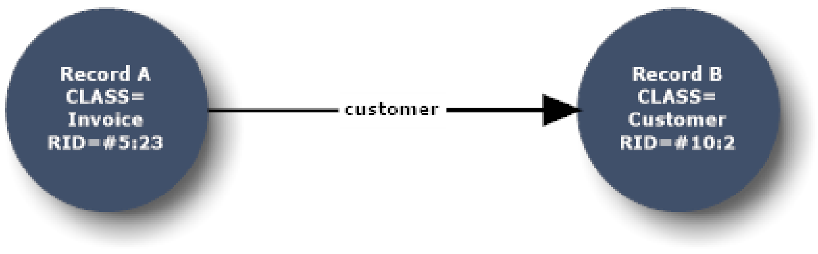


Figure 2-12 *Referenced relationship*

Record A will contain the reference to Record B in the property customer. Both records are reachable by each other.

- **Embedded:** Embedded records are contained inside the record that embeds them. It is similar to composition relationship in UML.

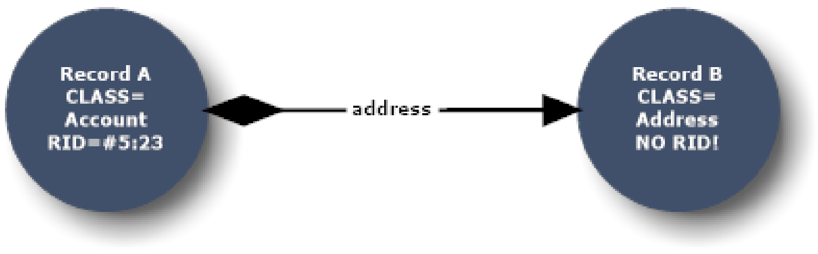


Figure 2-13 *Embedded relationship*

Record A will contain the entire record B in the property address.

One-to-one and many-to-one referenced/embedded relationships are expressed using *LINK/EMBEDDED* type. One-to-many and many-to-many referenced/embedded relationships are expressed by using collections of links: *LINKLIST/EMBEDDEDLIST* (ordered), *LINKSET/EMBEDDEDSET* (unordered), *LINKMAP/EMBEDDEDMAP* (ordered map of links with string key)

Vertices are records of type OGraphVertex and edges are records of type OGraphEdge by using graph Java API. Vertices and edges are records and they have their own recordID. Edges are always bidirectional, see Figure 2-14.

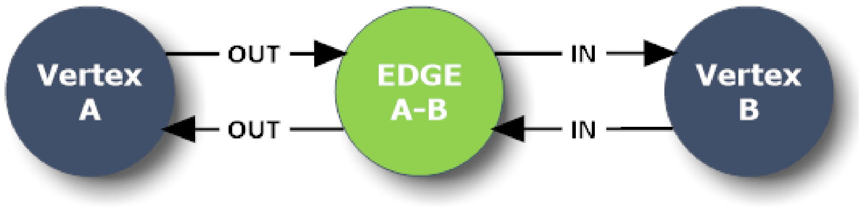


Figure 2-14 *Bidirectional edges in OrientDB graph model*

2.4.2 APIs

Information for this chapter was taken from [27]

OrientDB offers five types of API. APIs are different in their supported database model and in their level of abstraction. All APIs are native Java.

Table 2-1 *OrientDB API types*

API Type	Usage	Description	Speed
Object database	Object Oriented abstraction. All entities are bind to POJO.	Higher level database. It uses the Document to store objects at its core.	40%
TinkerPop graph database	It is designed for work with graphs. It is portable across TinkerPop Blueprints implementations.	It is the bridge to use OrientDB with all TinkerPop technologies	45%
Raw graph database	It is designed for work with graphs and for maximum performance.	Lower level graph API. It directly uses ODocument objects.	70%
Document database	It provides maximum performance and/or work with schema-less mode.	It handles records as documents. Fields can be any of supported types. It can be used in schema-less mode.	70%

Flat database	It provides maximum performance, but all records are Strings.	It contains only string content. There is no query capability or schema-full option, only direct access to records as strings.	100%

The speed column in Table 2-1 means speed comparison for generic CRUD operations, larger is better. Higher level of abstraction brings a speed penalty.

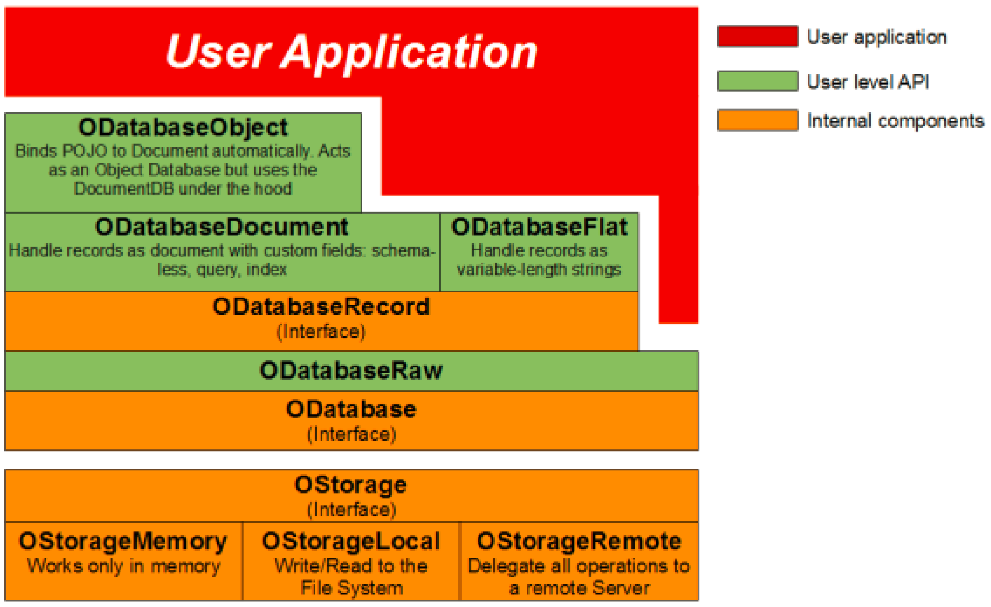

User Application
ODatabaseObject
Binds POJO to Document automatically. Acts as an Object Database but uses the DocumentDB under the hood
ODatabaseDocument
Handle records as document with custom fields: schema-less, query, index
ODatabaseFlat
Handle records as variable-length strings
ODatabaseRecord
(Interface)
ODatabaseRaw
ODatabase
(Interface)
OStorage
(Interface)
OStorageMemory
Works only in memory
OStorageLocal
Write/Read to the File System
OStorageRemote
Delegate all operations to a remote Server
User application
User level API
Internal components

***Figure 2-15** OrientDB – Java class stack [27]*

2.4.3 TINKERPOP BLUEPRINTS AND GRAPH MODEL

Information for this chapter was taken from [28]

As a suitable model for the EEG/ERP portal I choose OrientDB graph database API with TinkerPop Blueprints (generic graph API). The most important feature is the management of relationships as a *graph*, graph nodes are still documents. Blueprints provide interfaces and implementations for the property graph data model (see Figure 2-9). It is something like Java Database Connectivity (*JDBC*), but for graph databases. Blueprints allow to plug-and-play every compatible graph database backend, e.g. Neo4J supports Blueprints too.

Blueprints technology stack contains:

- **Pipes:** data flow framework
- **Gremlin:** a graph traversal language
- **Frames:** an object-to-graph mapper
- **Rexter:** a graph server

2.4.4 QUERY POSSIBILITIES

Information for this chapter was taken from [29] [30]

2.4.4.1 SQL

OrientDB is NoSQL database but it supports SQL as a language. SQL in OrientDB is unlike standard SQL extended by many new functions. It is important that in queries field names are case sensitive but class names are case insensitive.

OrientDB SQL supports these constructions:

WHERE conditions, SELECT projections, TRAVERSE to cross records by relationships, INSERT, UPDATE, DELETE, Create Vertex/Edge to work with graphs, GRANT, REVOKE, Create class/property, Alter class/property, Create index, Rebuild index, Create link, Alter cluster and next.

Typical query consist of:

- ***Items*** can be document fields, record attributes, columns, functions and context variables
- ***Operators*** can be applied to collections, any, strings, maps. There are classic logical operators, mathematical operators and new field operators
  - *Field operators* can be applied on document, map, lists, arrays, strings etc.
- ***Functions*** can be called in SQL SELECT and TRAVERSE statements. Some examples of functions are: `sysdate()`, `distance()`, `map(<field>|<key>,<value>>)`. We can also make own custom functions with a scripting language or via Java.

- ***Record attributes*** like *@this, @rid, @class* works directly with records

Query example with TRAVERSE command: Return all the vertices that have at least one friend (connected with out) up to the 3rd degree, that lives in Rome. [29]

```
SELECT FROM PROFILE
LET $temp = (
  SELECT FROM (
    TRAVERSE * FROM $current WHILE $depth <= 3
  )
  WHERE city = 'Rome'
)
WHERE $temp.size() > 0
```

Queries can be performed from OrientDB console, OrientDB Studio or directly via Java API.

2.4.4.2 GREMLIN LANGUAGE

Gremlin is a graph manipulation language. It is specialized to work with Property graphs. Gremlin is a part of TinkerPop Blueprints stack. It provides support for Java and it supports multiple traversal patterns.

Gremlin main features are:

- manual working with graph (create, delete, update, etc. vertices and edges, ensuring of integrity)
- to query graph; Gremlin is very efficient by querying the graph model
- exploring, analysis graphs
- exploring the semantic Web/Web of data; Gremlin can be used with RDF graphs and allows working with the semantic web in real-time
- gremlin is extensible with new methods and steps defined in Gremlin or in Java; Gremlin can take advantage of Java API
- it is a Turing complete language – it provides memory and computing constructs to support arbitrary path recognition

Gremlin has a collection of predefined steps. Gremlin steps map to a particular *Pipe*. Pipe<S,E> extends Iterator<E> and Iterable<E>. It takes object of type *S* and emits object of type *E*. Chaining pipes together creates *Pipeline*. Link in a pipeline is called *step* (see Figure 2-16).

Types of steps are: [30]

- **Transform**: take an object and emit a transformation of it (`map(strings..?), inV, gather{closure?}`, etc.)
- **Filter**: decide whether to allow an object to pass or not (`has('key',value), retain(collection),sort{closure?}` etc.)
- **SideEffect**: pass the object, but yield some side effect (`tree(map?, closures..?), groupCount(map?){closure?}{closure?}`, etc.)
- **Branch**: decide which step to take (`copySplit(pipes...)`, etc.)

Query example with pipeline demonstration (it is based on Figure 2-9): It gets all names and paths from vertex with ID = 1 (in Gremlin we have to choose arbitrary *root* vertex. The root vertex is the vertex from which searching starts. We can choose more than one vertex. Letter *g* is reference to the graph instance.

`g.v(1).out.name.paths`

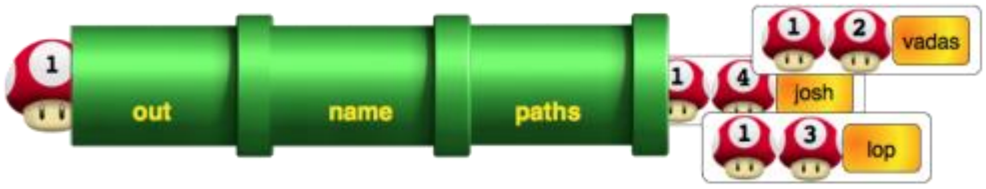


Figure 2-16 *Gremlin pipeline - get all names and paths from vertex with ID = 1 [20]*

Gremlin is usable from Gremlin console, OrientDB Studio or directly from Java API. Gremlin provides methods for working with graphs from Java API.

2.4.5 BINARY DATA MANAGEMENT

Information for this chapter was taken from [31]

OrientDB handles natively binary data (BLOB). There are different types of storing of BLOBs. BLOBs can be stored through Java API.

1) the data storing in a different path than the default database directory is

   This method can take advantage of faster hard disk (HD) like Solid State Disks (SSD) or it can utilize parallelism.

   E.g.: [31]

```
db.addDataSegment("binary", "/mnt/ssd");
db.addCluster("physical", "binary", "/mnt/ssd", "binary");
```

2) the data storing on file system and saving the path to the data in the document

   This method does not allow data distribution using the cluster.

   E.g.: [31]

```
ODocument doc = new ODocument(db);
doc.field("binary","/usr/local/orientdb/binary/test.pdf");
doc.save();
```

3) the data storing as a document's field

   ODocument class is able to manage BLOBs in form of byte[] (byte array). This is the easiest way to write BLOB but it is not effficient – content is serialized in Base64 – it means 33% more of disk space and a runtime cost in marschaling/unmarshaling of records.

   E.g.: [31]

```
ODocument doc = new ODocument(db);
doc.field("binary", "Binary data".getBytes());
doc.save();
```

4) the data storing by using ORecordBytes class

It is probably the best way to store BLOBs. ORecordBytes class is able to store binary content without conversions. It is the fastest way to handle BLOBs but it needs a separate record to handle it. Best way to reference it is to link it to a Document record.

E.g.: [31]

```
ORecordBytes record =
new ORecordBytes(db, "Binary data".getBytes());
ODocument doc = new ODocument(db);
doc.field("id", 12345);
doc.field("binary", record);
doc.save();
```

We can access binary data by traversing the *binary* field of the parent's document record.

E.g.: [31]

```
ORecordBytes record = doc.field("binary");
byte[] content = record.toStream();
```

ORecodrByte class can work with streams. For more information about this method see chapter 3.3.3.

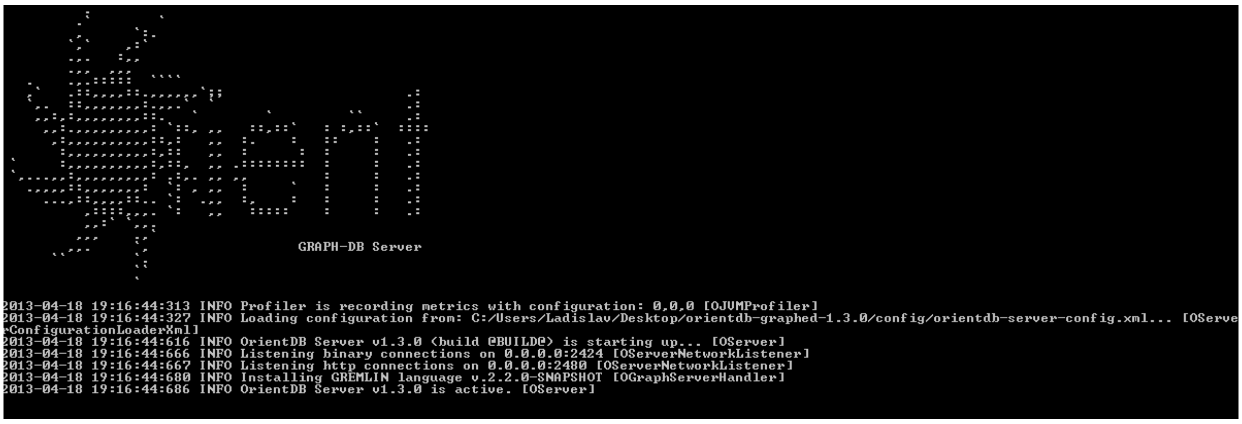
2.4.6 ORIENTDB SERVER

Information for this chapter was taken from [32]

The Server is a multi-thread Java application that listen remote commands and execute them in OrientDB database. It supports *binary* and Hyper Text Transfer Protocol (*HTTP*) protocols. Binary protocol is used by *OrientDB console* and HTTP protocol is used *by OrientDB Studio* application.

When server starts it is trying to acquire the port 2424 for the binary and 2480 for the HTTP one (if ports are already used, it will be taken the next one). We can configure multiple listeners by selecting the ip-address and *TCP/IP* (Transmission Control Protocol/Internet Protocol) port to bind in server's *configuration file*. Server can be extended by plug-ins, available plugins are: Automatic Backup, Email Plugin, Java Management Extension (*JMX*) Plugin, Distributed Server

Manager and Server-side script interpreter. More information about server configuration can be found in *Practical part* of this thesis.



```
                               GRAPH-DB Server

2013-04-18 19:16:44:313 INFO Profiler is recording metrics with configuration: 0,0,0 [OJVMProfiler]
2013-04-18 19:16:44:327 INFO Loading configuration from: C:/Users/Ladislav/Desktop/orientdb-graphed-1.3.0/config/orientdb-server-config.xml... [OServe
rConfigurationLoaderXml]
2013-04-18 19:16:44:616 INFO OrientDB Server v1.3.0 (build @BUILD@) is starting up... [OServer]
2013-04-18 19:16:44:666 INFO Listening binary connections on 0.0.0.0:2424 [OServerNetworkListener]
2013-04-18 19:16:44:667 INFO Listening http connections on 0.0.0.0:2480 [OServerNetworkListener]
2013-04-18 19:16:44:680 INFO Installing GREMLIN language v.2.2.0-SNAPSHOT [OGraphServerHandler]
2013-04-18 19:16:44:686 INFO OrientDB Server v1.3.0 is active. [OServer]
```

Figure 2-17 *Running OrientDB server listens on port 2424 for binary and on 2480 port for HTTP protocol*

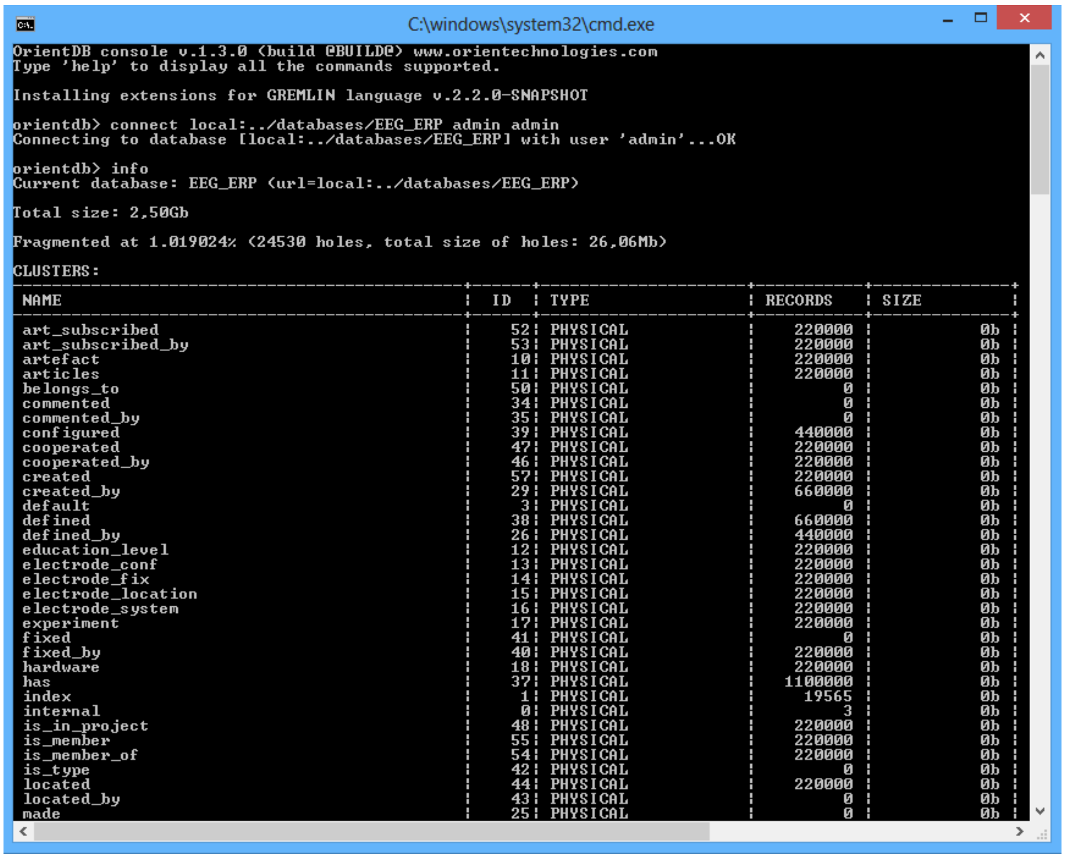
2.4.7 TOOLS FOR DATABASE MANAGEMENT

Information for this chapter was taken from [33] [34]

For database managing we have three following options:

1) OrientDB console

   It is a Java application and it works with OrientDB databases and Server instances. It supports multiple commands for database managing. It can work in *interactive mode* (default), when console is launched by executing the script or in a *batch mode*. The batch mode allows launching console script with parameters. Scripts are useful e.g. for data import. We can execute SQL and SQL-Gremlin mixed queries against OrientDB server, change server configuration, perform CRUD operations against records, classes and clusters, etc. If we use *local protocol,* then the access to the database is made without remote connection – this is much faster but the database must be on the same machine. Basic work with OrientDB console will be described in *Practical part* of this thesis.



```
OrientDB console v.1.3.0 (build @BUILD@) www.orientechnologies.com
Type 'help' to display all the commands supported.

Installing extensions for GREMLIN language v.2.2.0-SNAPSHOT

orientdb> connect local:../databases/EEG_ERP admin admin
Connecting to database [local:../databases/EEG_ERP] with user 'admin'...OK

orientdb> info
Current database: EEG_ERP (url=local:../databases/EEG_ERP)

Total size: 2,50Gb

Fragmented at 1.019024% (24530 holes, total size of holes: 26,06Mb)

CLUSTERS:
----------------------------------------------+------+---------------------+----------+------------+
 NAME                                         ¦  ID  ¦ TYPE                ¦ RECORDS  ¦ SIZE       ¦
----------------------------------------------+------+---------------------+----------+------------+
 art_subscribed                               ¦    52¦ PHYSICAL            ¦   220000 ¦         0b ¦
 art_subscribed_by                            ¦    53¦ PHYSICAL            ¦   220000 ¦         0b ¦
 artefact                                     ¦    10¦ PHYSICAL            ¦   220000 ¦         0b ¦
 articles                                     ¦    11¦ PHYSICAL            ¦   220000 ¦         0b ¦
 belongs_to                                   ¦    50¦ PHYSICAL            ¦        0 ¦         0b ¦
 commented                                    ¦    34¦ PHYSICAL            ¦        0 ¦         0b ¦
 commented_by                                 ¦    35¦ PHYSICAL            ¦        0 ¦         0b ¦
 configured                                   ¦    39¦ PHYSICAL            ¦   440000 ¦         0b ¦
 cooperated                                   ¦    47¦ PHYSICAL            ¦   220000 ¦         0b ¦
 cooperated_by                                ¦    46¦ PHYSICAL            ¦   220000 ¦         0b ¦
 created                                      ¦    57¦ PHYSICAL            ¦   220000 ¦         0b ¦
 created_by                                   ¦    29¦ PHYSICAL            ¦   660000 ¦         0b ¦
 default                                      ¦     3¦ PHYSICAL            ¦        0 ¦         0b ¦
 defined                                      ¦    38¦ PHYSICAL            ¦   660000 ¦         0b ¦
 defined_by                                   ¦    26¦ PHYSICAL            ¦   440000 ¦         0b ¦
 education_level                              ¦    12¦ PHYSICAL            ¦   220000 ¦         0b ¦
 electrode_conf                               ¦    13¦ PHYSICAL            ¦   220000 ¦         0b ¦
 electrode_fix                                ¦    14¦ PHYSICAL            ¦   220000 ¦         0b ¦
 electrode_location                           ¦    15¦ PHYSICAL            ¦   220000 ¦         0b ¦
 electrode_system                             ¦    16¦ PHYSICAL            ¦   220000 ¦         0b ¦
 experiment                                   ¦    17¦ PHYSICAL            ¦   220000 ¦         0b ¦
 fixed                                        ¦    41¦ PHYSICAL            ¦        0 ¦         0b ¦
 fixed_by                                     ¦    40¦ PHYSICAL            ¦   220000 ¦         0b ¦
 hardware                                     ¦    18¦ PHYSICAL            ¦   220000 ¦         0b ¦
 has                                          ¦    37¦ PHYSICAL            ¦  1100000 ¦         0b ¦
 index                                        ¦     1¦ PHYSICAL            ¦    19565 ¦         0b ¦
 internal                                     ¦     0¦ PHYSICAL            ¦        3 ¦         0b ¦
 is_in_project                                ¦    48¦ PHYSICAL            ¦   220000 ¦         0b ¦
 is_member                                    ¦    55¦ PHYSICAL            ¦   220000 ¦         0b ¦
 is_member_of                                 ¦    54¦ PHYSICAL            ¦   220000 ¦         0b ¦
 is_type                                      ¦    42¦ PHYSICAL            ¦        0 ¦         0b ¦
 located                                      ¦    44¦ PHYSICAL            ¦   220000 ¦         0b ¦
 located_by                                   ¦    43¦ PHYSICAL            ¦        0 ¦         0b ¦
 made                                         ¦    25¦ PHYSICAL            ¦        0 ¦         0b ¦
```

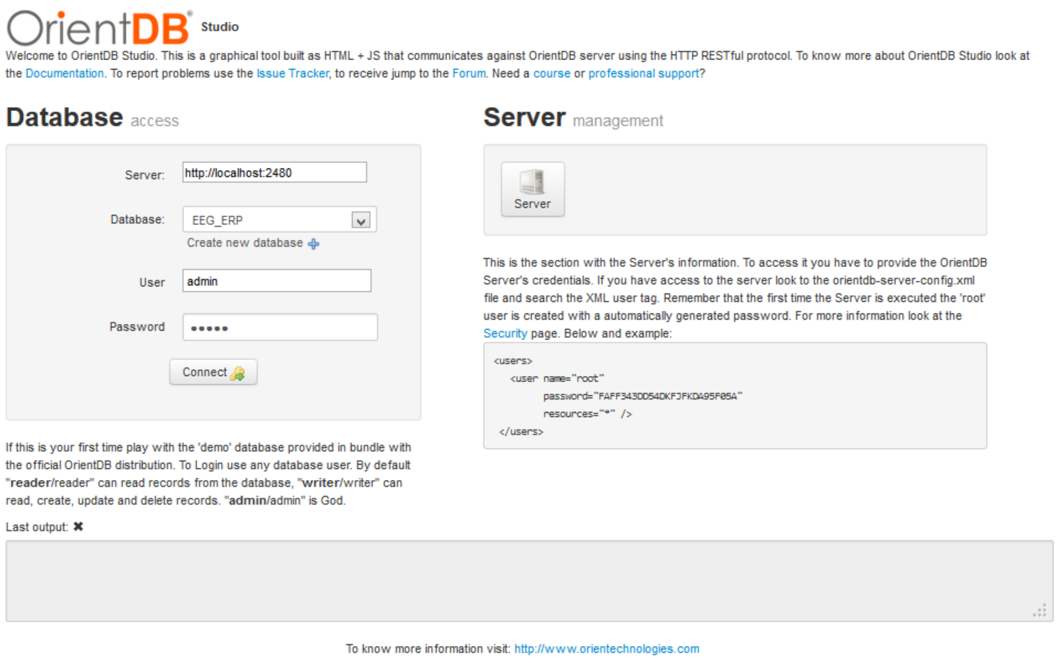
Figure 2-18 *OrientDB console with printed info about EEG_ERP database*

2) OrientDB Studio

It is a client-side web application, which is implemented using HyperText Markup Language (*HTML*), Cascading Style Sheets (*CSS*) and jQuery. OrientDB Studio requires running OrientDB server. Studio can be started in any web browser. OrientDB studio provides following functions:

- *executing the commands* against OrientDB server uses the OrientDB HTTP *REST* (REpresentational State Transfer) protocol and *AJAX* (Asynchronous Javascript and XML) calls, the response might be very slow on some browsers, especially with huge result sets
- *database tab* which contains information about currently-open database
- *managing the security* of database (roles and rules for roles)
- *executing of queries* in SQL/Gremlin
- *live editing* (on fly) - if query is executed, record‘s content can be directly edited

- *performing CRUD operations* on records, classes and clusters
- *monitoring* of server status and monitoring of all active connections


OrientDB Studio

Welcome to OrientDB Studio. This is a graphical tool built as HTML + JS that communicates against OrientDB server using the HTTP RESTful protocol. To know more about OrientDB Studio look at the Documentation. To report problems use the Issue Tracker, to receive jump to the Forum. Need a course or professional support?

Database access

Server: http://localhost:2480

Database: EEG_ERP

Create new database

User: admin

Password: •••••

Connect

If this is your first time play with the 'demo' database provided in bundle with the official OrientDB distribution. To Login use any database user. By default "**reader**/reader" can read records from the database, "**writer**/writer" can read, create, update and delete records. "**admin**/admin" is God.

Last output: ✖

Server management

Server

This is the section with the Server's information. To access it you have to provide the OrientDB Server's credentials. If you have access to the server look to the orientdb-server-config.xml file and search the XML user tag. Remember that the first time the Server is executed the 'root' user is created with a automatically generated password. For more information look at the Security page. Below and example:

```
<users>
   <user name="root"
         password="FAFF343DD54DKFJFKDA95F05A"
         resources="*" />
 </users>
```

To know more information visit: http://www.orientechnologies.com

Figure 2-19 *OrientDB Studio's logging page*

For more figures which show the work with OrientDB Studio see the Attachment F.

3) Java API

The last way to manage and configure database is through native Java API. With Java API we have the most complex option for database managing. It is the lowest level therefore we need advanced experiences in Java programming language and OOP principles. In Java API we have all and even more possibilities than in OrientDB console and OrientDB Studio (making all kinds of queries, creating of databases, clusters, records and relationships, making schema, making hooks (triggers), transaction management, importing and exporting the database, storing of binary files, etc.). Some basic work with OrientDB Java API can be found in *Practical part* of this thesis. The more information about OrientDB API we can found in *OrientDB API JavaDoc* or in *OrientDB Wiki pages.*

More information about OrientDB features can be found in the Attachment G.

3 PRACTICAL PART

The main goal of the practical part was to choose suitable NoSQL database system and attempt to improve current database layer of the EEG/ERP portal application by using the chosen NoSQL system. As it was mentioned I chose OrientDB database. The suitability of OrientDB for EEG/ERP portal was evaluated on the basis of its performance testing. OrientDB was compared with Oracle RDBMS on the sets of database queries and database commands on *the same* database model (the same relationships between records and the same database entities). In this part the whole process from the database model creation to the evaluation of measured results will be described.

3.1 CREATION OF EEG/ERP ORIENTDB GRAPH MODEL

It was necessary to make certain changes in model design before importing EEG/ERP model into OrientDB, because standard relational database model is very different from graph NoSQL database model.

3.1.1 LEFT OUT TABLES

As we know (see chapter 2.1.4.3), graph model manages relationships between records (nodes/vertices) like edges. Every node can have multiple *outgoing* and multiple *incoming* edges (this approach is similar to relational model when we have multiple relationships between tables).

OrientDB model supports collections. For more complex relationships like one-to-many, many-to-many *we do not need concept of foreign and primary keys anymore – no intersect tables (M:N).* All these types of tables were left out from the database model. This step very simplified the whole database model. Each M:N relationship was replaced by bidirectional edge (in both directions), the most intersect tables has suffix **_REL* in name*,* for better idea see Figure 3-1.

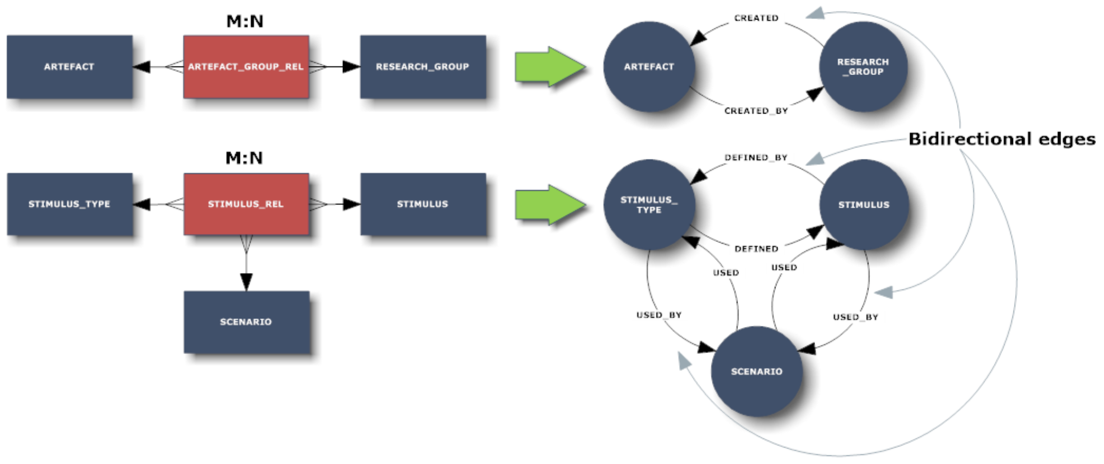


Figure 3-1 *Transformation M:N relationships into bidirectional edges*

Tables for managing XML files were also left out from the model. These tables contain XMLTYPE which is Oracle database native type – OrientDB does not support it and they are useless for testing purposes. See the Table 3-1 with left out tables. All used tables can be seen in the Attachment A - Table 1.

Table 3-1 *List of left out tables from OrientDB model*

Left out tables	Reason
All tables with suffix **_REL*	Intersect tables
RESEARCH_GROUP_MEMBERSHIP	Intersect table
EXPERIMENT_OPT_PARAM_VAL	Intersect table, property *PARAM_VALUE* is used as appropriate edge property
FILE_METADATA_PARAM_VAL	Intersect table, property *METADATA_VALUE* is used as appropriate edge property
PERSON_OPT_PARAM_VAL	Intersect table, property *PARAM_VALUE* is used as appropriate edge property
SCENARIO_TYPE_PARENT	Useless – it contains only *SCENARIO_ID* property (we do not need IDs)
Tables with these prefixes: *MD_*, MIGR_*, SYS_**	Oracle's tables created by recent database migration
Every table with prefix *SCENARIO_** except table *SCENARIO* and *SCENARIO_TYPE_NONXML*	These tables contain Oracle's XMLTYPE

3.1.2 DECOMPOSITION ON VERTICES

Every table in EEG/ERP portal RDBMS can be decomposed on vertices. One node represents one row in RDBMS table. In graph database are columns (known from relational databases) called *properties* and in document database are called *fields*, example:

***Table 3-2** Comparison of Oracle database table row record with appropriate OrientDB's node record*

One row of table RESEARCH_GROUP in Oracle RDBMS		One node of type RESEARCH_GROUP in OrientDB graph database	
Column	***Value***	***Property (field)***	***Value***
RESEARCH_GROUP_ID	204647	RESEARCH_GROUP_ID	204647
OWNER_ID	26916	OWNER_ID	26916
TITLE	Group title	TITLE	Group title
DESCRIPTION	Group description	DESCRIPTION	Group description
-	-	**@type**	d
-	-	**@rid**	#21:214207
-	-	**@version**	34
-	-	**@class** (node type)	RESEARCH_GROUP
-	-	**out** (list of outgoing vertices)	#28:12, #30:100…
-	-	**in** (list of incoming vertices)	#26:1622, #27:50…

3.1.3 DECOMPOSITION ON EDGES AND SEMANTICS DESCRIPTION OF EDGES BETWEEN NODES

Edges are also documents in OrientDB. Every single edge has own arbitrary properties and the list of outgoing and incoming nodes. Thanks this feature we can express the semantics between nodes. Property in edge type node is reserved for this purpose, this property is called *label* and it is of type *String*. These labels can be seen on Figure 3-1 (descriptions of edges).Essentially *label* property is like any other property (e.g. *NAME, TITLE,* etc.). Labels semantically express the relationships between nodes. We can utilize this fact for making more natural and readable queries, e.g.:

The relationship between article which was created by person can be described with label *CREATED_BY* and the query could sound like – *„Get all articles which were created by person with name Peter".*

The whole semantics between all nodes can be seen in Attachment A - Table A.1. In fact, it is *possible to construct the whole OrientDB graph model of EEG/ERP portal database* by this table – see the Attachment B.

3.1.4 INSTALLATION OF ORIENTDB

For my purposes I chose OrientDB Graph Edition v1.3.0. The whole database is distributed as *zip* archive. Zip archive can be extracted into arbitrary directory on the disk. Database is ready to run after extracting. For the correct run OrientDB requires Java Virtual Machine (*JVM*) 1.6 32/64 bit and higher. The most important directories of OrientDB *root* directory are:

- *bin* – It contains all executable *.bat* and *.sh* scripts for the server, console and gremlin console. Server or console can be simply started by clicking on appropriate file.
- *config* – It contains XML configuration files for server.
- *databases* – It contains all existing databases. It is a default path, but databases can be outside the root directory as well.
- other directories – They contain log files, OrientDB Studio source files, etc.

3.1.5 DATABASE CREATION

New database can be created through Java API, OrientDB studio or by using the OrientDB Console. I chose the third option – Console.

Example of statement syntax for database creation:

```
CREATE database <database-url> <user> <password> <storage-type> <db-type>
```

- **<database-url>** is an url of the created database, url is in the format `<mode>:<path>`
- **<user> is** the database's user name in *local* mode , in remote mode it is the Server's administrator name

- **<password> is** the server's password in *local* mode , in remote mode it is the Server's administrator name
- **<storage-type> is** the type of the storage ('local' for disk-based database, 'memory' for in-memory database)
- **<db-type>** is the type of database model, it can be chosen between *document* database (the default one) and *graph* database model; this parameter is optional

I created *graph* database with name *EEG_ERP* in *remote* mode. If we choose remote mode, then the *user name* must be *root* and the *password* can be found in server configuration file `orientdb-server-config.xml` in the xml element `<users>`.

Example of new OrientDB database creation in remote mode:

```
orientdb>create database remote:localhost/EEG_ERP root
        5B51EF035C4908D023C16227BEEA2BBED13EE40EC4890BF541026EFA1BBECBB2
        local graph
```

Example of new OrientDB database creation in local mode:

```
orientdb>create database local:../databases/EEG_ERP admin admin
        local graph
```

3.1.6 IMPORT OF EEG/ERP RDBMS MODEL INTO ORIENTDB GRAPH DATABASE

We have three previous options (see chapter 3.1.5) to import RDBMS model and data into OrientDB graph database. Following procedure concerns OrientDB Console.

OrientDB supports importing of SQL scripts with data from RDBMS but this procedure is not fully automatic. Script file can generally have arbitrary extension – not only **.sql.* It is necessary to use console in *batch* mode when we give SQL script as first argument. An example of importing script with name *database.sql*:

```
C:\orientdb-graphed-1.3.0\bin>console.bat database.sql
```

3.1.6.1 CREATION OF CLASSES FOR VERTICES AND EDGES

It is necessary to create appropriate *classes* before importing the data. As it was mentioned before, OrientDB supports custom types of vertices and edges *in object oriented manner.* Basic type of the vertex is *OGraphVertex* and basic type of the edge is *OGraphEdge*. From these basic types I created *sub-classes* for custom types. Names of the vertex types are derived from *chosen tables* (see the Attachment A - Table A.1) of the EEG/ERP portal database. Names of edge types are derived from *chosen labels* (see the Attachment A - Table A.1).

SQL script can be also used for this purpose. This script has the following parts:

1) **Header** which contains definition of the database connection. For better performance I choose *local* connection which avoids remote connection (without starting server) and it uses faster *binary* protocol. Command syntax is:

   ```
   CONNECT <path> <username> <password>;
   ```

   Example of the local protocol connection:

   ```
   CONNECT local:../databases/EEG_ERP admin admin;
   ```

   Example of the remote protocol connection:

   ```
   CONNECT remote:localhost/EEG_ERP admin admin;
   ```

2) List of *vertex classes* which are created as subclasses of *V* (alias for OGraphVertex in Java API). Command syntax is:

   ```
   CREATE CLASS <class-name> EXTENDS V;
   ```

   e.g.: `CREATE CLASS PERSON EXTENDS V;`

3) List of **edges classes** which are created as subclasses of **E** (alias for OGraphEdge in Java API). Command syntax is:

   ```
   CREATE CLASS <class-name> EXTENDS E;
   ```

   e.g.: `CREATE CLASS CREATED_BY EXTENDS E;`

Complete script file may look like:

```
CONNECT local:../databases/EEG_ERP admin admin;

CREATE CLASS PERSON EXTENDS V;

... ;

CREATE CLASS CREATED_BY EXTENDS E;

... ;
```

Database is ready for data import after creating the appropriate classes.

Note: Any supported commands of OrientDB Console can be used in the scripts.

3.1.6.2 IMPORT OF DATA INTO VERTICES

OrientDB supports SQL command *INSERT INTO TABLE (VALUES)*. This feature is useful when we want to import data from existing RDBMS database. We only need export data from Oracle database appropriate table.

Exported SQL script with data from RDBMS table *RESEARCH_GROUP* may look like:

```
Insert into RESEARCH_GROUP (RESEARCH_GROUP_ID,OWNER_ID,TITLE,DESCRIPTION)
values (151,1744,'Saab','Description number 5483222');
Insert into RESEARCH_GROUP (RESEARCH_GROUP_ID,OWNER_ID,TITLE,DESCRIPTION)
values (152,1754,'Suzuki','This is a description 6112802');
...
```

Do not forget that one row of RDBMS table corresponds with one OrientDB vertex, see Figure 3-2.

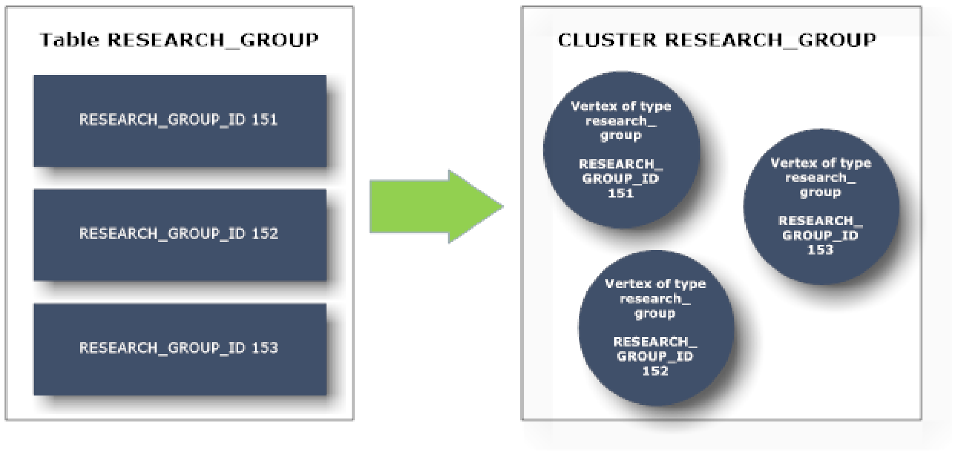


Table RESEARCH_GROUP
RESEARCH_GROUP_ID 151
RESEARCH_GROUP_ID 152
RESEARCH_GROUP_ID 153
CLUSTER RESEARCH_GROUP
Vertex of type research_ group RESEARCH_ GROUP_ID 151
Vertex of type research_ group RESEARCH_ GROUP_ID 153
Vertex of type research_ group RESEARCH_ GROUP_ID 152

Figure 3-2 *Table rows decomposed on OrientDB vertices*

The table name RESEARCH_GROUP corresponds with created vertex named RESEARCH_GROUP. Unlike RDBMS we do not need fixed schema (like RDBMS tables). OrientDB itself stores data into appropriate vertices of type RESEARCH_GROUP in the correct data types thanks to schema-less approach.

This script is different from the script for vertices and edges creation. A difference is in its *header*. The script may contain thousands of records - for large datasets we need to declare massive insertion intent. "Massive insert" intent will auto tune the OrientDB engine for fast insertion. I used *local* connection for faster insertion.

Complete script may look like:

```
CONNECT local:../databases/EEG_ERP admin admin;

DECLARE INTENT massiveinsert;

Insert into RESEARCH_GROUP (RESEARCH_GROUP_ID,OWNER_ID,TITLE,DESCRIPTION)
values (151,1744,'Saab','Description number 5483222');
Insert into RESEARCH_GROUP (RESEARCH_GROUP_ID,OWNER_ID,TITLE,DESCRIPTION)
values (152,1754,'Suzuki','This is a description 6112802');
...
```

Note: Only problem may occurs by storing the date. Oracle database stores date in this way:

```
to_date('21.11.2010','DD.MM.RRRR')
```

OrientDB does not support function `to_date`. Instead of this function it has `sysdate` function. We need to replace `to_date` with `sysdate` function.

3.1.6.3 CREATION OF EDGES BETWEEN RELEVANT NODES

The creation of edges between relevant nodes is not so straightforward. For this purpose I used original *ID values* of records from RDBMS, e.g.:

In Oracle RDBMS we have relation between record PERSON with ID = 178 and more records of RESEARCH_GROUP. This relationship expresses ownership between person and research groups (person with ID = 178 owns some research groups).

SQL *sub-queries*, which are also supported by OrientDB, were used for edges creation between relevant vertices. Using of sub-queries is exactly querying the database. For better understanding see Figure 3-3.

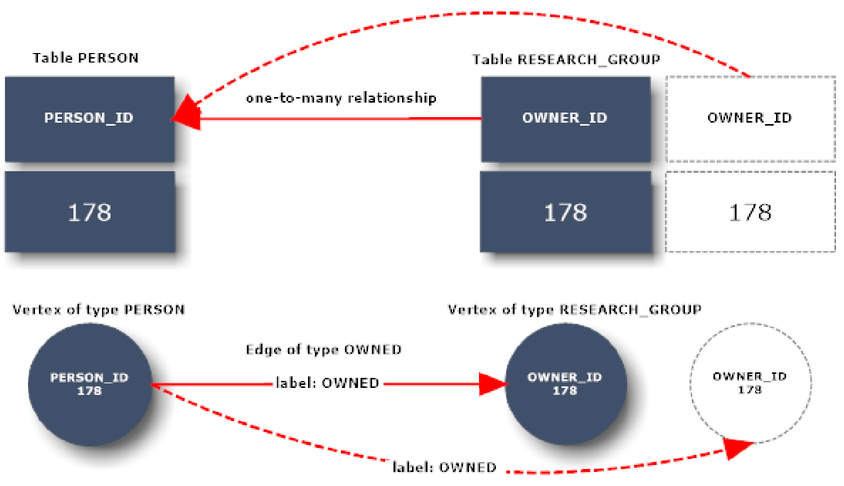


Figure 3-3 *RDBMS one-to-many relationship transformation into OrientDB form (we can see that relationships are reversed)*

Relevant OrientDB sub-query in SQL script for Figure 3-3 looks like:

```
create edge OWNED from (select from PERSON where PERSON_ID = 178)

to (select from RESEARCH_GROUP where OWNER_ID = 178) set label = 'OWNED';
```

Indexes are very important thing by using approach with sub-query. Without indexes the performance of insertion is very slow (in fact query look up is slow) because it leads to the linear scanning of all vertices located in the database graph. Index tree is used with indexes. It is necessary to put unique index of type integer on every *ID* property in *WHERE* condition. The indexes of OrientDB are applicable only on schema's properties (all properties are in schema-less mode after importing the data). Index creating from console is composed from these two steps:

1) schema property creation (properties are accessed by using dot notation like in OOP approach)

   ```
   orientdb>create property person.PERSON_ID integer
   ```

2) unique index creating

   ```
   orientdb>create index person.PERSON_ID unique integer
   ```

I applied this procedure on every possible relationship among vertices (see the Attachment A – Table A.1). It is important to declare *massive insertion intent* in every script**.** Creation of edges, which expresses *OWNED* relationship between vertices of the type PERSON and vertices of type RESEARCH_GROUP, may look like:

```
CONNECT local:../databases/EEG_ERP admin admin;
DECLARE INTENT massiveinsert;
create edge OWNED from (select from PERSON where PERSON_ID = 178)
to (select from RESEARCH_GROUP where OWNER_ID = 178) set label = 'OWNED';
...
```

The edge‘s name *OWNED* corresponds with created *edge type OWNED*.

If we need to add other properties to the edge, we must separate properties by commas:

```
...set label = 'OWNED', AUTHORITY = '1';
```

I generated the script files by Java program directly from Oracle database. All what is need is to acquire the connection with Oracle database through JDBC. Finally, appropriate methods for saving of script files must be written.

3.1.6.4 PERFORMANCE TUNING

Enough memory should be set for Client/Server Java heap (*-Xmx* parameter for Java process) before importing of data. . OrientDB’s Client/Server *file.mmap.maxMemory* is next very important parameter. This parameter has the influence on value of the max memory for OrientDB’s *memory mapping manager*. This parameter is important especially on 32bit architectures. Difference is when we are using 32bit architecture with 32bit Java Virtual Machine (JVM) or 64bit architecture with 64bit JVM. The speed of insertion drastically slows down without following settings – especially for big datasets (GBs of data). Below, you can see my settings of OrientDB Server/ Client on different architectures.

- 32bit architecture (3GB RAM)
    - `-Xmx800m`
    - for right functionality of the file.mmap.maxMemory parameter we must enable OrientDB's *Old manager*
        - -D`file.mmap.useOldManager=true`
        - -D`file.mmap.maxMemory=1300mb` (default value is only 134MB)
- 64bit architecture (6GB RAM)
    - `-Xmx2048m`
    - it is not necessary to set the file.mmap.maxMemory parameter because the default value is (maxOsMemory – maxProcessHeapMemory)/2

All mentioned parameters can be set directly on JVM level in `Console.bat` or in `Server.bat` scripts.

Xmx parameter can be set on the following line:

```
set JAVA_OPTS_SCRIPT=-XX:+AggressiveOpts -XX:CompileThreshold=200
-Xmx2048m
```

-*D`file.mmap.useOldManager`* and -*D`file.mmap.maxMemory`* parameters can be set on the following line:

```
set ORIENTDB_SETTINGS=-Dfile.mmap.useOldManager=true,
-Dfile.mmap.maxMemory=1300mb
```

3.2 CREATION OF DATABASE MODEL FOR TESTING PURPOSES

I will describe the way of creating Oracle test database and OrientDB test database in this chapter.

Note: Firstly, I created the whole model of the EEG/ERP portal test database, which is running on the server `students.kiv.zcu.cz`. However, the database contains little amount of testing data, therefore I decided to create sub-model of EEG/ERP portal with generated data for testing purposes.

3.2.1 SOFTWARE ASSUMPTIONS

- Oracle test database
  - Server: Oracle Server 11g
  - Client: Oracle SQL Developer 3.2
  - JVM 1.6 and higher
- OrientDB test database
  - OrientDB graphed 1.3.0
  - JVM 1.6 and higher

I used Windows 8 64bit operating system for both databases.

3.2.2 DATABASE MODEL

Previous chapter 3.1 expected the whole EEG/ERP portal model. I locally created new Oracle RDBMS sub-model of EEG/ERP portal model on my machine for testing purposes. I imported new database model (without data) as the script from the original model. Sub-model is composed from chosen following tables (it is a core of the original database, see Table 3-3). Appropriate OrientDB alternatives of vertices for every chosen Oracle database table are listed in Table 3-3. For more information about direction of edges see Table 1 in the Attachment A.

Table 3-3 *OrientDB's alternatives for Oracle database's tables*

Table of Oracle test database	Alternative in OrientDB test database
ARTEFACT	Vertices of type ARTEFACT
ARTEFACT_GROUP_REL	Edges of type CREATED Edges of type CREATED_BY
ARTICLES	Vertices of type ARTICLES
ARTICLES_GROUP_SUBSCRIBTIONS	Edges of type ART_SUBSCRIBED Edges of type ART_SUBSCRIBED_BY
COEXPERIMENTER_REL	Edges of type COOPERATED Edges of type COOPERATED_BY
EDUCATION_LEVEL	Vertices of type EDUCATION_LEVEL
EDUCATION_LEVEL_GROUP_REL	Edges of type DEFINED_BY

	Edges of type HAS
ELECTRODE_CONF	Vertices of type ELECTRODE_CONF
ELECTRODE_FIX	Vertices of type ELECTRODE_FIX
ELECTRODE_FIX_GROUP_REL	Edges of type USED Edges of type USED_BY
ELECTRODE_LOCATION	Vertices of type ELECTRODE_LOCATION
ELECTRODE_LOCATION_GROUP_REL	Edges of type LOCATED Edges of type LOCATED_BY
ELECTRODE_LOCATION_REL	Edges of type HAS Edges of type CONFIGURED
ELECTRODE_SYSTEM	Vertices of type ELECTRODE_SYSTEM
ELECTRODE_SYSTEM_GROUP_REL	Edges of type DEFINED Edges of type DEFINED_BY
EXPERIMENT	Vertices of type EXPERIMENT
HARDWARE	Vertices of type HARDWARE
HARDWARE_GROUP_REL	Edges of type USED Edges of type USED_BY
HARDWARE_USAGE_REL	Edges of type USED Edges of type USED_BY
PERSON	Vertices of type PERSON
PROJECT_TYPE	Vertices of type PROJECT_TYPE
PROJECT_TYPE_GROUP_REL	Edges of type OWNED Edges of type OWNED_BY
PROJECT_TYPE_REL	Edges of type IS_IN_PROJECT Edges of type HAS
RESEARCH_GROUP	Vertices of type RESEARCH_GROUP
RESEARCH_GROUP_MEMBERSHIP	Edges of type IS_MEMBER Edges of type IS_MEMBER_OF
SCENARIO	Vertices of type SCENARIO
SUBJECT_GROUP	Vertices of type SUBJECT_GROUP
WEATHER	Vertices of type WEATHER
WEATHER_GROUP_REL	Edges of type HAS Edges of type OWNED_BY

It is possible to construct the whole OrientDB test model from this table (see Figure 3-4).

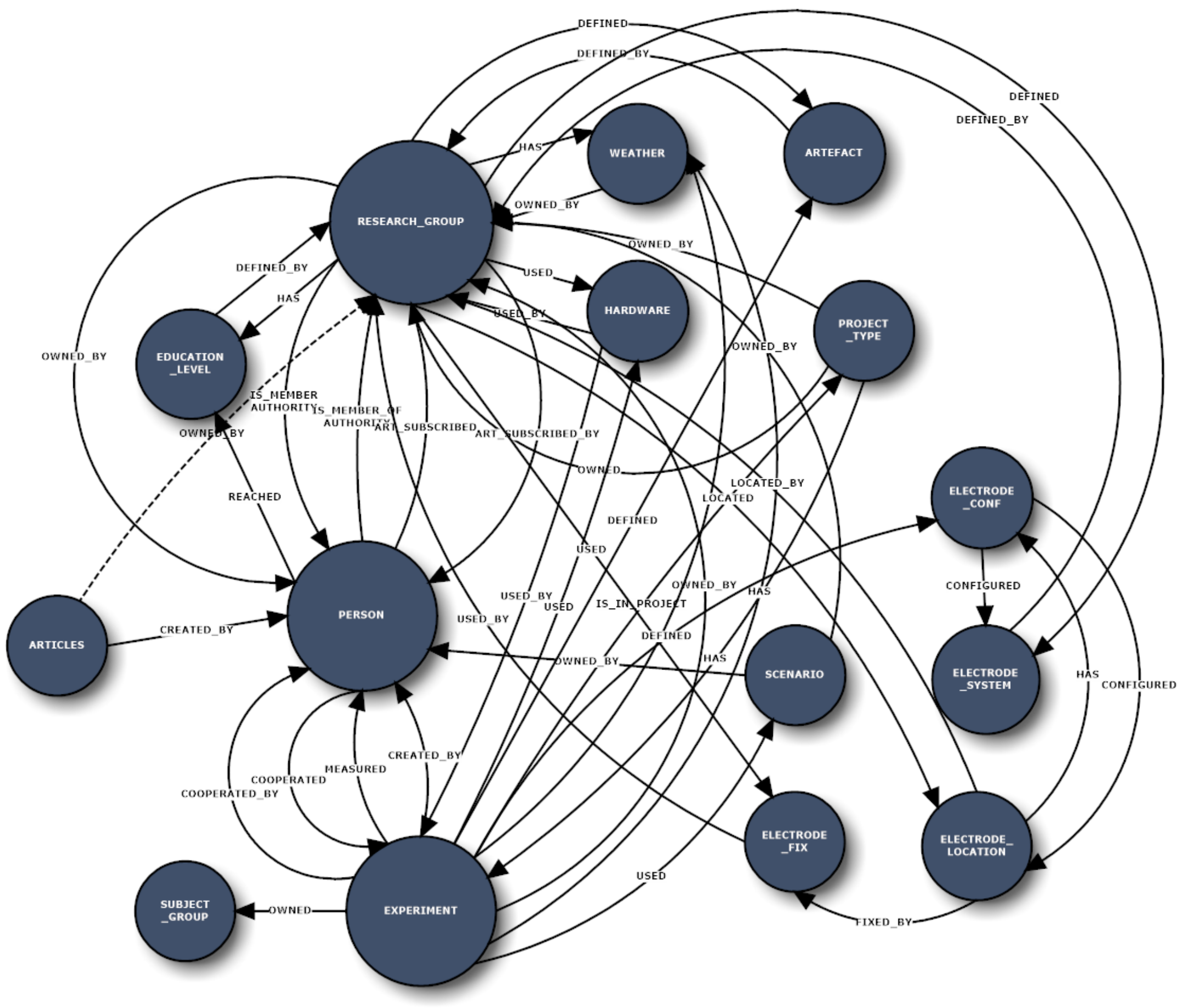


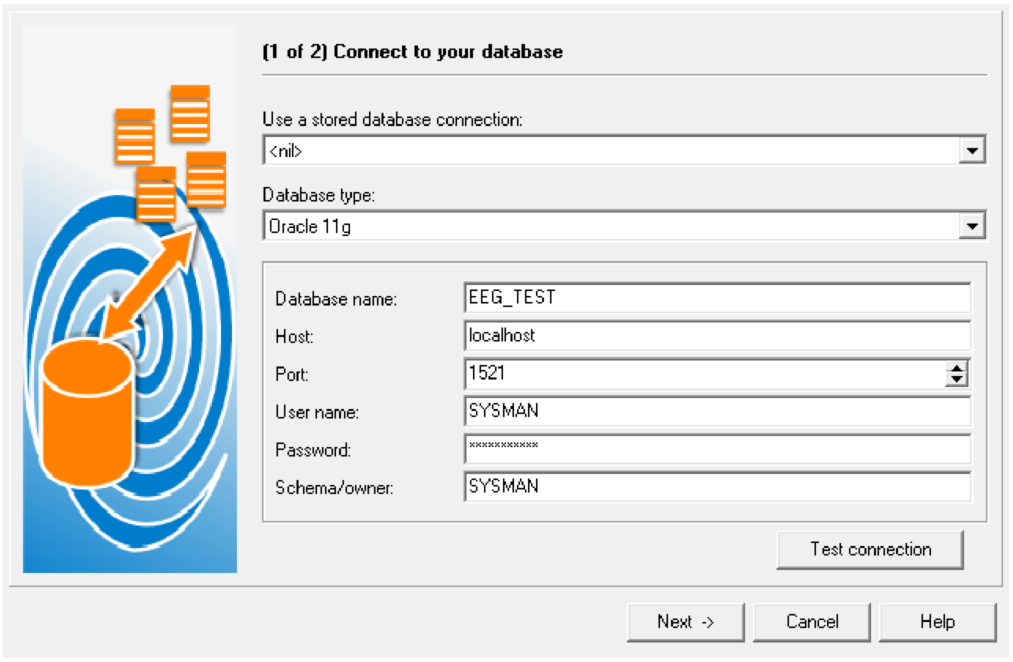
Figure 3-4 *OrientDB test database model*

Note: Dashed edges on Figure 3-4 indicate that these relationships were optional in Oracle database. This fact loses significance in OrientDB schema-less graph model.

3.2.3 GENERATING OF TESTING DATA

Testing data were generated into Oracle RDBMS test database by *Data Generator for Oracle 2011* which is developed by *Datanamic*. University of West Bohemia owns license for this software. Basic process for data generation is following:

1) connecting to the Oracle test database



(1 of 2) Connect to your database

Use a stored database connection: <nil>

Database type: Oracle 11g

Database name: EEG_TEST

Host: localhost

Port: 1521

User name: SYSMAN

Password: ***********

Schema/owner: SYSMAN

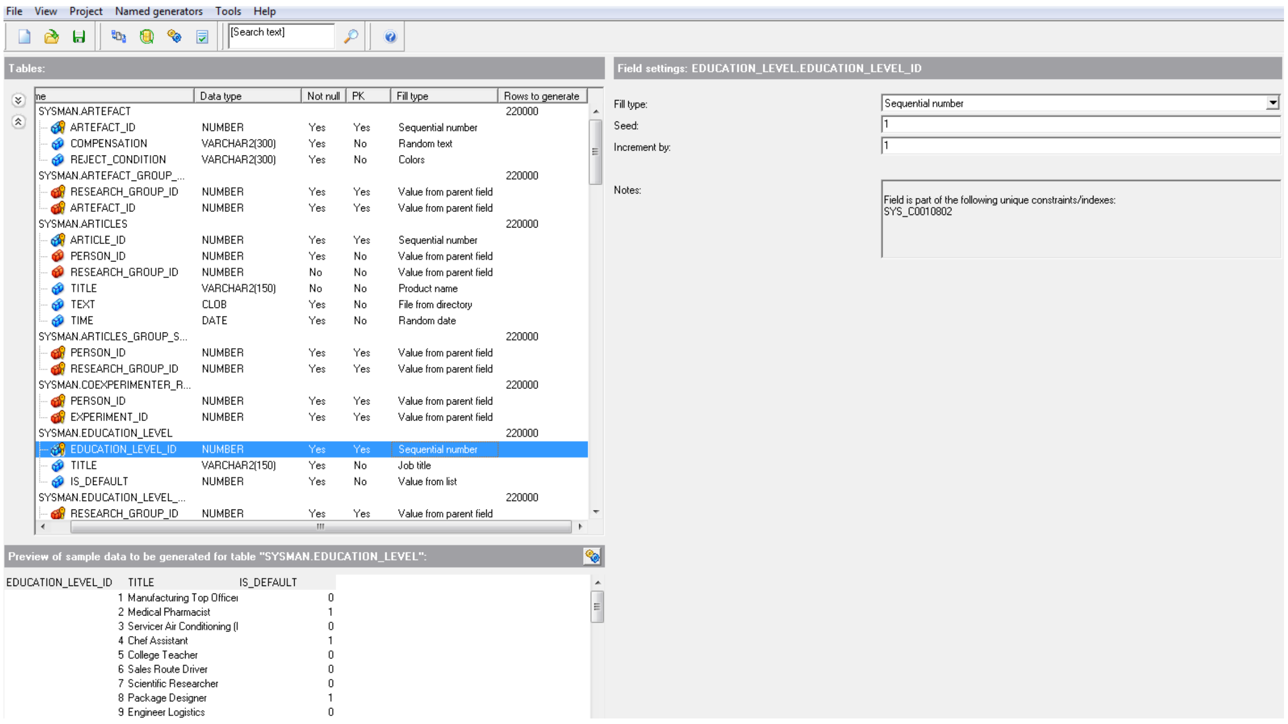
Figure 3-5 *Oracle data generator – Connection window*

2) selection of tables for which we want to generate data

3) settings of the generated data

   220 000 records were chosen for data generating for each relational table of the test database. Primary and foreign keys are generated automatically. By default, appropriate generators are automatically assigned for columns. However, manually assigned data generators were used for each table column. User generators can be also defined but predefined generators were used in this case. For the window with generating settings see Figure 3-6.

4) starting of generating (the generation may take a long time for big datasets, it depends on used configuration)



ne	Data type	Not null	PK	Fill type	Rows to generate
SYSMAN.ARTEFACT					220000
ARTEFACT_ID	NUMBER	Yes	Yes	Sequential number	
COMPENSATION	VARCHAR2(300)	Yes	No	Random text	
REJECT_CONDITION	VARCHAR2(300)	Yes	No	Colors	
SYSMAN.ARTEFACT_GROUP_...					220000
RESEARCH_GROUP_ID	NUMBER	Yes	Yes	Value from parent field	
ARTEFACT_ID	NUMBER	Yes	Yes	Value from parent field	
SYSMAN.ARTICLES					220000
ARTICLE_ID	NUMBER	Yes	Yes	Sequential number	
PERSON_ID	NUMBER	Yes	No	Value from parent field	
RESEARCH_GROUP_ID	NUMBER	No	No	Value from parent field	
TITLE	VARCHAR2(150)	No	No	Product name	
TEXT	CLOB	Yes	No	File from directory	
TIME	DATE	Yes	No	Random date	
SYSMAN.ARTICLES_GROUP_S...					220000
PERSON_ID	NUMBER	Yes	Yes	Value from parent field	
RESEARCH_GROUP_ID	NUMBER	Yes	Yes	Value from parent field	
SYSMAN.COEXPERIMENTER_R...					220000
PERSON_ID	NUMBER	Yes	Yes	Value from parent field	
EXPERIMENT_ID	NUMBER	Yes	Yes	Value from parent field	
SYSMAN.EDUCATION_LEVEL					220000
EDUCATION_LEVEL_ID	NUMBER	Yes	Yes	Sequential number	
TITLE	VARCHAR2(150)	Yes	No	Job title	
IS_DEFAULT	NUMBER	Yes	No	Value from list	
SYSMAN.EDUCATION_LEVEL_...					220000
RESEARCH_GROUP_ID	NUMBER	Yes	Yes	Value from parent field	

Preview of sample data to be generated for table "SYSMAN.EDUCATION_LEVEL":

EDUCATION_LEVEL_ID	TITLE	IS_DEFAULT
1	Manufacturing Top Officer	0
2	Medical Pharmacist	1
3	Servicer Air Conditioning (I	0
4	Chef Assistant	1
5	College Teacher	0
6	Sales Route Driver	0
7	Scientific Researcher	0
8	Package Designer	1
9	Engineer Logistics	0

Field settings: EDUCATION_LEVEL.EDUCATION_LEVEL_ID

Fill type: Sequential number
Seed: 1
Increment by: 1
Notes: Field is part of the following unique constraints/indexes:
SYS_C0010802

Figure 3-6 Oracle data generator – Settings window

OrientDB test database was made according to chapter 3.1. OrientDB test database statistics can be seen in the Attachment C – Table C.1 and Table C.2.

3.3 TESTING OF ORIENTDB

Testing procedure of OrientDB database in comparison with Oracle database will be described in this chapter and consequently the analysis of results will be made.

3.3.1 USED TESTING METHOD

The alternative is provided for every tested *query* and *command* in Oracle database SQL. *Ten* measurements were performed for every query and *five* measurements were performed for commands. Times of individual measurements of queries are summed and averaged. Measurements are made on different number of returned records for all queries (except queries with JOINs, because JOIN operations do not return so big number of results).

The difference in the performance of both tested databases is shown by using *percentage score.* Percentage scores for various types of measured queries and commands were evaluated. Score for each measurement is evaluated for different number of returned results.

3.3.2 TESTING CONFIGURATION, TESTING SOFTWARE AND TESTED DATABASES SETTING

Following tables show specification of used HW and SW resources, specification of testing software and settings of database servers.

Table 3-4 *Testing configuration specification*

Hardware	
Component	**Specification**
Central Processing Unit (*CPU*)	Intel(R) Core ™ i3-3110M CPU @ 2.40GHz (2 cores, 4 logical processors)
Random Access Memory (*RAM*)	6 GB DDR3, 1600 MHz
Hard Disk Drive (*HDD*)	500 GB, 7200 rpm

Table 3-5 *Testing software specification*

Software	
OrientDB	**Oracle database**
Server: OrientDB 1.3.0 graphed	**Server:** Oracle 11g Enterprise edition 64bit
Testing client: Java application	**Testing client**: Oracle SQL developer 3.2.20.09 64bit
Operating system: Windows 8 64bit	
JVM: 1.7 64bit	

Table 3-6 *Database servers settings*

OrientDB server settings on JVM level	
JVM parameters set JAVA_OPTS_SCRIPT=	-XX:+AggressiveOpts -XX:CompileThreshold=200 -Xmx2048m
OrientDB parameters (cache settings) set ORIENTDB_SETTINGS=	-Dcache.level1.enabled=false -Dcache.level2.enabled=false
Oracle server settings	
memory_target parameter	2048MB
cache settings	After every executed query - flushing of cache: ALTER SYSTEM FLUSH BUFFER_ CACHE

3.3.3 ORIENTDB JAVA API BASICS

OrientDB testing queries and commands were tested through Java API. I mention here basic OrientDB Java API structures. These structures are assumptions for snippets of Java code in following chapters.

1) needed `*.jar` libraries

   blueprints-core-2.2.0, blueprints-orient-graph-2.2.0, gremlin-java-2.2.0, pipes-2.2.0, orientdb-core-1.3.0, orientdb-graphdb-1.3.0, orient-commons -1.3.0, orientdb-enterprise-1.3.0

2) new database instance creation; remote connection to the database server

```
//OrientGraph is Blueprints implementation of the graph database
//OrientDB (see chapter 2.4.3)

OrientGraph graph = new OrientGraph("remote:localhost/EEG_ERP");
```

3) new database instance creation for batch processing tasks

```
OrientBatchGraph bgraph =

//method getRawGraph makes available the underlaying OrientDB graph

new OrientBatchGraph(graph.getRawGraph());
```

4) closing the database after the end of the work with it (closing is strongly recommended, otherwise data corruption may be caused)

```
graph.shutdown();
```

5) shutting down the memory cache (it has to be done before database instance creation)

```
OGlobalConfiguration.CACHE_LEVEL1_ENABLED.setValue(false);

OGlobalConfiguration.CACHE_LEVEL2_ENABLED.setValue(false);
```

6) massive insertion settings

```
bgraph.getRawGraph().declareIntent(new OIntentMassiveInsert());

bgraph.getRawGraph().setValidationEnabled(false);
```

   Before creation of the database instance following settings have to be done.

```
OGlobalConfiguration.STORAGE_KEEP_OPEN.setValue(false);

OGlobalConfiguration.TX_USE_LOG.setValue(false);
OGlobalConfiguration.ENVIRONMENT_CONCURRENT.setValue(false);
```

7) enabling of usage of custom type vertices

```
graph.getRawGraph().setUseCustomTypes(true);
```

8) Java query/command execution time was measured in milisecs

```
long startTime = System.currentTimeMillis();

//place for code of query/command

double estimatedTime =

(System.currentTimeMillis() - startTime) / 1000.0;
```

3.3.4 BINARY FILES STORING

Process of binary data storage into testing database will be described before testing of queries. This method is based on the method no. 4 in chapter 2.4.5. Basic assumption for this method is that we have exported BLOBs from Oracle database somewhere on the File System. BLOBs uploading method was tested on files with size about 200 MB (it is standard size of the data from one experimental measurement in EEG laboratory. Upload was performed without any problems.

1) ORecordBytes instance creation with appropriate parameter

```
//ORecordBytes takes parameter which return bytes array
//method BytesReader is general method which returns bytes array
ORecordBytes BLOBrecord =
new ORecordBytes(BytesReader.getBytesFromFile("fileName"));
```

2) selection of appropriate document (vertex) for binary file storing

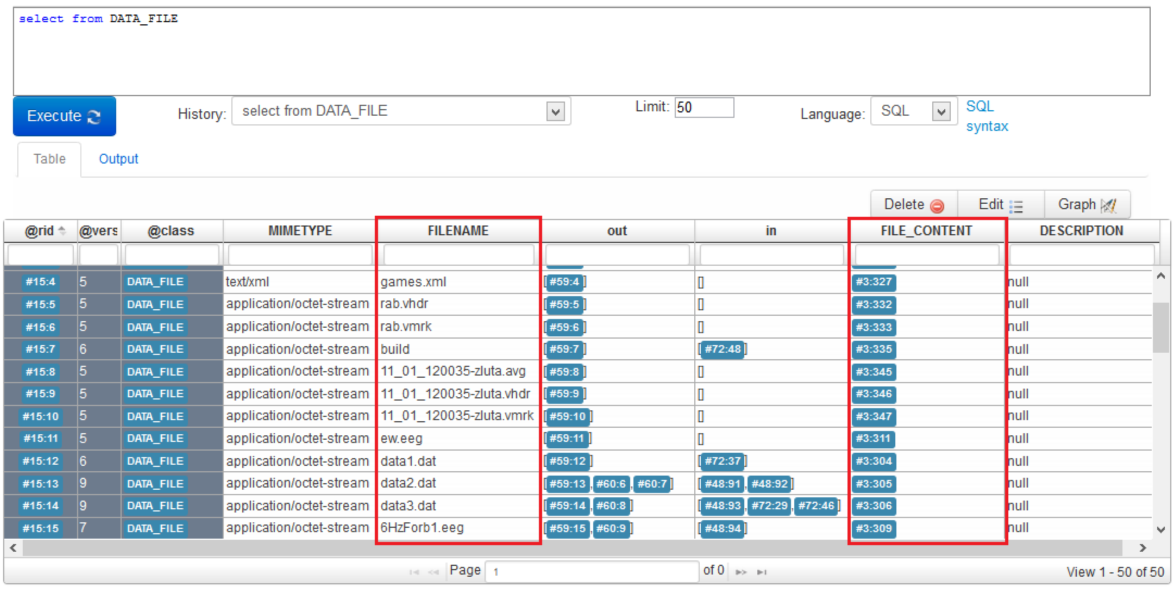
```
List<ODocument> result =  bgraph.getRawGraph()
.query(new OSQLSynchQuery<ODocument>
//execution of query to retrieve record to witch will be linked
//binary data
("SELECT FROM " +targetVertexName+"  WHERE " + IDcolumnName + "  =
" + numberOfID ));
```

3) binary data saving into appropriate record as a field

```
result.get(0).field(BLOBfieldName, BLOBrecord);

result.get(0).save();
```

On the Figure 3-7 we can see how binary files of different types are linked to appropriate records of type DATA_FILE (marked columns). It can be seen that file content is linked to the record in the similar way like edges – BLOB records have got own record ID too.



```
select from DATA_FILE
```

Execute | History: select from DATA_FILE | Limit: 50 | Language: SQL | SQL syntax

Table | Output

Delete | Edit | Graph

@rid	@vers	@class	MIMETYPE	FILENAME	out	in	FILE_CONTENT	DESCRIPTION
#15:4	5	DATA_FILE	text/xml	games.xml	[#59:4]	[]	#3:327	null
#15:5	5	DATA_FILE	application/octet-stream	rab.vhdr	[#59:5]	[]	#3:332	null
#15:6	5	DATA_FILE	application/octet-stream	rab.vmrk	[#59:6]	[]	#3:333	null
#15:7	6	DATA_FILE	application/octet-stream	build	[#59:7]	[#72:48]	#3:335	null
#15:8	5	DATA_FILE	application/octet-stream	11_01_120035-zluta.avg	[#59:8]	[]	#3:345	null
#15:9	5	DATA_FILE	application/octet-stream	11_01_120035-zluta.vhdr	[#59:9]	[]	#3:346	null
#15:10	5	DATA_FILE	application/octet-stream	11_01_120035-zluta.vmrk	[#59:10]	[]	#3:347	null
#15:11	5	DATA_FILE	application/octet-stream	ew.eeg	[#59:11]	[]	#3:311	null
#15:12	6	DATA_FILE	application/octet-stream	data1.dat	[#59:12]	[#72:37]	#3:304	null
#15:13	9	DATA_FILE	application/octet-stream	data2.dat	[#59:13, #60:6, #60:7]	[#48:91, #48:92]	#3:305	null
#15:14	9	DATA_FILE	application/octet-stream	data3.dat	[#59:14, #60:8]	[#48:93, #72:29, #72:46]	#3:306	null
#15:15	7	DATA_FILE	application/octet-stream	6HzForb1.eeg	[#59:15, #60:9]	[#48:94]	#3:309	null

Page 1 of 0 View 1 - 50 of 50

***Figure 3-7** We can see how binary files are linked to appropriate records in OrientDB Studio*

Note: *Text* is need to export from Oracle database's Character Large Objects (*CLOBs*) before storing them into OrientDB. Exported files with text can be stored into OrientDB in the same way like BLOBs. The only difference is that *String* class instead *ORecordBytes* class is used, e.g.:

```
String CLOBrecord =
new String(BytesReader.getBytesFromFile("fileName"));
```

All data from Oracle database are imported into OrientDB in this moment. The IDs properties for primary and foreing keys (remainder from relational database), which were used for edge import needs (see chapter 3.1.6.3), are now useless. They can be removed from vertices.

3.3.5 TYPES OF TESTED QUERIES AND COMMANDS

I tested OrientDB database on three following different groups of tasks:

1) single table queries

   These queries were executed on one type of table.

2) queries with JOIN operation

   These queries were executed over multiple tables.

3) standard commands INSERT, UPDATE and DELETE

3.3.5.1 INDEXES IN ORIENTDB

Index should be created on fields of vertices which are used in WHERE clause, ORDER BY clause, etc. to speed up queries. *UNIQUE, NOTUNIQUE, FULLTEXT or DICTIONARY* types of index can be put on the properties. Indexes can be *manual* or *automatic* (within the meaning of updating index). Automatic indexes are automatically updated by OrientDB engine and they are bound to *schema properties*. Manual indexes are handled by SQL commands. For my purpose I used automatic indexes.

There is a difference in indexes creation when we want to use *OrientDB SQL* queries and when we want to use *Gremlin* queries. For better understanding see example:

Index Creation/dropping for RESEARCH_GROUP.TITLE property in OrientDB SQL (console syntax):

1) schema property creation (see chapter 3.1.6.3)

   ```
   CREATE PROPERTY RESEARCH_GROUP.TITLE STRING
   ```

2) index on *TITLE* property creation

   ```
   CREATE INDEX RESEARCH_GROUP.TITLE NOTUNIQUE STRING
   ```

3) index on *TITLE* property dropping

   ```
   DROP INDEX RESEARCH_GROUP.TITLE
   ```

If we want to use *Blueprints Gremlin* language, *vertex types* like *RESEARCH_GROUP* cannot be used because Blueprints does not support them. Basic type of vertices – *OGraphVertex* must be used in this case (see chapter 3.1.6.1) and schema property does not need to be made. For better understanding see following example:

```
CREATE INDEX OGraphVertex.TITLE NOTUNIQUE STRING
```

This index type is put on all properties named *TITLE* in database (not only for one vertex type like *RESEARCH_GROUP*).

Note: I put indexes on properties in WHERE, ORDER BY, etc. clauses when I used OrientDB SQL. I put indexes on property of searched vertices when I used Gremlin. These indexed properties will be mentioned by properly queries. *UNIQUE* indexes are used on foreign keys in Oracle test database. All indexes used in OrientDB test database can be found in the Attachment C - Table C.2.

3.3.5.2 SINGLE TABLE QUERIES

I decided for this test because OrientDB SQL supports some constructions like *WHERE*, *ORDER BY, etc.* which are well known from standard SQL. For all queries on flat table I used OrientDB SQL language (see 2.4.4). All results of time measurement can be found in the Attachment D - Table D.1. Graph comparison of the averaged times for the different size of resultset can be found in the Attachment E. Queries are marked by number in brackets – [*query number*]. For the overall percentage score of these queries set of both databases see Graph 3-1 in this chapter.

OrientDB syntax is the same as Oracle SQL syntax for the same queries. Only difference is in restriction settings of returned results number, e.g.:

Oracle syntax: `SELECT * FROM EXPERIMENT WHERE ROWNUM <= 1000`

OrientDB syntax: `SELECT * FROM EXPERIMENT LIMIT = 1000`

1) simple query with selection of all columns/fields [1]

   Verbal description:

   Get all column values from experiment.

   Syntax:

   ```
   SELECT * FROM EXPERIMENT
   ```

OrientDB index:

There is no need for index.

2) simple query with *WHERE* clause[2]

Verbal description:

Get all records from experiment where temperature is bigger than 18.

Syntax:

```
SELECT * FROM EXPERIMENT WHERE TEMPERATURE > 18
```

OrientDB index:

Automatic index on property *TEMPERATURE (NOTUNIQUE, INTEGER)*

3) simple query with *WHERE* clause no. 2 [3]

Verbal description:

Get all persons with given name that starts with letter *A*.

Syntax:

```
SELECT * FROM PERSON WHERE GIVENNAME LIKE ‘A%’
```

OrientDB index:

Automatic index on property *GIVENNAME (NOTUNIQUE, STRING)*

4) simple query with *ORDER BY* clause [4]

Verbal description:

Get title and time from articles and order them by time in descending order.

Syntax:

```
SELECT TITLE, TIME FROM ARTICLES ORDER BY TIME DESC
```

OrientDB index:

Automatic index on property *TIME (NOTUNIQUE, DATE)*

5) simple query with *GROUP BY* clause [5]

Verbal description:

Get count for each electrode location and group it by title.

Syntax:

```
SELECT TITLE, COUNT(*) FROM ELECTRODE_LOCATION GROUP BY TITLE
```

OrientDB index:

Automatic index on property *TITLE (NOTUNIQUE, STRING)*

6) simple query with *BETWEEN* clause[6]

Verbal description:

Get impedance of electrode configuration where impedance is between 20 and 900.

Syntax:

```
SELECT * FROM ELECTRODE_CONF WHERE IMPEDANCE BETWEEN 20 AND 900
```

OrientDB index:

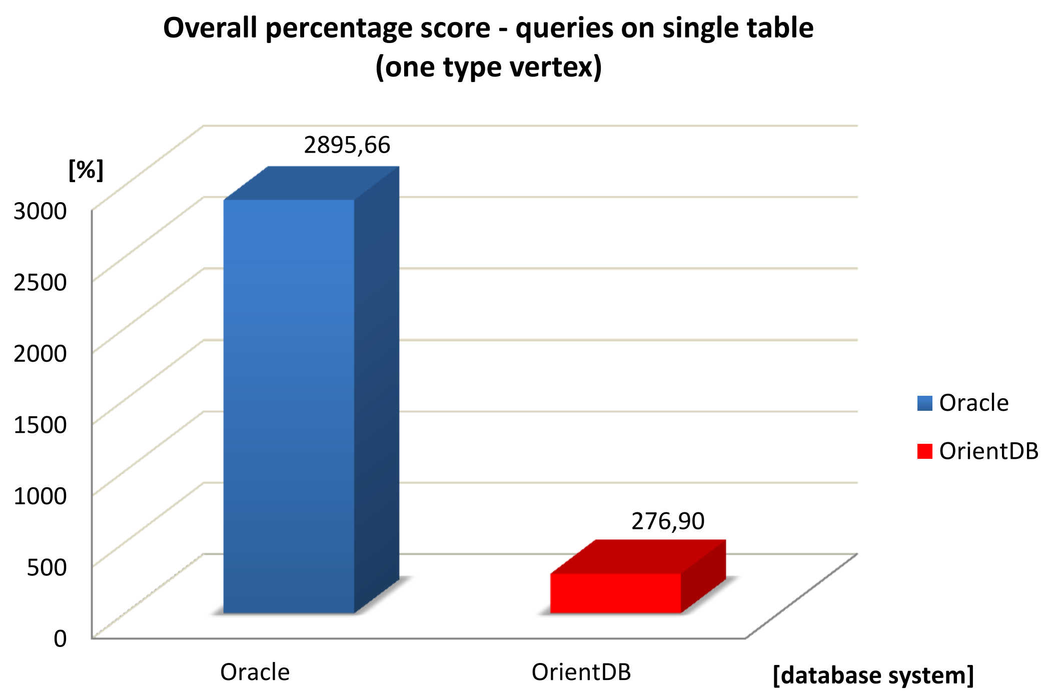
Automatic index on property *IMPEDANCE (NOTUNIQUE, INTEGER)*

I tested OrientDB SQL queries by using OrientDB Java API. I mentioned the code snippet for testing speed query below.

Code snippet for testing queries (synchronous query):

```
//list of returned records, records are returned as types of
//ODocument but they can also be returned as types of Vertex
List<ODocument> records= graph.getRawGraph().query(new
OSQLSynchQuery("SELECT * FROM EXPERIMENT LIMIT = 100000"));
```

Note: OrientDB also supports *asynchronous* queries that do not consumes Java heap. The time of query execution is similar.



Graph 3-1 *Percentage difference of both databases in queries on single table/vertices (higher is better)*

As we can see on Graph 3-1 OrientDB is much slower than Oracle in the overall perspective. However, it must be considered (according to the Attachment D - Table D.1) that query execution time depends on the number of returned records. Query execution time is nearly similar for smaller resultsets (about 1000 records). Oracle database clearly dominates for bigger resultsets (except one case with *ORDER BY* clause). Indexes on properties must be used for better performance by using OrientDB queries (otherwise, the time of query execution is a little slower). The biggest time difference is in query with *GROUP BY* clause when Oracle is up to 1 500% faster than OrientDB.

However, we must take into consideration that OrientDB and generally graph databases are not designed for these types of queries on a single type of vertex. It is logical that relational table structure have big advantage by these queries because table structure keeps well-structured data – table contains rows with records. Although one vertex is similar to row in relational model, vertices are stored in cluster which does not have so fixed structure like table therefore the retrieving records can be slower.

The results of this query set showed that if we have well-structured data without need for JOIN operations, Oracle database is the better choice than OrientDB.

3.3.5.3 QUERIES OVER MULTIPLE TABLES (JOINS)

I used TinkerPop Blueprints *Gremlin language* for all queries over multiple tables/vertices. I chose *Gremlin* thanks to its efficiency on property graph database model. Gremlin is designed especially for traversal over multiple vertices (see chapter 2.4.4.2).

All time measurement results can be found in the Attachment D - Table D.2. Graph comparison of the averaged times can be found in the Attachment E. Ten measurements were performed on one-size resultset for every query. Queries are marked by number in brackets – [*number query*]. For the overall percentage score of this queries set of both databases see Graph 3-2 in this chapter.

The syntax of Gremlin queries is *very different* from Oracle SQL syntax in this case**.** I provide Gremlin *console syntax* and *Java API syntax* for each query below.

The traversal over vertices in OrientDB database model is showed on Figure 3-8, each query traversal is distinguished by colour. Query path with specific colour starts in vertex with *START* clause and it ends in vertex with *END* clause*.* Basic Descriptions of *newly used* steps for OrientDB queries are provided by query syntax (query is executed like steps in pipe e.g. see chapter 2.4.4.2) in cells.

How it was mentioned, Gremlin queries in console version can be executed through OrientDB studio or through Gremlin console. We have to firstly connect to the database and create graph instance in Gremlin console, e.g.:

```
gremlin>g = new OrientGraph("remote:localhost/EEG_ERP");
==>orientgraph[remote:localhost/EEG_ERP]
```

Now *g* is the reference on the OrientDB graph.

1) query with 1x INNER JOIN clause [1] (for traversing over the graph see blue path on Figure 3-8)

   Verbal description:

      Get education level title of person with surname Walker.

Oracle SQL syntax:

```
SELECT TITLE FROM PERSON p
INNER JOIN EDUCATION_LEVEL e ON p.EDUCATION_LEVEL_ID =
e.EDUCATION_LEVEL_ID
WHERE p.SURNAME = 'Walker'
```

Gremlin console syntax:

g.V('SURNAME','Walker')	.out('REACHED')	.TITLE.as('title')	.table().cap()
Look for all vertices with surname property with value *Walker* → (Capital letter V means **all** vertices, lower case letter would mean **one** vertex, we would use vertex ID as a parameter.)	which are connected with vertices by outgoing edges with label *REACHED*→	and get *TITLE* of these outgoing vertices and set alias *title* properties in resultset →	and store values in the table. It emits the values of the previous step.

Gremlin Java API syntax:

```
//creation of new pipe
GremlinPipeline pipe = new GremlinPipeline();
//starting of pipe
pipe.start(graph.getVertices("SURNAME","Walker"))
.out("REACHED").property("TITLE")
.as("title").table().cap();
//save resultset into list
pipe.toList();
```

OrientDB indexes:

Automatic index on property *SURNAME (NOTUNIQUE, STRING)*

2) query with 4x INNER JOIN clause [2] (for traversing over the graph see red path on Figure 3-8)

Verbal description:

Get all reject conditions which belongs to persons which are members and owners of research group with name *DAV* from artefact. Get all usernames and title of group too.

Oracle SQL syntax:

```
SELECT TITLE, USERNAME, REJECT_CONDITION FROM
RESEARCH_GROUP r
INNER JOIN
```

```
  RESEARCH_GROUP_MEMBERSHIP m ON
(r.RESEARCH_GROUP_ID=m.RESEARCH_GROUP_ID)
INNER JOIN
  PERSON p ON (p.PERSON_ID = m.PERSON_ID)
INNER JOIN
  EXPERIMENT e ON (e.OWNER_ID= p.PERSON_ID)
INNER JOIN
  ARTEFACT t ON (t.ARTEFACT_ID = e.ARTEFACT_ID)
WHERE r.TITLE='DAV'
```

Gremlin console syntax:

```
g.V('TITLE','DAV').TITLE.as('title').back(1).out('IS_MEMBER')
```

Backtrack pattern – it means one step back (because we want to get TITLE from RESEARCH_GROUP, then we have to return before g.V(...) clause to process vertices connected to RESEARCH_GROUP by outgoing edges with label IS_MEMBER)

```
.USERNAME.as('username').back(1).in('MEASURED') .out('DEFINED')
```

Incoming edges with label MEASURED – we can traverse in reverse direction too. See Figure 3-8

```
.has('TYPE','ARTEFACT') .REJECT_CONDITION.as('reject_cond')
```

Filter method for searched vertices– as you can see on Figure 3-8 vertex of type EXPERIMENT has more outgoing edges of type DEFINED, so we need to distinguish (Gremlin does not support vertex types) them. This could be done by adding new property named TYPE with value ARTEFACT into vertices of type ARTEFACT.

```
.table().cap()
```

Gremlin Java API syntax:

```
GremlinPipeline pipe = new GremlinPipeline();
pipe.start(graph.getVertices("TITLE","DAV"))
.property("TITLE").as("title").back(1).out("IS_MEMBER")
.property("USERNAME").as("username").back(1).in("MEASURED")
.out("DEFINED").has("TYPE","ARTEFACT")
.property("REJECT_CONDITION").as("reject_cond");
```

OrientDB indexes:

Automatic index on property *TITLE (NOTUNIQUE, STRING)*

Automatic index on property *TYPE (NOTUNIQUE, STRING)*

3) query with SEMI RIGHT JOIN clause [3] (for traversing over the graph see orange path on Figure 3-8)

Verbal description:

Get all persons which own research group with name *Eagle.* Get username, e-mail and phone number of each person.

Oracle SQL syntax:

```
SELECT USERNAME, EMAIL, PHONE_NUMBER FROM PERSON
WHERE PERSON_ID IN
(SELECT OWNER_ID FROM RESEARCH_GROUP WHERE TITLE = 'Eagle')
```

Gremlin console syntax:

```
g.V('TITLE','Eagle').TITLE.as('t').back(1).out('OWNED_BY')
.transform
({[username:it.USERNAME,email:it.EMAIL,phone:it.PHONE_NUMBER]})
```

If we want to get **some (not all)** properties of some vertices located in the middle of pipe, we have to use *transform(..)* function. If we want to do the same thing but we will need get **all** properties, we will use function *map()*. We have to call *it* – iterator that iterates over all specific properties (syntax: alias: it.property_name) for each property in transform function.

```
.as('person_properties').table().cap()
```

Gremlin Java API syntax:

```
GremlinPipeline pipe = new GremlinPipeline();
pipe.start(graph.getVertices("TITLE","Eagle")).property("TITLE")
.as("t").back(1).out("OWNED_BY")
//we use inner classes to perform specific actions for some
//functions in Gremlin
.transform(new PipeFunction<OrientVertex, List<String>>(){
     @Override
//we can use list as collection for PERSON'S properties
//we have to not use iterator because pipe itself is iterable
     public List<String> compute(OrientVertex s) {
            List<String> properties = new ArrayList<String>();
            properties.add((String) s.getProperty("USERNAME"));
            properties.add((String) s.getProperty("EMAIL"));
            properties.add((String) s.getProperty("PHONE_NUMBER"));
            return properties;
     }
      }).as("person_properties").table().cap();
pipe.toList();
```

OrientDB indexes:

Automatic index on property *TITLE (NOTUNIQUE, STRING)*

4) query with 8x INNER JOIN clause [4] (for traversing over the graph see green path on Figure 3-8)

Verbal description:

Get all project type titles, electrode configuration impedances, research group titles, electrode location titles and electrode fixation titles for all project types with title *Supplies.*

Oracle SQL syntax:

```
SELECT pt.TITLE, ec.IMPEDANCE, r.TITLE, el.TITLE, ef.TITLE FROM
  PROJECT_TYPE pt
INNER JOIN
  PROJECT_TYPE_REL ptr ON
(ptr.PROJECT_TYPE_ID = pt.PROJECT_TYPE_ID)
INNER JOIN
  EXPERIMENT e ON
(e.EXPERIMENT_ID = ptr.EXPERIMENT_ID)
INNER JOIN
  ELECTRODE_CONF ec ON
(ec.ELECTRODE_CONF_ID = e.ELECTRODE_CONF_ID)
INNER JOIN
  ELECTRODE_LOCATION_REL elr ON
(elr.ELECTRODE_CONF_ID = ec.ELECTRODE_CONF_ID)
INNER JOIN
  ELECTRODE_LOCATION el ON
(el.ELECTRODE_LOCATION_ID = elr.ELECTRODE_LOCATION_ID)
INNER JOIN
  ELECTRODE_FIX ef ON
(el.ELECTRODE_FIX_ID = ef.ELECTRODE_FIX_ID)
INNER JOIN
  ELECTRODE_FIX_GROUP_REL efgr ON
(efgr.ELECTRODE_FIX_ID = ef.ELECTRODE_FIX_ID)
INNER JOIN
  RESEARCH_GROUP r ON
(r.RESEARCH_GROUP_ID = efgr.ELECTRODE_FIX_ID)
WHERE pt.TITLE = 'Supplies'
```

Gremlin console syntax:

```
g.V('TITLE','Supplies').TITLE.as('title_pt').back(1)
.out('HAS')
.out('DEFINED').IMPEDANCE.as('impedance').back(1)
.filter{it.TYPE=='ELECTRODE_CONF'}
```

Filter(..) method can be used for filtering of searched vertices Instead of *has(..)* method . Filter offers more possibilities how to filter records. Iterator *it* must be used in this case.

```
E.as('title_el').back(1)
.out('FIXED_BY').TITLE.as('title_ef').back(1)
.out('USED_BY').TITLE.as('title_r').table().cap()
```

Gremlin Java API syntax:

```
GremlinPipeline pipe = new GremlinPipeline();
pipe.start(graph.getVertices("TITLE","Supplies")).property("TITLE")
.as("title_pt").back(1).out("HAS").out("DEFINED")
.property("IMPEDANCE").as("impedance").back(1).
//we have to use appropriate inner class to perform filtering by
//using filter method
filter(new PipeFunction<OrientVertex, Boolean>() {
   @Override
   public Boolean compute(OrientVertex v) {
           return v.getProperty("TYPE").equals("ELECTRODE_CONF");
   }
}).out("CONFIGURED").has("TYPE","ELECTRODE_LOCATION")
.property("TITLE").as("title_el").back(1).out("FIXED_BY")
.property("TITLE").as("title_ef").back(1).out("USED_BY")
.property("TITLE").as("title_r").table().cap();
pipe.toList();
```

OrientDB indexes:

Automatic index on property *TITLE (NOTUNIQUE, STRING)*

Automatic index on property *TYPE (NOTUNIQUE, STRING)*

5) query with 3x INNER JOIN clause [5] (for traversing over the graph see yellow path on Figure 3-8)

Verbal description:

Get all names of scenarios for all persons with surname *Wilson* which co-operated on these scenarios.

Oracle SQL syntax:

```
SELECT p.USERNAME, s.SCENARIO_NAME
FROM SCENARIO s
INNER JOIN
  EXPERIMENT e ON (e.EXPERIMENT_ID = s.SCENARIO_ID)
INNER JOIN
  COEXPERIMENTER_REL c ON (c.EXPERIMENT_ID = e.EXPERIMENT_ID)
INNER JOIN
  PERSON p ON (p.PERSON_ID = c.PERSON_ID)
WHERE p.SURNAME = 'Wilson'
```

Gremlin Console syntax:

```
g.V('SURNAME','Wilson').USERNAME.as('username').back(1)

.out('COOPERATED').out('USED').has('TYPE','SCENARIO')

.property('SCENARIO_NAME').as('scenario_name').back(1)

.table().cap();
```

Gremlin Java API syntax:

```
GremlinPipeline pipe = new GremlinPipeline();
pipe.start(graph.getVertices("SURNAME","Wilson"))
.property("USERNAME").as("username").back(1).out("COOPERATED")
.out("USED").has("TYPE","SCENARIO")
.property("SCENARIO_NAME").as("scenario_name").back(1)
.table().cap();
pipe.toList();
```

OrientDB indexes:

Automatic index on property *SURNAME (NOTUNIQUE, STRING)*

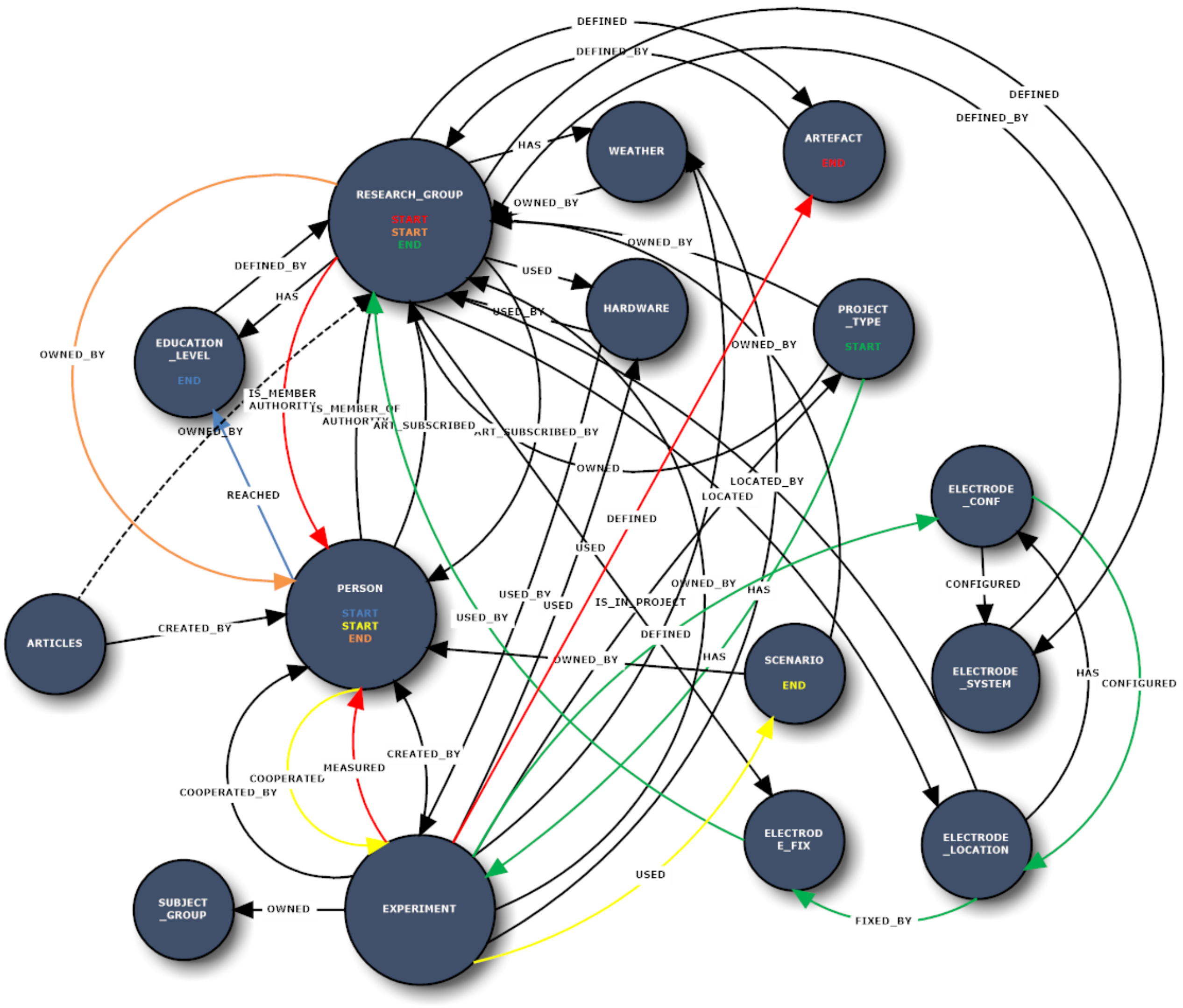
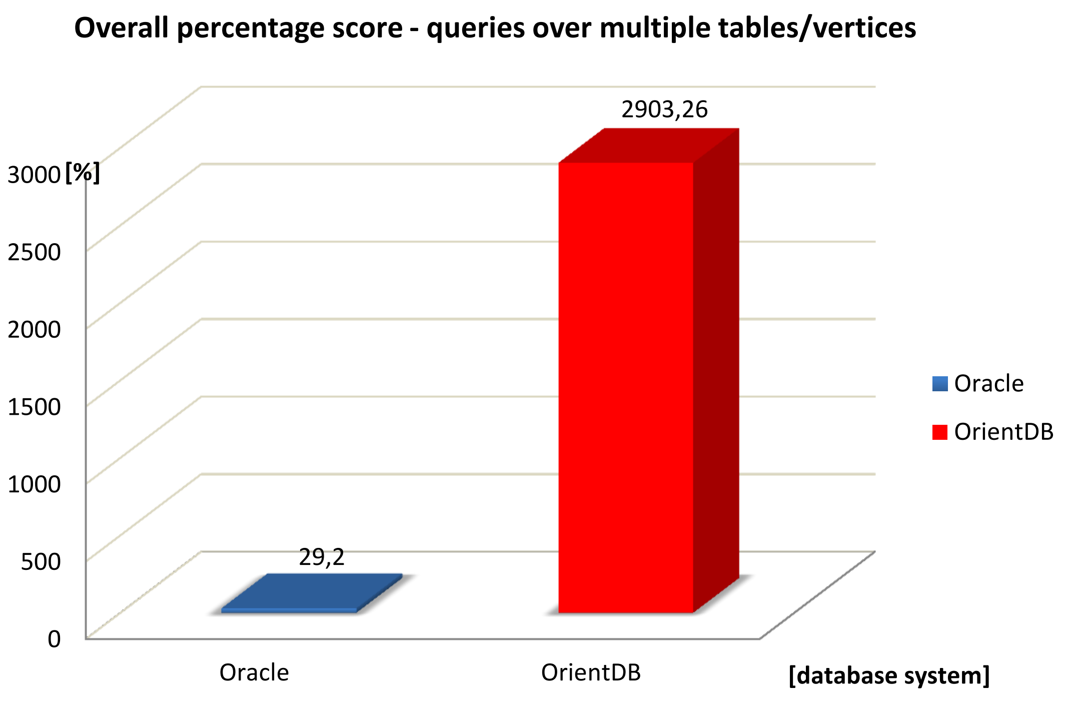


Figure 3-8 *Gremlin traversal of each query over multiple vertices in OrientDB database model*



Graph 3-2 *Percentage difference of both databases in queries on multiple types of tables/vertices*

As we can see on Graph 3-2 OrientDB dominates in queries over multiple types of vertices in overall perspective – see the Attachment D - Table D.2. OrientDB is slightly slower only in case with Oracle database SEMI JOIN clause – full join operation is not performed in this case (only sub-query) but this difference is insignificant. The assumptions from chapter 2.1.2.5 and chapter 2.2 were confirmed. The query speed strongly depends on the deep of JOIN operations. OrientDB graph database model shows the **best performance** on queries with **deep joins between** records - see the difference in query execution times when clause with 8x INNER JOIN is used - OrientDB is up to 2 100% faster than Oracle database. As it was mentioned in chapter 2.1.4.3, graph model is designed for big connection complexity among records. Moreover, graph database model brings much better options for data analysis. The data analysis is provided by tools for graph analysis or by graph query languages (**Gremlin)**. Gremlin is very efficient by **traversing** the graph – we can write the query with the same result in more ways. OrientDB graph model also provides good possibilities for semantics between records –you can see this fact by comparing of Oracle SQL and Gremlin query syntax. Gremlin queries have more natural ability to express query syntax than Oracle SQL and they are easier to read – thanks to labels on edges.

All results of time measurement show that for big connection complexity among records, when complex queries with deep JOIN operations are used, the OrientDB is better option than Oracle database. However, we have to use **indexes** for this good performance. The searching without indexes leads to the linear scanning of all vertices in dataset and it could lead to unacceptable degradation of the performance.

3.3.5.4 COMMANDS INSERT, UPDATE, DELETE

I used OrientDB SQL for commands UPDATE and DELETE. I used OrientDB Java API for command INSERT. All results of time measurement can be found in the Attachment D - Table D.3. Graph comparison of averaged times for the different number of processed records can be found in the Attachment E. For the overall percentage score of these command sets of both databases see Graph 3-3 in this chapter.

OrientDB syntax for commands UPDATE and DELETE is the same as Oracle SQL alternative. Only difference is by settings of restriction of number of returned results (see chapter 3.3.5.2). All commands were performed on table/vertices of type *SUBJECT_GROUP.*

1) insertion speed test – INSERT command

   Oracle SQL syntax:

```
INSERT INTO SUBJECT_GROUP (SUBJECT_GROUP_ID, TITLE, DESCRIPTION)
values (1, 'Title', 'This is a description')
...
```

   OrientDB Java API syntax:

   Massive insertion intent is desirable to be set before the records insertion, e.g.:

```
OrientBatchGraph bgraph = null;
bgraph = new OrientBatchGraph(graph.getRawGraph());
bgraph.getRawGraph().declareIntent(new OIntentMassiveInsert());
```

Records insertion:

```
ODocument doc = bgraph.getRawGraph().createVertex();
//iteration over number of chosen records to store
for (int i = 0; i < NUMBER_OF_RECORDS; i++)
{
//resets the record to be reused, it recycles records avoiding the
//creation of them stressing the JVM Garbage
    doc.reset();
//class name for insertion
    doc.setClassName("SUBJECT_GROUP");
//fields(properties) with values
    doc.field("SUBJECT_GROUP_ID", i);
    doc.field("TITLE", "Title");
    doc.field("DESCRIPTION", "This is a description");
//save record into database
    doc.save();
}
```

2) test of updating speed – UPDATE command

Syntax:

```
UPDATE SUBJECT_GROUP SET TITLE = 'This is a new title'
```

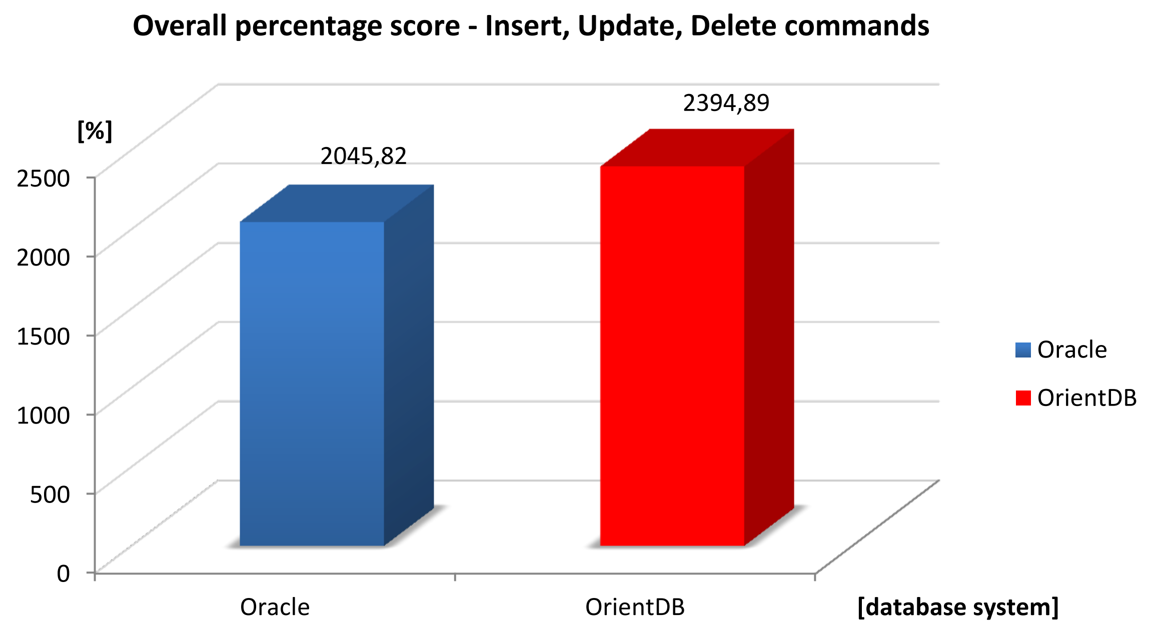
3) updating speed test – DELETE command

Syntax:

```
DELETE FROM SUBJECT_GROUP
```

I tested UPDATE and DELETE commands by using OrientDB Java API. Below, I mentioned the code snippet for testing query speed. Code snippet for testing commands:

```
graph.getRawGraph().command(new OCommandSQL("UPDATE SUBJECT_GROUP
SET TITLE = 'This is a new title' limit = 220000")).execute();
```



Graph 3-3 *Percentage difference of both databases in standard database commands – INSERT, UPDATE, DELETE*

OrientDB is in overall perspective slightly better as we can see on Graph 3-3. We must considerate (according to the Attachment D - Table D.3) that command execution times strongly depend on the number of processed records. OrientDB is better than Oracle database for a small resultset sizes (about 1000 records) in all cases. OrientDB especially dominates in any number of records *insertion* when it is up to 1150% faster than Oracle database with 220 000 records . On the contrary, Oracle database is much faster for bigger number of processed records in operations UPDATE and DELETE (up to 846% by f 100 000 records deleting).

All results of time measurement show that OrientDB offers superfast records insertion. Insertion can be much faster (about 1 000 000 inserted records in less than 20 seconds) if OrientDB uses local protocol (see chapter 2.4.1.1). OrientDB is faster on smaller number of processed records in all cases. Oracle database is better in case of UPDATE and DELETE commands for bigger number of processed records. The performance degradation by deleting/updating records is logical because graph model is much more complex than relational table structure and the linear scanning must look up for all records in main memory. The performance could be probably enhanced by putting indexes on some deleted/updated vertices sub-set.

3.3.6 ORIENTDB GRAPH MODEL SUITABILITY FOR EEG/ERP DOMAIN

I mention here following insights which are evaluation of working with OrientDB. These insights are based on my own experience and results from chapters 3.3.5.1 – 3.3.5.4.

1) **working with the database**

   Firstly I tried to test the database on 32bit architecture with 32bit JVM and I was not able to tune database to work without errors (see chapter 3.1.6.4) with bigger dataset (GBs). It was caused by small amount of RAM (Random Access Memory) for Java process and by bad memory mapping of OrientDB on 32bit architecture. Problems were solved when I switched to the 64bit architecture. The absence of good database client is the biggest lack of working with OrientDB (unlike robust Oracle SQL Developer). It is true that OrientDB offers *Console* and *OrientDB Studio* clients. However, the console has limited possibilities of result formatting and it has not fully support of Gremlin language (Gremlin can be partially combined with OrientDB SQL). *Gremlin Console* has to be used for pure Gremlin queries. Moreover, there are minimal options for additional results processing from consoles. OrientDB studio is web client which offers more comfort by working with the database, but if we want to execute query which returns big dataset (thousands of records) then the web browser freezes and it is impossible to work with it. On the Other hand, OrientDB Studio offers good options for the database model visualization and it supports Gremlin and SQL query languages. I prefer native Java API as the best way to work with the database. I think Java API is the best OrientDB advantage – it means no third party drivers (like JDBC) and no ORM (like Hibernate). Only Java API disadvantage may be the need for advanced experiences with Java programing language for some users.

2) **database model**

   EEG/ERP portal application is not the typical case for graph database model (unlike social network, transportation network, etc.). However, portal complexity is every year more complicated and the requirement to use more complicated queries and relationships between records is increased. More complex relationships between records can be used with graph database model. The semantics possibilities are good assumptions for f semantic web application creation and the queries can be more natural and much more readable. The whole EEG/ERP portal database model would be greatly simplified (see the chapter 3.1) with OrientDB usage from my perspective and it could be more transparent and flexible. There would be no need for cumbersome ORM thanks to native Java API.

Only problem could be missing support for Spring Framework. EEG/ERP portal uses Spring Framework for application layer. Currently, OrientDB has not direct support for Spring Framework and the OrientDB configuration for Spring Framework usage could be more complicated. However, there are some projects in development which will make easier to build Spring-powered applications with OrientDB, e.g. *Spring Data OrientDB*[1].

3) query possibilities and query speed

In addition to what were mentioned in chapters 3.3.5.1 – 3.3.5.4 I think this is the biggest OrientDB advantage. OrientDB supports own SQL implementation which can be combined with Gremlin language or just Gremlin language can be used. OrientDB offers really good query possibilities (Oracle database supports “only” standard SQL). An extensive data analysis can be done with Gremlin language power (also thanks to good semantic expression ability of relationships between records). We are not only restricted on queries over vertices but we can also query edges. Moreover, external tools for graph analysis like *Gephi*[2] tool can be used with OrientDB. OrientDB dominates when we mainly need complex queries with deep JOIN operations. Oracle database dominates when we mainly need simple queries on single table with well-structured data. EEG/ERP portal does not use queries with very deep JOIN operations. Querying the EEG/ERP portal with current HQL queries is suitable but OrientDB SQL could fully replace these current queries and Gremlin powerful queries could be used with increasing EEG/ERP portal database model complexity in the future. The number of returned records from EEG/ERP portal queries is not so big and for these smaller datasets OrientDB is faster than Oracle database or as fast as Oracle database in all cases (see the Attachment D). Moreover, The OrientDB queries are more readable than HQL from my viewpoint. I can recommend OrientDB for querying EEG/ERP portal on the base of previous information.

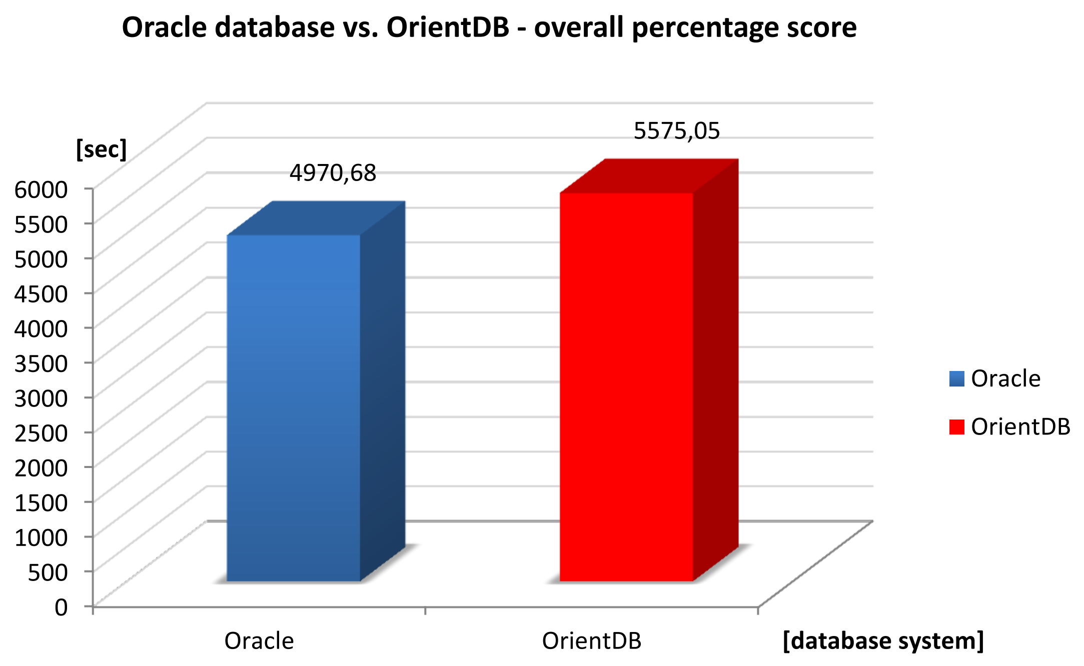
[1] Spring data project make it easier to build Spring-powered applications. Spring data project will implement

[2] Gephi is an interactive visualization platform for networks and dynamic and hierarchical graphs. It supports all Blueprints graph implementations (including OrientDB or Neo4J) [36]

4) features

OrientDB supports ACID transactions, Locks, Hooks (triggers), database import and export etc. See the Attachment G for main OrientDB features.

Finally, see Graph 3-4 which shows the overall percentage score of both tested databases. This score is composed from all sub-measurements., Databases are not so different in their performance from the overall view, but the concrete use case has to be considered. Current relational EEG/ERP portal database is suitable for querying the database on the current level (lower connectivity between records and queries with smaller JOINs depth). However, OrientDB offers more simple and more flexible database model with own Java API and without ORM. I think that the change of EEG/ERP portal database to OrientDB would be useful.



Graph 3-4 *Overall percentage score of both tested database systems*

4 CONCLUSION

This thesis was created on the base of EEG/ERP research group requirement. The main goal of my work was to choose appropriate NoSQL database system which could improve and replace current Oracle relational database system used in EEG/ERP portal application database layer. As the most suitable OrientDB document/graph NoSQL system was chosen (selection is described in [23]). The OrientDB suitability for EEG/ERP domain was evaluated on the base of performance testing both database systems and on the base of working convenience with both database systems.

I provided comparison of relational database system and NoSQL database system in the theoretical part. I mentioned advantages and disadvantages of these both database approaches. In the last part of the theoretical part the main NoSQL database models were described. Finally, I acquainted with OrientDB graph model concepts which were the main assumptions for the practical part.

I focused on the realization of database systems testing in the practical part of this work. The first phase of this part was focused on the customization and preparation of current EEG/ERP portal database. I built graph database model from this customized relational database model in OrientDB. The main customization content was to decompose the relational tables and relationships on graph vertices and edges.

I chose the sub-model from the original EEG/ERP portal relational database for testing purposes in the next phase. I build this sub-model in OrientDB graph database model too. Finally, both databases contained the same data and relationships.

The test part followed. The same configuration was used for both databases. Both databases were tested on the same set of database queries and database commands. I performed ten measurements for each database query and five measurements for each database command. I tested databases on different numbers of processed records. Each set of measured query and command was averaged. I made the speed comparison of both databases on the base of percentage difference between averages of measured sets.

The testing results confirmed basic assumptions. Relational database can better handle with queries on single table. Relational table takes advantage of its fixed structure in this case. In contrast to relational database, NoSQL graph database can better handle with queries over multiple vertices. NoSQL database takes advantage of its robust graph database model which is able to manage relations among vertices like direct links – queries are performed without demanding JOIN operations. Test of database commands showed that NoSQL dominates in INSERT command but it is worse in DELETE and UPDATE commands than relational database. Query and command execution speed depends on number of processed records in all cases. The testing results showed that it depends on types of used queries and commands and from overall view both databases are similar in this case.

I also tested NoSQL database handling with binary files. Binary files were stored efficiently without taking up additional disk space.

In the overall perspective I cannot say which database system is better or faster. From my point of view, EEG/ERP portal’s database model is not the typical case for graph database model but I believe that OrientDB could bring certain advantages for EEG/ERP portal application. OrientDB could especially simplify the whole database model. The database model would be more flexible in terms of variability. I think that avoiding of ORM and third party drivers could be also the significant advantage. OrientDB solution offers more natural syntax of queries too. However, OrientDB does not offer so good working convenience concerning to database clients like Oracle database. OrientDB’s tools for database managing need additional improvement. It is because of OrientDB is open source project and it is still in the development process. Currently, full tutorials are not also available yet but the community around OrientDB project is still bigger.

If we can accept all mentioned restrictions, OrientDB could be very useful for EEG/ERP domain in terms of future development of EEG/ERP portal application.

5 LIST OF ABBREVIATIONS

Table 5-1 *Alphabetical list of abbreviations*

Abbreviation	Explanation
ACID	Atomicity, Consistency, Isolation, Durability
AJAX	Asynchronous Javascript and XML
API	Application Program Interface
BLOB	Binary Large Object
CLOB	Character Large Object
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
E-R	Entity Relationship
EEG	Electroencephalography
ERP	Even-Related Potentials
HDD	Hard Disk Drive
HQL	Hibernate Query Language
HTTP	HyperText Transfer Protocol
Java EE	Java Enterprise Edition
JDBC	Java Data Base Connectivity
JMX	Java Management Extension
JSON	JavaScript Object Notation
JSP	Java Server Pages
JVM	Java Virtual Machine
NoSQL	Not Only Structured Query Language
oda	OrientDB data
odh	OrientDB data holes
OOP	Object Oriented Programming
ORM	Object Relational Mapping
PL/SQL	Procedural Language/Structured Query Language
POJO	Plain Old Java Object
RDBMS	Relational Database Management System
RDF	Resource Description Framework
RAM	Random Access Memory
REST	REpresentational State Transfer
TCP/IP	Transmission Control Protocol/Internet Protocol
XML	Extensible Markup Language

6 BIBLIOGRAPHY

[1] A. Juozapavičius: *Introducing the database.* Vilnius University, Faculty of Mathematics and Informatics [online], last visited 1.4. 2013. Available from: <http://mif.vu.lt/cs2/en/courses/infsyst/files/infos2.pdf>

[2] Ben Stopford: *Thoughts on Big Data Technologies (1) (20 Jun, 2012).* www.BenStopfort.com [online], last visited 29.4. 2013. Available from: <http://www.benstopford.com/2012/06/30/thoughts-on-big-data-technologies-part-1/>

[3] Diana Lorentz: *Oracle Database SQL reference, 10g Release 2 (10.2).* Oracle [online], last visited 1.4. 2013. Available from: <http://docs.oracle.com/cd/B19306_01/server.102/b14200.pdf>

[4] *Structure-Related Terms.* eTutorials.org [online], last visited 1.4. 2013. Available from: <http://etutorials.org/SQL/Database+design+for+mere+mortals/Part+I+Relational+Database+Design/Chapter+3.+Terminology/Structure-Related+Terms/>

[5] Daniel Bartholomew: *SQL vs. NoSQL (01 Sep, 2010)*. Linux Journal [online], last visited 1.4. 2013. Available from: <http://www.linuxjournal.com/article/10770?page=0,0>

[6] *NOSQL*. NOSQL databases [online], last visited 1.4. 2013.
Available from: <http://nosql-database.org/>

[7] Tim Perdue: *NoSQL: An Overview of NoSQL Databases*. About.com New Tech [online], last visited 1.4. 2013 Available from: <http://newtech.about.com/od/databasemanagement/a/Nosql.htm>

[8] *Why NoSQL?* Couchbase [online], last visited 1.4. 2013. Available from: <http://www.couchbase.com/why-nosql/nosql-database>

[9] Martin Brown: *Document databases in predictive modelling (08 Oct, 2012).* IBM [online], last visited 1.4. 2013. Available from: <http://www.ibm.com/developerworks/library/ba-docdbpmml/index.html>

[10] Ronald Bourret: *Use cases for native XML databases*. Roland Bourret [online], last visited 1.4. 2013. Available from: <http://www.rpbourret.com/xml/UseCases.htm>

[11] Paul Williams: *The NoSQL Movement: Key-Value Databases (30 Oct, 2012)*. Dataversity [online], last visited 1.4. 2013. Available from: <http://www.dataversity.net/the-nosql-movement-key-value-databases/>

[12] *What is a NoSQL Key-Value Store?* Aerospike [online], last visited 1.4.2013.
Available from: <http://www.aerospike.com/what-is-a-nosql-key-value-store/>

[13] Renzo Angels, Claudio Guituerrez: *Survey of Graph Database Models*. Computer Science Department, Universidad de Chile

[14] *What is a Graph database?* Neo4J [online], last visited 1.4.2013,
Available from: <http://www.neo4j.org/learn/graphdatabase>

[15] *Property Graph Model (2012).* Github – TinkerPop/Blueprints [online], last visited 1.4.2013. Available from: <https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model>

[16] Michael Domenjoud, Thomas Vlal: *Graph databases: an overview (12 Jul, 2012).* Octo talks [online], last visited 1.4.2013.
Available from: <http://blog.octo.com/en/graph-databases-an-overview/>

[17] *35+ Use Cases For Choosing Your Next NoSQL Database (20 Jun, 2011).* High Scalability [online], last visited 1.4.2013. Available from:
<http://highscalability.com/blog/2011/6/20/35-use-cases-for-choosing-your-next-nosql-database.html>

[18] Abel Avram: *Transitioning from RDBMS to NoSQL. Interview with Couchbases's Dipti Bokar (8 Sep, 2012).* Infoq [online], last visited 29.4. 2013. Available from:
<http://www.infoq.com/articles/Transition-RDBMS-NoSQL>

[19] Avishkar Meshram: *NoSQL Key-Value store (20 Feb, 2013).* Business intelligence [online], last visited 1.4.2013.
Available from: <http://avishkarm.blogspot.cz/>

[20] Marko Rodriguez: *Pipes: The Data Flow Framework for Gremlin – GraphDB Traversal (02 Aug, 2012)*. DZone [online], last visited 27.4.2013. Available from:
<http://architects.dzone.com/articles/nature-pipes>

[21] Ankit Mathur: *Up close and Personal with NoSQL (01 Feb, 2011).* Linux for you [online], last visited 1.4.2013.
Available from: <http://www.linuxforu.com/2011/02/up-close-and-personal-with-nosql/>

[22] Michael Kopp: *NoSQL or RDBMS – Are we asking the right questions? (05 Oct, 2012)* Compuware [online], last visited 1.4.2013. Available from:
<http://apmblog.compuware.com/2011/10/05/nosql-or-rdbms-are-we-asking-the-right-questions/>

[23] Mikayel Valdanyan: *Picking the Right NoSQL Database Tool (22 May, 2011).* Monitis [online], last visited 1.4.2013. Available from:
<http://blog.monitis.com/index.php/2011/05/22/picking-the-right-nosql-database-tool/>

[24] Ladislav Janák: *No-SQL databases in EEG/ERP domain – Thematic project (Pilsen, 2013).* University of West Bohemia, Faculty of applied sciences, Department of computer science and Engineering.

[25] OrientDB Wiki pages: *Concepts (24 Oct, 2012)*. OrientDB Google project [online], last visited 27.4.2013. Available from: <http://code.google.com/p/orient/wiki/Concepts>

[26] Nuvolabase/orientdb: *Tutorial: Clusters (29 Feb, 2013).* GitHub [online], last visited 29.4.2013. Available from: <https://github.com/nuvolabase/orientdb/wiki/Tutorial:-Clusters>

[27] OrientDB Wiki pages: *Java APIs (08 Sep, 2012)*. OrientDB Google project [online], last visited 27.4.2013. Available from: <http://code.google.com/p/orient/wiki/JavaAPI>

[28] TinkerPop Wiki pages: *Blueprints (26 April, 2013)*. Github [online], last visited 27.4.2013. Available from: <https://github.com/tinkerpop/blueprints/wiki>

[29] OrientDB Wiki pages: *SQL (08 Sep, 2012)*. OrientDB Google project [online], last visited 27.4.2013. Available from: <http://code.google.com/p/orient/wiki/SQL>

[30] TinkerPop Wiki pages: *Gremlin (27 March, 2013)*. Github [online], last visited 27.4.2013. Available from: <https://github.com/tinkerpop/gremlin/wiki>

[31] OrientDB Wiki pages: *BinaryData (08 Sep, 2012)*. OrientDB Google project [online], last visited 27.4.2013. Available from: <http://code.google.com/p/orient/wiki/BinaryData>

[32] OrientDB Wiki pages: *DBserver (28 Nov, 2012)*. OrientDB Google project [online], last visited 27.4.2013. Available from: <http://code.google.com/p/orient/wiki/DBServer>

[33] OrientDB Wiki pages: *ConsoleCommands (03 Oct, 2012)*. OrientDB Google project [online], last visited 27.4.2013. Available from: <http://code.google.com/p/orient/wiki/ConsoleCommands>

[34] OrientDB Wiki pages: *OrientDB_Studio (08 Sep, 2012)*. OrientDB Google project [online], last visited 27.4.2013. Available from: <http://code.google.com/p/orient/wiki/OrientDB_Studio>

[35] GitHub: *nuvolabase/spring-data-orientdb.* GitHub [online], last visited 12.5.2013. Available from: <https://github.com/nuvolabase/spring-data-orientdb>

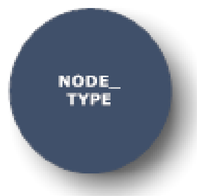
[36] Gephi [online], last visited 13.5.2013. Available from: <https://gephi.org/>

7 ATTACHMENTS

ATTACHMENT A

Semantics between all nodes of OrientDB EEG/ERP graph model

Table A.1 *Semantics between vertices*

Node type (connected by *out* edge)	Edge label [other property]	Target node (connected by *in* edge)
	EDGE LABEL [OTHER PROPERTY]	TARGET NODE
ANALYSIS	MADE_BY	RESEARCH_GROUP
ARTEFACT	DEFINED_BY	RESEARCH_GROUP
ARTEFACT_REMOVING_METHOD	USED_BY	EXPERIMENT
	USED_BY	RESEARCH_GROUP
ARTICLES	CREATED_BY	PERSON
	OWNED_BY	RESEARCH_GROUP
	SUBSCRIBED_BY	PERSON
ARTICLES_COMMENTS	COMMENTED	ARTICLES
	COMMENTED_BY	ARTICLES_COMMENTS
	COMMENTED_BY	PERSON
DATA_FILE	PERFORMED	ANALYSIS
	HAS	EXPERIMENT
	DEFINED [METADATA_VALUE]	FILE_METADATA_PARAM_DEF
DIGITIZATION	OWNED_BY	RESEARCH_GROUP
DISEASE	OWNED_BY	RESEARCH_GROUP
	OWNED_BY	EXPERIMENT
EDUCATION_LEVEL	DEFINED_BY	RESEARCH_GROUP
ELECTRODE_CONF	HAS	DATA_FILE
	CONFIGURED	ELECTRODE_SYSTEM

	CONFIGURED	ELECTRODE_LOCATION
ELECTRODE_FIX	USED_BY	RESEARCH_GROUP
ELECTRODE_LOCATION	FIXED_BY	ELECTRODE_FIX
	IS_TYPE	ELECTRODE_TYPE
	HAS	ELECTRODE_CONF
	LOCATED_BY	RESEARCH_GROUP
ELECTRODE_SYSTEM	DEFINED_BY	RESEARCH_GROUP
ELECTRODE_TYPE	USED_BY	RESEARCH_GROUP
EXPERIMENT	DEFINED	ARTEFACT
	OWNED	DIGITIZATION
	DEFINED	ELECTRODE_CONF
	CREATED_BY	PERSON
	MEASURED	PERSON
	OWNED_BY	RESEARCH_GROUP
	USED	SCENARIO
	OWNED	SUBJECT_GROUP
	HAS	WEATHER
	USED	ARTERFACT_REMOVING_METHOD
	COOPERATED_BY	PERSON
	HAS	DISEASE
	USED	HARDWARE
	HAS	PHARMACEUTICAL
	IS_IN_PROJECT	PROJECT_TYPE
	USED	SOFTWARE
	DEFINED [PARAM_VALUE]	EXPERIMENT_OPT_PARAM_DEF
EXPERIMENT_OPT_PARAM_DEF	DEFINED_BY	RESEARCH_GROUP
	DEFINED_BY [PARAM_VALUE]	EXPERIMENT
FILE_METADATA_PARAM_DEF	DEFINED_BY	RESEARCH_GROUP
	DEFINED_BY [METADATA_VALUE]	DATA_FILE
GROUP_PERMISSION_REQUEST	REQUESTED_BY	PERSON

	REQUESTED_BY	RESEARCH_GROUP
HARDWARE	USED_BY	RESEARCH_GROUP
	USED_BY	EXPERIMENT
HISTORY	BELONGS_TO	DATA_FILE
	BELONGS_TO	EXPERIMENT
	BELONGS_TO	PERSON
	BELONGS_TO	SCENARIO
KEYWORDS	DEFINED_BY	RESEARCH_GROUP
PERSON	REACHED	EDUCATION_LEVEL
	ART_SUBSCRIBED	RESEARCH_GROUP
	IS_MEMBER_OF [AUTHORITY]	RESEARCH_GROUP
	SUBSCRIBED	ARTICLES
	COOPERATED	EXPERIMENT
	DEFINED_BY [PARAM_VALUE]	PERSON_OPT_PARAM_DEF
	OWNED	RESEARCH_GROUP
PERSON_OPT_PARAM_DEF	DEFINED_BY	RESEARCH_GROUP
	HAS [PARAM_VALUE]	PERSON
PHARMACEUTICAL	USED_BY	RESEARCH_GROUP
	USED_BY	EXPERIMENT
PROJECT_TYPE	HAS	EXPERIMENT
	OWNED_BY	RESEARCH_GROUP
RESEARCH_GROUP	IS_MEMBER [AUTHORITY]	PERSON
	MADE	ANALYSIS
	USED	ARTEFACT_REMOVING_METHOD
	ART_SUBSCRIBED_BY	PERSON
	HAS	DIGITIZATION
	HAS	DISEASE
	HAS	EDUCATION_LEVEL
	DEFINED	EXPERIMENT_OPT_PARAM_DEF

	DEFINED	FILE_METADATA_PARAM_DEF
	USED	HARDWARE
	DEFINED	PERSON_OPT_PARAM_DEF
	HAS	PHARMACEUTICAL
	USED_BY	SOFTWARE
	HAS	WEATHER
	USED	STIMULUS_TYPE
	OWNED_BY	PERSON
	DEFINED	ARTEFACT
	FIXED	ELECTRODE_FIX
	LOCALTED	ELECTRODE_LOCATION
	DEFINED	ELECTRODE_SYSTEM
	USED	ELECTRODE_TYPE
	OWNED	PROJECT_TYPE
RESERVATION	RESERVED_BY	PERSON
	RESERVED_BY	RESEARCH_GROUP
SCENARIO	OWNED_BY	PERSON
	OWNED_BY	RESEARCH_GROUP
	USED	STIMULUS
	USED	STIMULUS_TYPE
SCENARIO_TYPE_NONXML	DEFINED_BY	SCENARIO
SERVICE_RESULT	BELONGS_TO	PERSON
SOFTWARE	USED_BY	RESEARCH_GROUP
	USED_BY	EXPERIMENT
STIMULUS	USED_BY	SCENARIO
	DEFINED_BY	STIMULUS_TYPE
STIMULUS_TYPE	USED_BY	SCENARIO
	DEFINED	STIMULUS
	USED_BY	RESEARCH_GROUP
WEATHER	OWNED_BY	RESEARCH_GROUP

ATTACHMENT B

EEG/ERP portal test model in OrientDB and the equivalent EEG/ERP portal test model in Oracle database

RESEARCH_GROUP
PERSON
EXPERIMENT
KEYWORDS
USER CONNECTION
PERSON_OPT_PARAM_DEF
EDUCATION_LEVEL
SERVICE_RESULT
RESERVATION
ARTICLE_COMMENTS
ARTICLES
GROUP_PERMISION_REQ
ARTEFACT_REMOVING_METHOD
WEATHER
HARDWARE
PROJECT_TYPE
HISTORY
DISEASE
ARTEFACT
DIGITIZATION
ANALYSIS
FILE_METADATA_PARAM_DEF
STIMULUS_TYPE
DATA_FILE
STIMULUS
SCENARIO
PHARMACEUTICAL
ELECTRODE_CONF
ELECTRODE_TYPE
SCENARIO_TYPE_NONXML
SCENARIO_SCHEMA
SOFTWARE
EXPERIMENT_OPT_PARAM_DEF
SUBJECT_GROUP
ELECTRODE_FIX
ELECTRODE_LOCATION
ELECTRODE_SYSTEM

Figure B.1 *EEG/ERP portal model in OrientDB graph database based on the Attachment A - Table 1 (red sub-model marks OrientDB test database model)*

SYSMAN.RESEARCH_GROUP
P * RESEARCH_GROUP_ID NUMBER
F * OWNER_ID NUMBER
* TITLE VARCHAR2 (100 BYTE)
* DESCRIPTION VARCHAR2 (250 BYTE)
RESEARCH_GROUP_PK (RESEARCH_GROUP_ID)
SYS_C0010810 (RESEARCH_GROUP_ID)
SYSMAN.ARTICLES
P * ARTICLE_ID NUMBER
F * PERSON_ID NUMBER
F RESEARCH_GROUP_ID NUMBER
TITLE VARCHAR2 (150 BYTE)
* TIME DATE
ARTICLES_PK (ARTICLE_ID)
ARTICLES_PK (ARTICLE_ID)
SYSMAN.ARTICLES_GROUP_SUBSCRIBTIONS
PF* PERSON_ID NUMBER
PF* RESEARCH_GROUP_ID NUMBER
ARTICLES_GROUP_SUBSCRIBTI_PK (PERSON_ID, RESEARCH_GROUP_ID)
ARTICLES_GROUP_SUBSCRIBTI_PK (PERSON_ID, RESEARCH_GROUP_ID)
SYSMAN.EDUCATION_LEVEL_GROUP_REL
PF* RESEARCH_GROUP_ID NUMBER
PF* EDUCATION_LEVEL_ID NUMBER
EDUCATION_LEVEL_GROUP_REL_PK (EDUCATION_LEVEL_ID, RESEARCH_GROUP_ID)
SYS_C0010899 (EDUCATION_LEVEL_ID, RESEARCH_GROUP_ID)
SYSMAN.EDUCATION_LEVEL
P * EDUCATION_LEVEL_ID NUMBER
* TITLE VARCHAR2 (150 BYTE)
* IS_DEFAULT NUMBER
EDUCATION_LEVEL_PK (EDUCATION_LEVEL_ID)
SYS_C0010802 (EDUCATION_LEVEL_ID)
SYSMAN.ELECTRODE_FIX_GROUP_REL
PF* RESEARCH_GROUP_ID NUMBER
PF* ELECTRODE_FIX_ID NUMBER
ELECTRODE_FIX_GROUP_REL_PK (ELECTRODE_FIX_ID, RESEARCH_GROUP_ID)
SYS_C0010924 (ELECTRODE_FIX_ID, RESEARCH_GROUP_ID)
SYSMAN.ELECTRODE_FIX
P * ELECTRODE_FIX_ID NUMBER
* TITLE VARCHAR2 (150 BYTE)
* DESCRIPTION VARCHAR2 (300 BYTE)
* IS_DEFAULT NUMBER
ELECTRODE_FIX_PK (ELECTRODE_FIX_ID)
SYS_C0010867 (ELECTRODE_FIX_ID)
SYSMAN.ELECTRODE_LOCATION_GROUP_REL
PF* RESEARCH_GROUP_ID NUMBER
PF* ELECTRODE_LOCATION_ID NUMBER
ELECTRODE_LOCATION_GROUP_REL_PK (RESEARCH_GROUP_ID, ELECTRODE_LOCATION_ID)
SYS_C0010927 (RESEARCH_GROUP_ID, ELECTRODE_LOCATION_ID)
SYSMAN.HARDWARE_USAGE_REL
PF* HARDWARE_ID NUMBER
PF* EXPERIMENT_ID NUMBER
HARDWARE_USAGE_REL_PK (HARDWARE_ID, EXPERIMENT_ID)
SYS_C0010938 (HARDWARE_ID, EXPERIMENT_ID)
SYSMAN.HARDWARE
P * HARDWARE_ID NUMBER
* TITLE VARCHAR2 (50 BYTE)
* TYPE VARCHAR2 (30 BYTE)
DESCRIPTION VARCHAR2 (30 BYTE)
* IS_DEFAULT NUMBER (1)
PK_HARDWARE (HARDWARE_ID)
PK_HARDWARE (HARDWARE_ID)
SYSMAN.ELECTRODE_LOCATION
P * ELECTRODE_LOCATION_ID NUMBER
* TITLE VARCHAR2 (150 BYTE)
* SHORTCUT VARCHAR2 (30 BYTE)
* DESCRIPTION VARCHAR2 (300 BYTE)
F * ELECTRODE_FIX_ID NUMBER
* ELECTRODE_TYPE_ID NUMBER
* IS_DEFAULT NUMBER
ELECTRODE_LOCATION_PK (ELECTRODE_LOCATION_ID)
SYS_C0010873 (ELECTRODE_LOCATION_ID)
SYSMAN.ELECTRODE_SYSTEM_GROUP_REL
PF* RESEARCH_GROUP_ID NUMBER
PF* ELECTRODE_SYSTEM_ID NUMBER
ELECTRODE_SYSTEM_GROUP_REL_PK (ELECTRODE_SYSTEM_ID, RESEARCH_GROUP_ID)
SYS_C0010930 (ELECTRODE_SYSTEM_ID, RESEARCH_GROUP_ID)
SYSMAN.WEATHER_GROUP_REL
PF* WEATHER_ID NUMBER
PF* RESEARCH_GROUP_ID NUMBER
REL_WEATHER_GROUP_PK (WEATHER_ID, RESEARCH_GROUP_ID)
REL_WEATHER_GROUP_PK (WEATHER_ID, RESEARCH_GROUP_ID)
SYSMAN.PROJECT_TYPE_GROUP_REL
PF* PROJECT_TYPE_ID NUMBER
PF* RESEARCH_GROUP_ID NUMBER
PROJECT_TYPE_GROUP_REL_PK (RESEARCH_GROUP_ID, PROJECT_TYPE_ID)
SYS_C0010941 (RESEARCH_GROUP_ID, PROJECT_TYPE_ID)
SYSMAN.ELECTRODE_SYSTEM
P * ELECTRODE_SYSTEM_ID NUMBER
* TITLE VARCHAR2 (150 BYTE)
* DESCRIPTION VARCHAR2 (300 BYTE)
* IS_DEFAULT NUMBER
ELECTRODE_SYSTEM_PK (ELECTRODE_SYSTEM_ID)
SYS_C0010880 (ELECTRODE_SYSTEM_ID)
SYSMAN.WEATHER
P * WEATHER_ID NUMBER
DESCRIPTION VARCHAR2 (30 BYTE)
* TITLE VARCHAR2 (30 BYTE)
* IS_DEFAULT NUMBER (1)
WEATHER_PK (WEATHER_ID)
SYS_C0010828 (WEATHER_ID)
SYSMAN.PERSON
P * PERSON_ID NUMBER
GIVENNAME VARCHAR2 (50 BYTE)
* SURNAME VARCHAR2 (50 BYTE)
DATE_OF_BIRTH DATE
* GENDER CHAR (1 BYTE)
EMAIL VARCHAR2 (50 BYTE)
PHONE_NUMBER VARCHAR2 (20 BYTE)
NOTE VARCHAR2 (255 BYTE)
USERNAME VARCHAR2 (50 BYTE)
PASSWORD VARCHAR2 (80 BYTE)
AUTHORITY VARCHAR2 (50 BYTE)
F DEFAULT_GROUP_ID NUMBER
REGISTRATION_DATE DATE
CONFIRMED NUMBER
AUTHENTICATION VARCHAR2 (50 BYTE)
FB_UID NUMBER
* LATERALITY CHAR (1 BYTE)
F * EDUCATION_LEVEL_ID NUMBER
PERSON_PK (PERSON_ID)
SYS_C0010822 (PERSON_ID)
SYSMAN.COEXPERIMENTER_REL
PF* PERSON_ID NUMBER
PF* EXPERIMENT_ID NUMBER
COEXPERIMENTER_REL_PK (PERSON_ID, EXPERIMENT_ID)
SYS_C0010935 (PERSON_ID, EXPERIMENT_ID)
SYSMAN.EXPERIMENT
P * EXPERIMENT_ID NUMBER
F * SCENARIO_ID NUMBER
START_TIME DATE
END_TIME DATE
TEMPERATURE NUMBER
ENVIRONMENT_NOTE VARCHAR2 (255 BYTE)
F * SUBJECT_PERSON_ID NUMBER
F * WEATHER_ID NUMBER
F * OWNER_ID NUMBER
F * RESEARCH_GROUP_ID NUMBER
* PRIVATE NUMBER (1)
F * ARTEFACT_ID NUMBER
F * ELECTRODE_CONF_ID NUMBER
F SUBJECT_GROUP_ID NUMBER
EXPERIMENT_PK (EXPERIMENT_ID)
SYS_C0010851 (EXPERIMENT_ID)
SYSMAN.HARDWARE_GROUP_REL
PF* HARDWARE_ID NUMBER
PF* RESEARCH_GROUP_ID NUMBER
REL_HARDWARE_GROUP_PK (HARDWARE_ID, RESEARCH_GROUP_ID)
REL_HARDWARE_GROUP_PK (HARDWARE_ID, RESEARCH_GROUP_ID)
SYSMAN.ELECTRODE_CONF
P * ELECTRODE_CONF_ID NUMBER
* IMPEDANCE NUMBER
F * ELECTRODE_SYSTEM_ID NUMBER
DESC_IMG_ID NUMBER
ELECTRODE_CONF_PK (ELECTRODE_CONF_ID)
SYS_C0010885 (ELECTRODE_CONF_ID)
SYSMAN.ARTEFACT_GROUP_REL
PF* RESEARCH_GROUP_ID NUMBER
PF* ARTEFACT_ID NUMBER
ARTEFACT_GROUP_REL_PK (ARTEFACT_ID, RESEARCH_GROUP_ID)
SYS_C0010915 (ARTEFACT_ID, RESEARCH_GROUP_ID)
SYSMAN.ARTEFACT
P * ARTEFACT_ID NUMBER
* COMPENSATION VARCHAR2 (300 BYTE)
* REJECT_CONDITION VARCHAR2 (300 BYTE)
ARTEFACT_PK (ARTEFACT_ID)
ARTEFACT_PK (ARTEFACT_ID)
SYSMAN.RESEARCH_GROUP_MEMBERSHIP
PF* PERSON_ID NUMBER
PF* RESEARCH_GROUP_ID NUMBER
* AUTHORITY VARCHAR2 (30 BYTE)
RESEARCH_GROUP_MEMBERSHIP_PK (PERSON_ID, RESEARCH_GROUP_ID)
SYS_C0010905 (PERSON_ID, RESEARCH_GROUP_ID)
SYSMAN.SUBJECT_GROUP
P * SUBJECT_GROUP_ID NUMBER
* TITLE VARCHAR2 (150 BYTE)
* DESCRIPTION VARCHAR2 (250 BYTE)
SUBJECT_GROUP_PK (SUBJECT_GROUP_ID)
SYS_C0010859 (SUBJECT_GROUP_ID)
SYSMAN.ELECTRODE_LOCATION_REL
PF* ELECTRODE_CONF_ID NUMBER
PF* ELECTRODE_LOCATION_ID NUMBER
ELECTRODE_LOCATION_REL_PK (ELECTRODE_CONF_ID, ELECTRODE_LOCATION_ID)
SYS_C0010947 (ELECTRODE_CONF_ID, ELECTRODE_LOCATION_ID)
SYSMAN.SCENARIO
P * SCENARIO_ID NUMBER
* TITLE VARCHAR2 (40 BYTE)
SCENARIO_LENGTH NUMBER
DESCRIPTION VARCHAR2 (255 BYTE)
F * OWNER_ID NUMBER
F * RESEARCH_GROUP_ID NUMBER
* PRIVATE NUMBER (1)
SCENARIO_NAME VARCHAR2 (100 BYTE)
MIMETYPE VARCHAR2 (30 BYTE)
SCENARIO_PK (SCENARIO_ID)
SYS_C0010894 (SCENARIO_ID)
SYSMAN.PROJECT_TYPE_REL
PF* EXPERIMENT_ID NUMBER
PF* PROJECT_TYPE_ID NUMBER
PROJECT_TYPE_REL_PK (EXPERIMENT_ID, PROJECT_TYPE_ID)
SYS_C0010944 (EXPERIMENT_ID, PROJECT_TYPE_ID)
SYSMAN.PROJECT_TYPE
P * PROJECT_TYPE_ID NUMBER
* TITLE VARCHAR2 (150 BYTE)
* DESCRIPTION VARCHAR2 (250 BYTE)
PROJECT_TYPE_PK (PROJECT_TYPE_ID)
SYS_C0010837 (PROJECT_TYPE_ID)

Figure B.2 *Oracle database EEG/ERP test*

ATTACHMENT C

OrientDB test database statistics

***Table C.1** OrientDB graph database statistics - OrientDB default classes are marked, they show the total number of vertices and edge in the database*

Class name	superClass	Alias	Cluster number	Default cluster number	Number of records
ARTEFACT	OGraphVertex		10	10	220000
ARTICLES	OGraphVertex		11	11	220000
ART_SUBSCRIBED	OGraphEdge		52	52	220000
ART_SUBSCRIBED_BY	OGraphEdge		53	53	220000
CONFIGURED	OGraphEdge		39	39	440000
COOPERATED	OGraphEdge		47	47	220000
COOPERATED_BY	OGraphEdge		46	46	220000
CREATED	OGraphEdge		57	57	220000
CREATED_BY	OGraphEdge		29	29	660000
DEFINED	OGraphEdge		38	38	660000
DEFINED_BY	OGraphEdge		26	26	440000
EDUCATION_LEVEL	OGraphVertex		12	12	220000
ELECTRODE_CONF	OGraphVertex		13	13	220000
ELECTRODE_FIX	OGraphVertex		14	14	220000
ELECTRODE_LOCATION	OGraphVertex		15	15	220000
ELECTRODE_SYSTEM	OGraphVertex		16	16	220000
EXPERIMENT	OGraphVertex		17	17	220000
FIXED_BY	OGraphEdge		40	40	220000
HARDWARE	OGraphVertex		18	18	220000
HAS	OGraphEdge		37	37	1100000
IS_IN_PROJECT	OGraphEdge		48	48	220000
IS_MEMBER	OGraphEdge		55	55	220000
IS_MEMBER_OF	OGraphEdge		54	54	220000
LOCATED	OGraphEdge		44	44	220000
MEASURED	OGraphEdge		45	45	220000
OFunction			6	6	0
OGraphEdge		**E**	**9**	**9**	**9635463**
OGraphVertex		**V**	**8**	**8**	**3300000**
OIdentity			-1	-1	6
ORIDs			7	7	4054725
ORestricted			-1	-1	0
ORole	OIdentity		4	4	3
OUser	OIdentity		5	5	3

OWNED	OGraphEdge	31	31	440000
OWNED_BY	OGraphEdge	30	30	1495463
PERSON	OGraphVertex	19	19	220000
PROJECT_TYPE	OGraphVertex	20	20	220000
REACHED	OGraphEdge	51	51	220000
RESEARCH_GROUP	OGraphVertex	21	21	220000
SCENARIO	OGraphVertex	22	22	220000
SUBJECT_GROUP	OGraphVertex	23	23	220000
USED	OGraphEdge	28	28	1100000
USED_BY	OGraphEdge	27	27	660000
WEATHER	OGraphVertex	24	24	220000

Table C.2 *OrientDB indexes*

Name	Type	Class	Records
OrientDB indexes (for OrientDB SQL)			
ARTICLES.TIME	NOTUNIQUE	ARTICLES	220000
ARTICLES.TITLE	NOTUNIQUE	ARTICLES	175586
ELECTRODE_CONF.IMPEDANCE	NOTUNIQUE	ELECTRODE_CONF	220000
ELECTRODE_LOCATION.TITLE	NOTUNIQUE	ELECTRODE_LOCATION	220000
EXPERIMENT.TEMPERATURE	NOTUNIQUE	EXPERIMENT	175439
PERSON.GIVENNAME	NOTUNIQUE	PERSON	175693
PERSON.SURNAME	NOTUNIQUE	PERSON	220000
Blueprints indexes (for Gremlin)			
OGraphVertex.SURNAME	NOTUNIQUE	OGraphVertex	220000
OGraphVertex.TITLE	NOTUNIQUE	OGraphVertex	2155613
OGraphVertex.TYPE	NOTUNIQUE	OGraphVertex	1980000
Total = 11			**5762331**

ATTACHMENT D

Tabular results of measurement (Green percentages – OrientDB is faster, Red percentages – OrientDB is slower)

***Table D.1** Results of measurements for queries on single table/one vertex type*

Test	Description	Time in sec – average of 10 measurements (less is better)		OrientDB vs. Oracle
		Oracle database 11g	OrientDB 1.3.0 graphed	(+) faster, (-) slower
[1] Simple *SELECT* query with retrieving all values	1000 retrieved records	0,275	0,135	**102,78%**
	10 000 retrieved records	0,872	1,328	**-52,22%**
	100 000 retrieved records	8,109	12,989	**-60,19%**
	220 000 retrieved records (All)	18,788	28,414	**-51,23%**
[2] Simple *SELECT* query with *WHERE* condition no. 1 OrientDB: notunique index on property **TEMPERATURE**	1000 retrieved records	0,132	0,164	**-23,83%**
	10 000 retrieved records	0,846	1,495	**-76,65%**
	87 563 retrieved records (All)	6,715	12,992	**-93,45%**
[3] Simple *SELECT* query with *WHERE* condition no. 2 OrientDB: notunique index on property **GIVENNAME**	1000 retrieved records	0,221	0,641	**-189,57%**
	12 568 retrieved records (All)	1,540	8,756	**-468,32%**
[4] Simple *SELECT* query with *ORDER BY* OrientDB: notunique index on property **TIME**	1000 records retrieved	0,080	0,050	**58,92%**
	10 000 records retrieved	0,541	0,382	**41,64%**
	100 000 records retrieved	4,768	3,461	**37,77%**
	220 000 retrieved records (All)	10,184	7,500	**35,79%**
[5] Simple *SELECT* query with *GROUP_BY* OrientDB: notunique index on property **TITLE**	13 242 retrieved records (All)	0,573	9,086	**-1483,89%**
[6] Simple *SELECT* query with *BETWEEN* OrientDB: notunique index on property **IMPEDANCE**	1000 retrieved records	0,082	0,084	**-3,30%**
	10 000 retrieved records	0,287	0,821	**-186%**
	128 486 retrieved records (All)	3,225	9,905	**-207,10%**

***Table D.2** Results of measurements for queries on multiple tables (JOINs)*

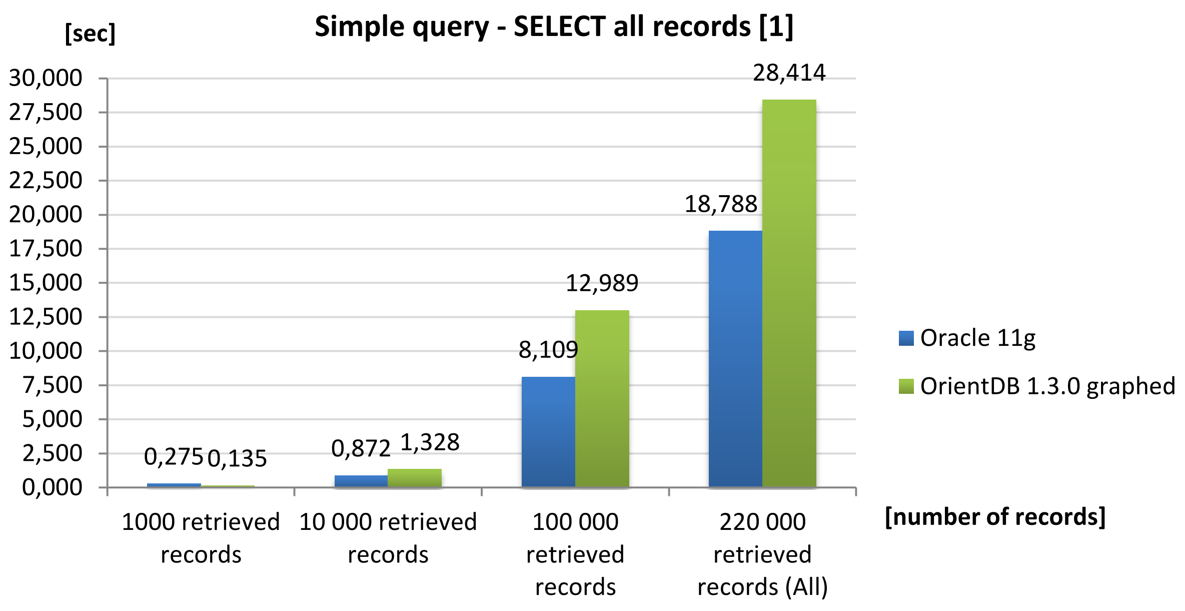
Test	Time in sec – average of 10 measurements (less is better)			OrientDB vs. Oracle
	Description	**Oracle 11g**	**OrientDB 1.3.0 graphed**	**(+) faster, (-) slower**
[1] Join query no. 1, Oracle: unique index on **foreign keys** , OrientDB: notunique index on property **SURNAME,**	613 retrieved records	0,925	0,131	**602,96%**
[2] Join query no. 2, Oracle: unique index on **foreign keys**, OrientDB: notunique index on properties **TITLE** and **TYPE,**	641 retrieved records	1,925	1,233	**56,09%**
[3] Join query no. 3, Oracle: unique index on **foreign keys**, OrientDB: notunique index on properties **TITLE,**	3173 retrieved records	1,003	1,296	**-29,20%**
[4] Join query no. 4, Oracle: unique index on **foreign keys**, OrientDB: notunique index on properties **TITLE** and **TYPE,**	768 retrieved records	10,700	0,487	**2095,90%**
[5] Join query no. 5, Oracle: unique index on **foreign keys**, OrientDB: notunique index on properties **SURNAME** and **TYPE,**	1196 retrieved records	1,407	0,567	**148,31%**

Table D.3 *Results of measurements for commands INSERT, UPDATE, DELETE*

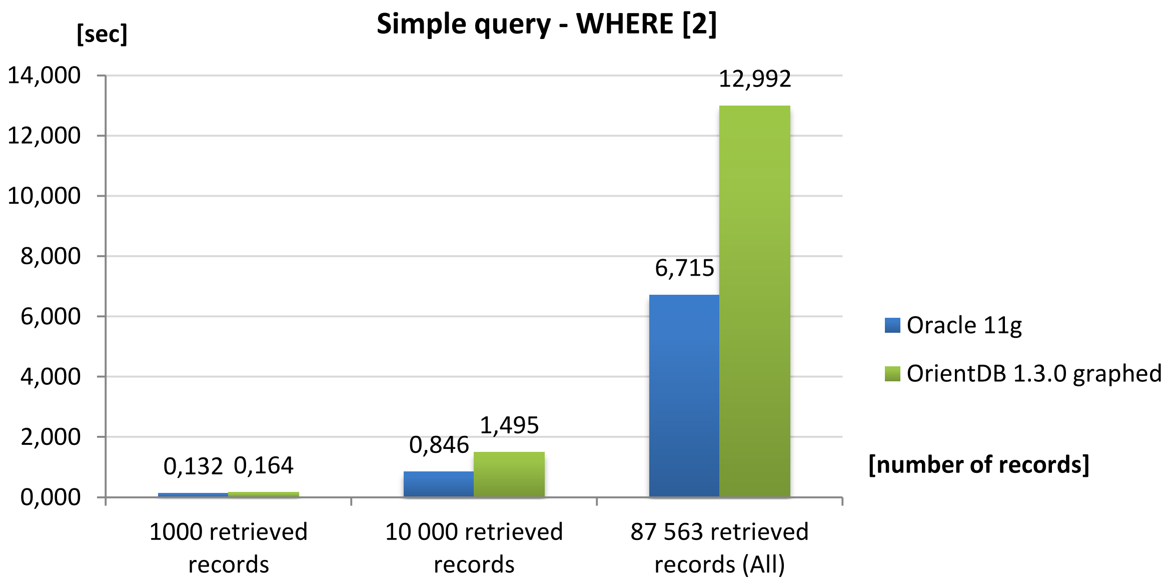
Test	Time in sec - average of 5 measurements (less is better)		OrientDB vs. Oracle
	Oracle 11g	OrientDB 1.3.0 graphed	(+) faster, (-) slower
Insert of 1000 records	1,425	0,778	83,18%
Insert of 10 000 records	19,614	8,820	122,39%
Insert of 100 000 records	251,464	26,535	847,65%
Insert of 220 000 records	638,618	50,939	1153,68%
Update one column on 1000 records	0,412	0,165	149,94%
Update one column on 10 000 records	0,481	0,716	-48,73%
Update one column on 100 000 records	1,406	4,100	-191,59%
Update one column on 220 000 records	4,026	8,991	-123,32%
Delete of 1000 records	0,312	0,235	33,05%
Delete of 10 000 records	0,500	2,139	-327,92%
Delete of 100 000 records	1,420	13,443	-846,32%
Delete of 220 000 records	4,642	28,222	-507,94%

ATTACHMENT E

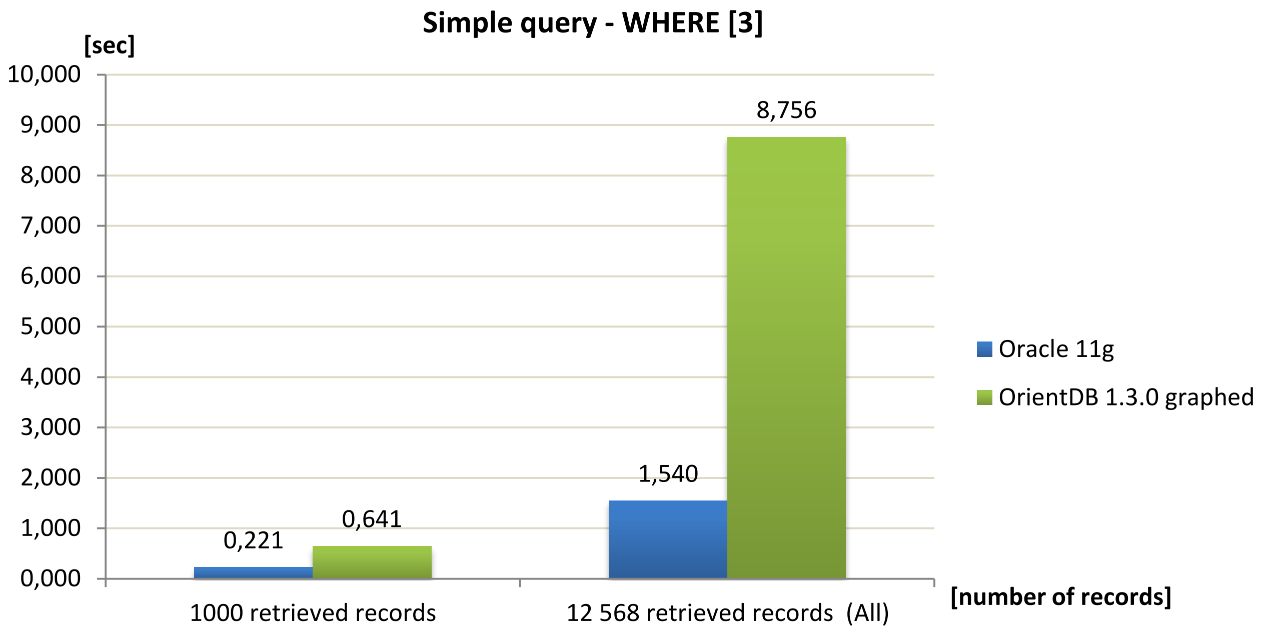
Graph comparison of query time executions averages of both tested database systems for all tested queries and commands



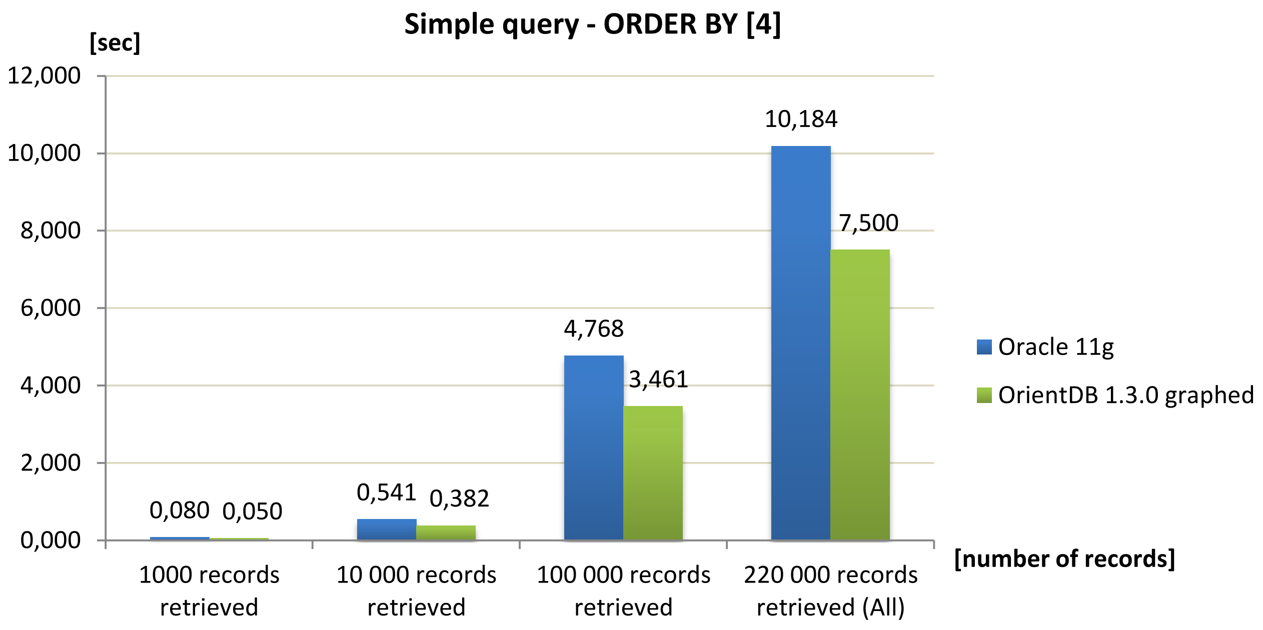
***Graph E.1** Average time of query execution depending on number of retrieved records*



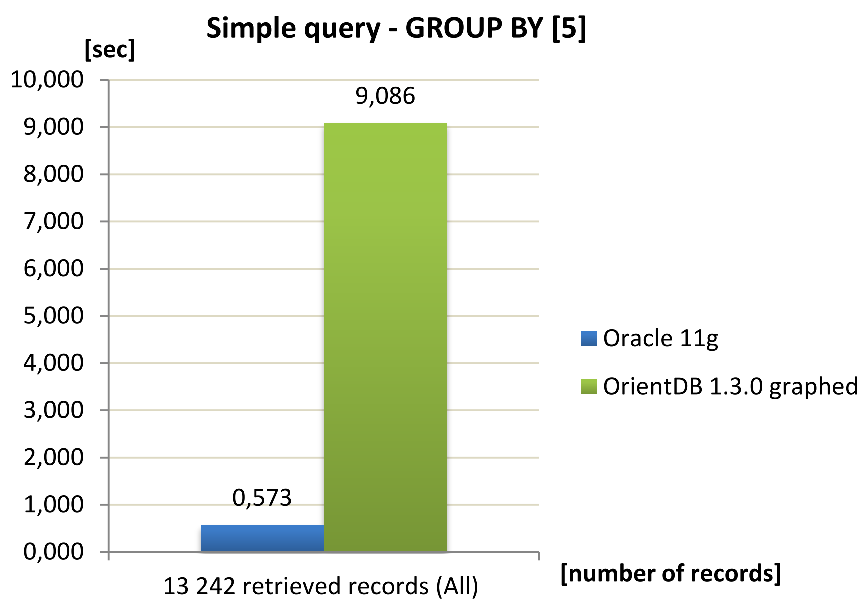
***Graph E.2** Average time of query execution depending on number of retrieved records (query contains WHERE clause)*



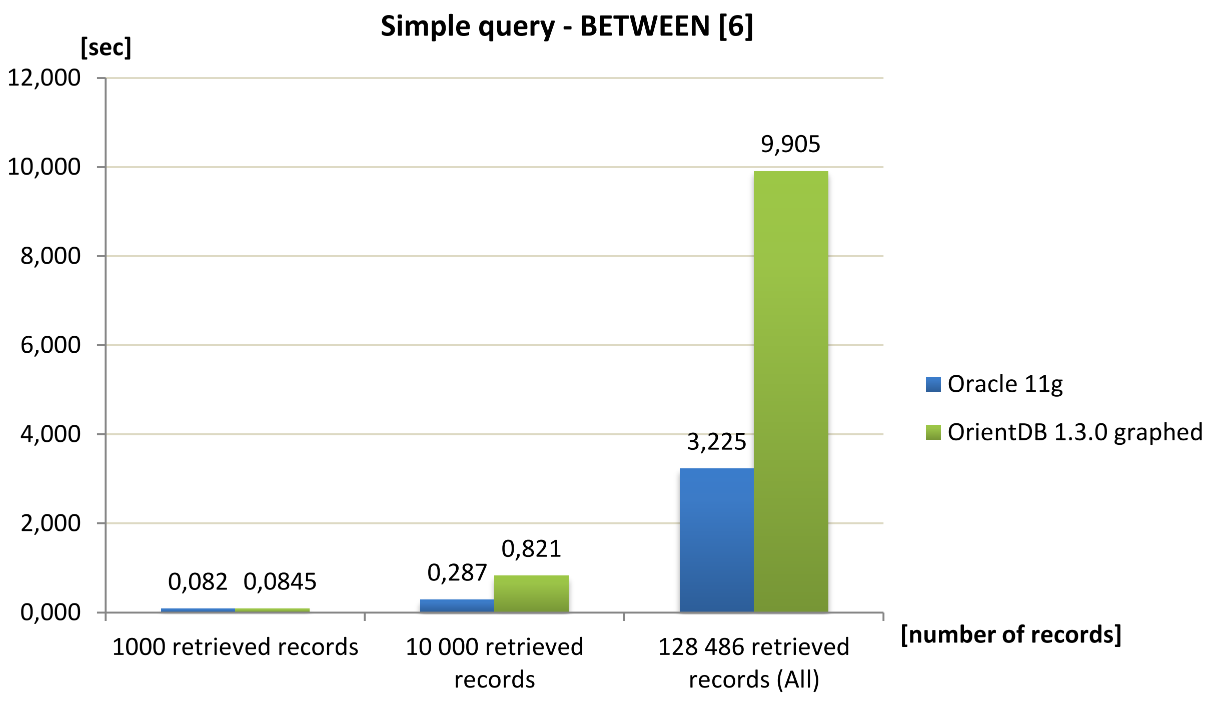
Graph E.3 *Average time of query execution depending on number of retrieved records (query contains WHERE clause)*



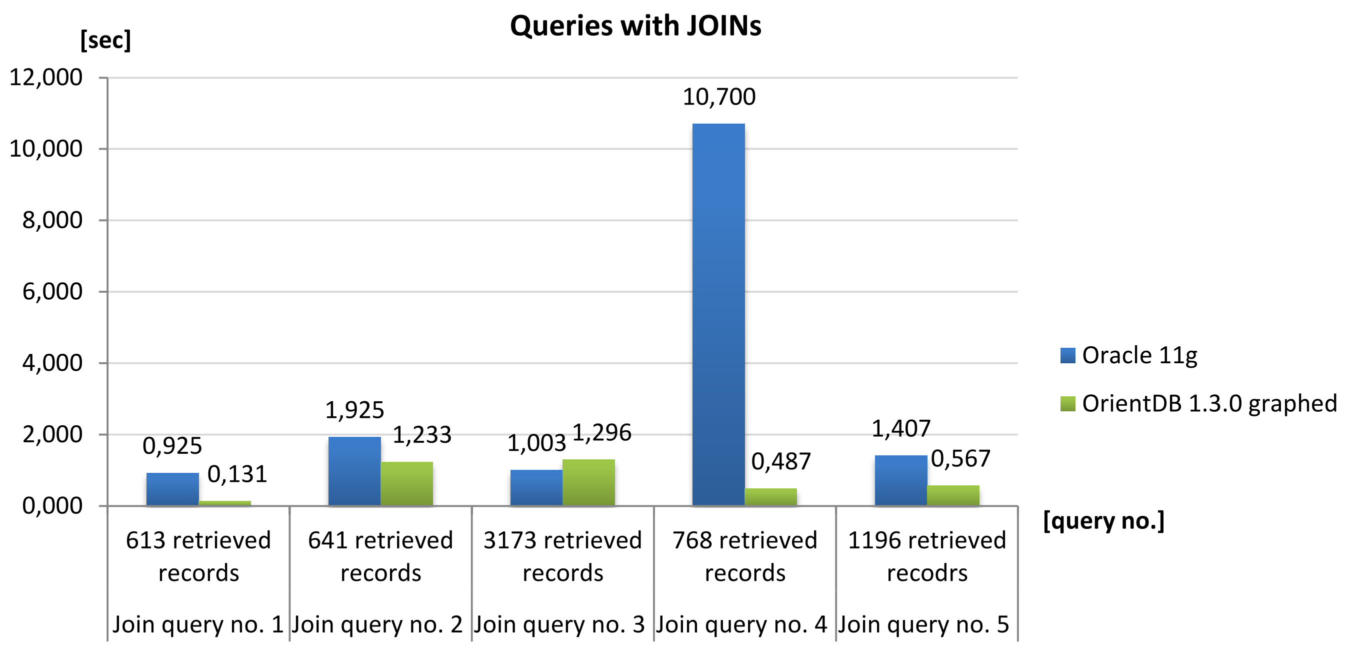
Graph E.4 *Average time of query execution depending on number of retrieved records (query contains ORDER_BY clause)*



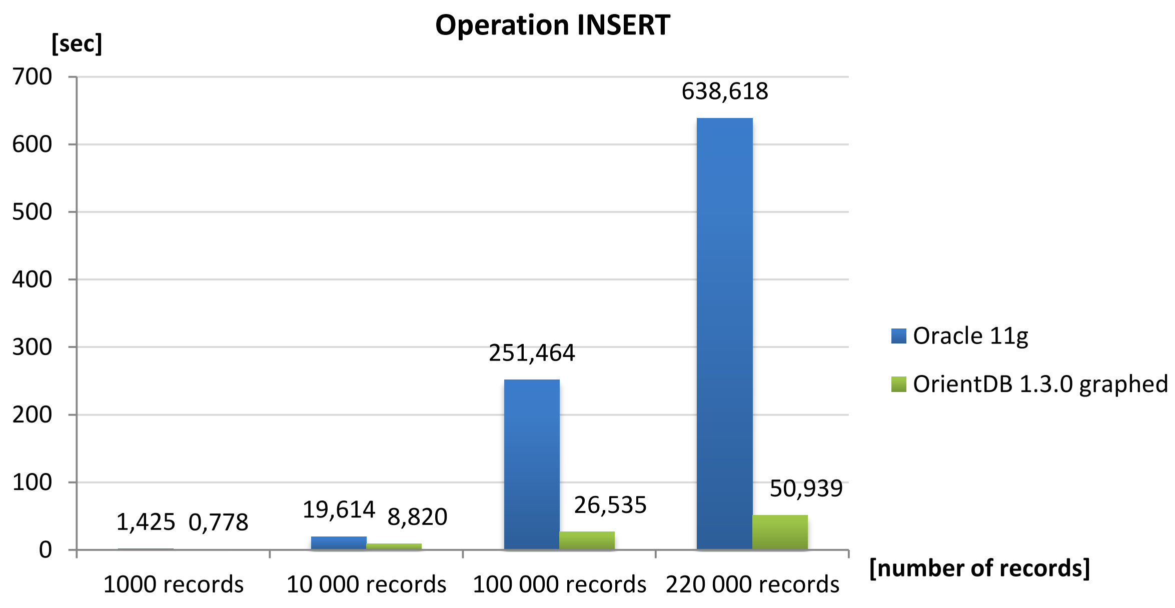
Graph E.5 *Average time of query execution depending on number of retrieved records (query contains GROUP BY clause)*



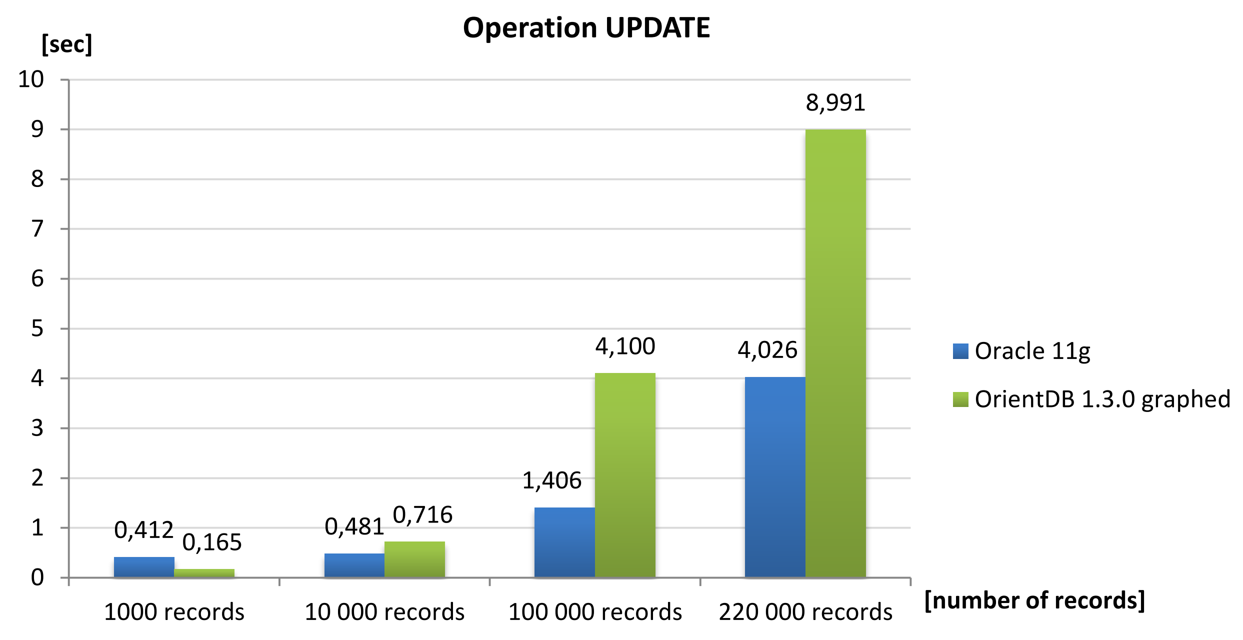
Graph E.6 *Average time of query execution depending on number of retrieved records (query contain BETWEEN clause)*



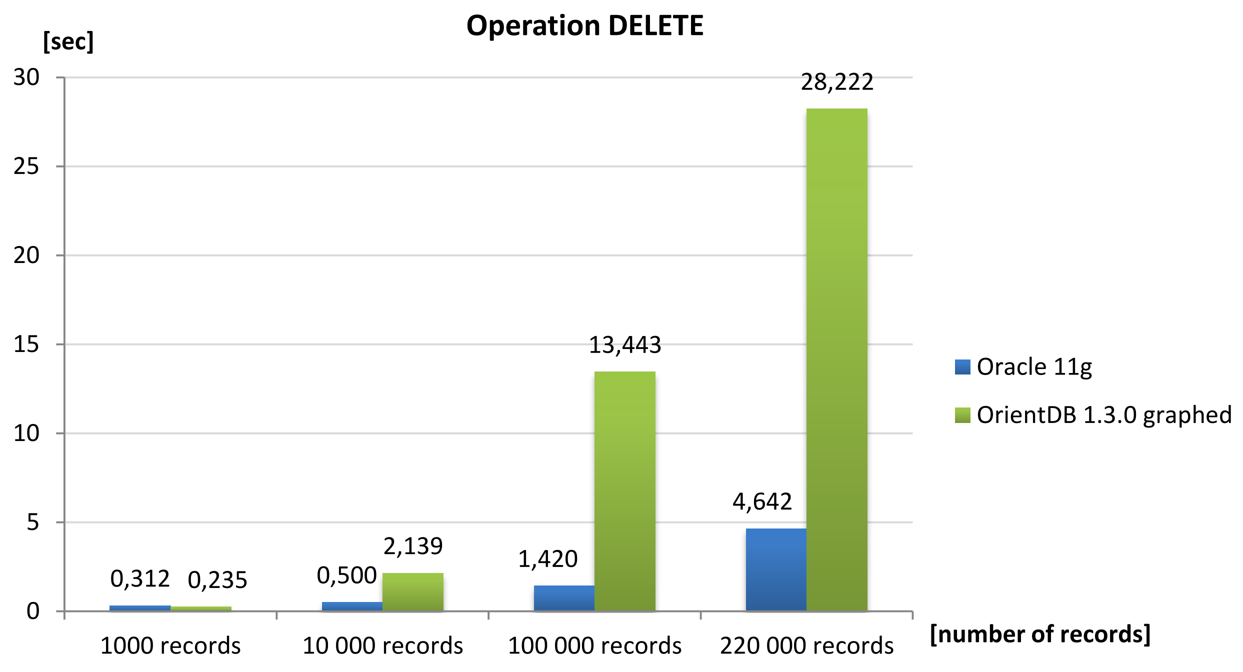
Graph E.7 *Average time of query execution depending on number of browsed tables/node types (queries over multiple tables and vertices types)*



Graph E.8 *Average time of command execution depending on number of retrieved records (command contains INSERT clause)*



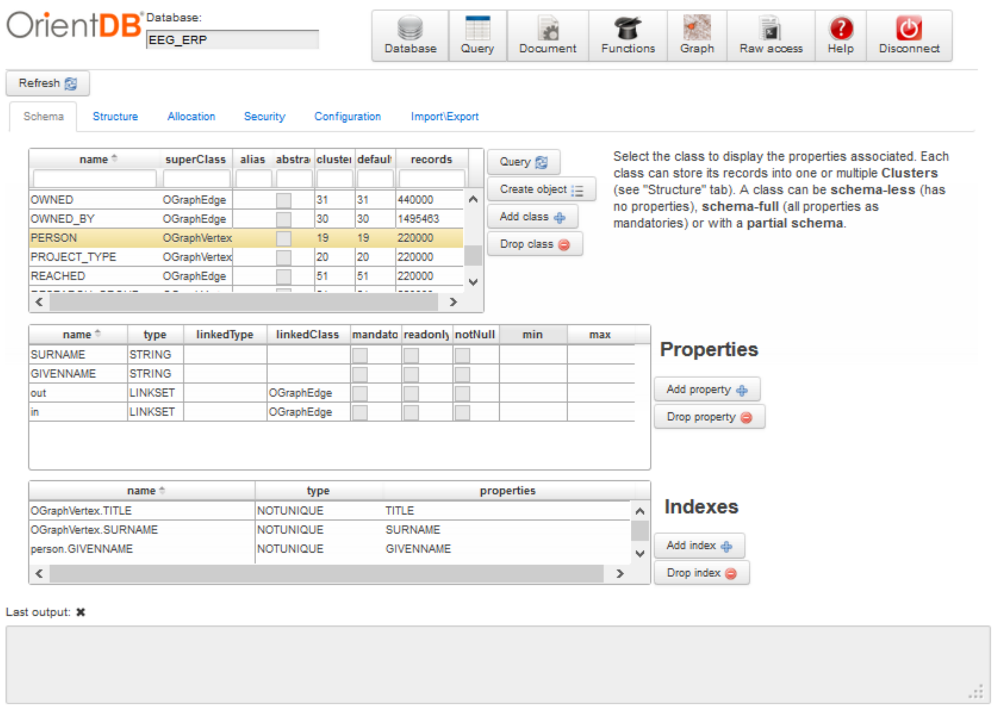
Graph E.9 *Average time of command execution depending on number of retrieved records (command contains UPDATE clause)*



Graph E.10 *Average time of command execution depending on number of retrieved records (command contains DELETE clause)*

ATTACHMENT F

OrientDB Studio – Samples of working with tested OrientDB graph database



Database: EEG_ERP

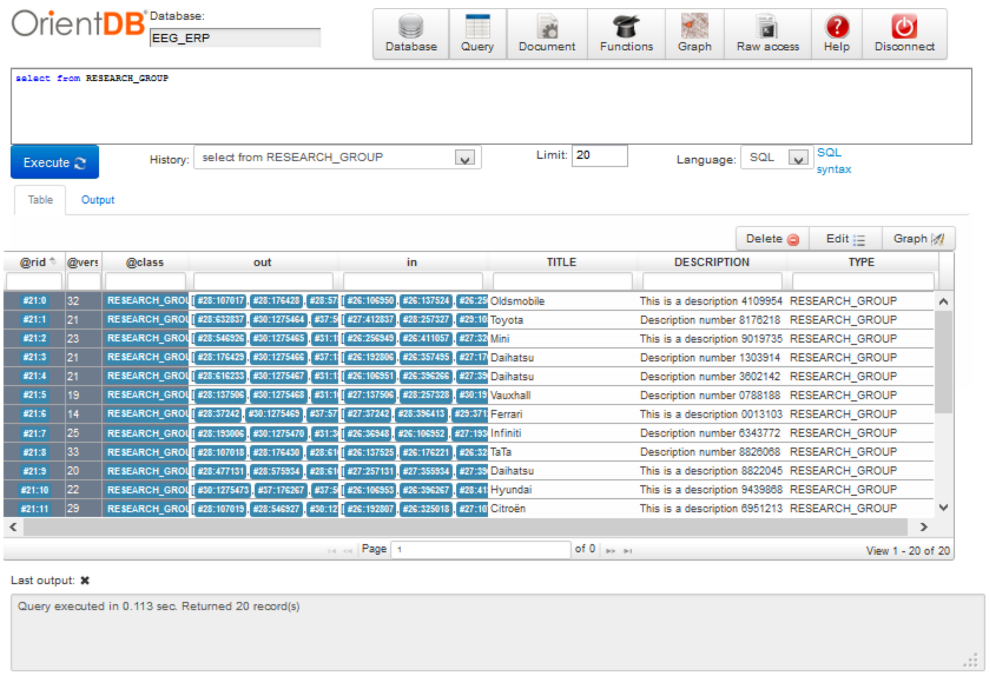
name	superClass	alias	abstra	cluste	defaul	records
OWNED	OGraphEdge		☐	31	31	440000
OWNED_BY	OGraphEdge		☐	30	30	1495463
PERSON	OGraphVertex		☐	19	19	220000
PROJECT_TYPE	OGraphVertex		☐	20	20	220000
REACHED	OGraphEdge		☐	51	51	220000

Select the class to display the properties associated. Each class can store its records into one or multiple **Clusters** (see "Structure" tab). A class can be **schema-less** (has no properties), **schema-full** (all properties as mandatories) or with a **partial schema**.

name	type	linkedType	linkedClass	mandato	readonly	notNull	min	max
SURNAME	STRING			☐	☐	☐		
GIVENNAME	STRING			☐	☐	☐		
out	LINKSET		OGraphEdge	☐	☐	☐		
in	LINKSET		OGraphEdge	☐	☐	☐		

name	type	properties
OGraphVertex.TITLE	NOTUNIQUE	TITLE
OGraphVertex.SURNAME	NOTUNIQUE	SURNAME
person.GIVENNAME	NOTUNIQUE	GIVENNAME

***Figure F.1** Main page with database statistics and options for managing the database*



Database: EEG_ERP

```
select from RESEARCH_GROUP
```

History: select from RESEARCH_GROUP

Limit: 20

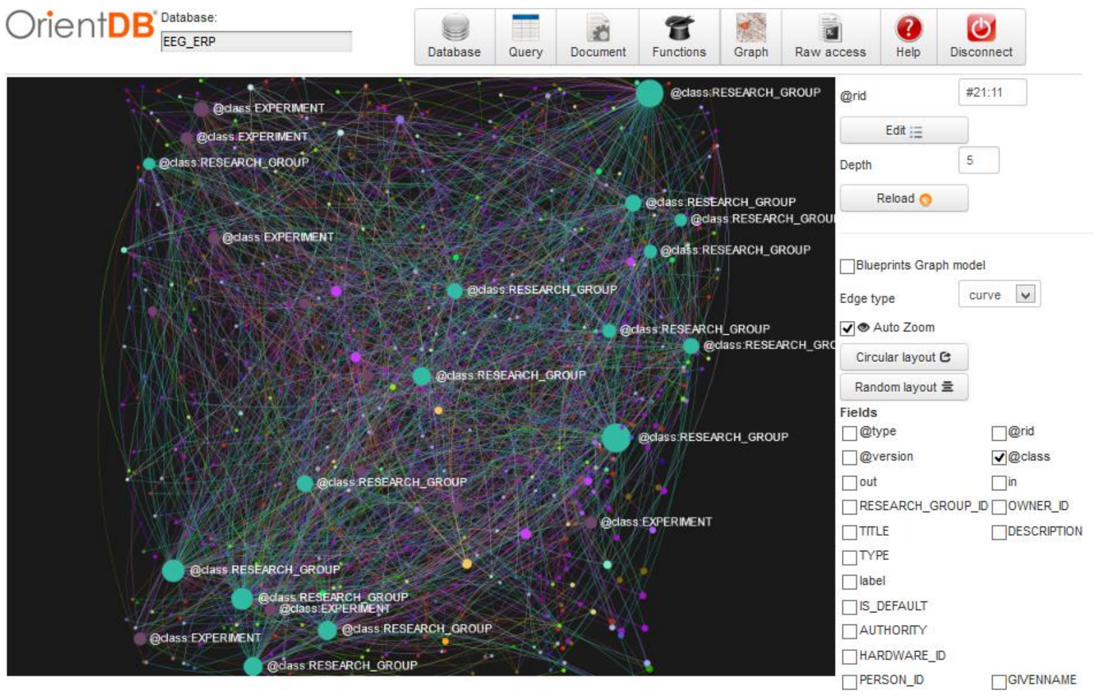
Language: SQL

@rid	@vers	@class	out	in	TITLE	DESCRIPTION	TYPE
#21:0	32	RESEARCH_GROU	[#28:107017, #28:176428, #28:57	[#26:106950, #26:137524, #26:25	Oldsmobile	This is a description 4109954	RESEARCH_GROUP
#21:1	21	RESEARCH_GROU	[#28:632837, #30:1275464, #37:5	[#27:412837, #28:257327, #29:10	Toyota	Description number 8176218	RESEARCH_GROUP
#21:2	23	RESEARCH_GROU	[#28:546926, #30:1275465, #31:1	[#26:256949, #26:411057, #27:32	Mini	This is a description 9019735	RESEARCH_GROUP
#21:3	21	RESEARCH_GROU	[#28:176429, #30:1275466, #37:1	[#26:192806, #26:357495, #27:17	Daihatsu	Description number 1303914	RESEARCH_GROUP
#21:4	21	RESEARCH_GROU	[#28:616233, #30:1275467, #31:1	[#26:106951, #26:396266, #27:39	Daihatsu	Description number 3602142	RESEARCH_GROUP
#21:5	19	RESEARCH_GROU	[#28:137506, #30:1275468, #31:1	[#27:137506, #28:257328, #30:19	Vauxhall	Description number 0788188	RESEARCH_GROUP
#21:6	14	RESEARCH_GROU	[#28:37242, #30:1275469, #37:57	[#27:37242, #28:396413, #29:371	Ferrari	This is a description 0013103	RESEARCH_GROUP
#21:7	25	RESEARCH_GROU	[#28:193006, #30:1275470, #31:3	[#26:36948, #26:106952, #27:193	Infiniti	Description number 6343772	RESEARCH_GROUP
#21:8	33	RESEARCH_GROU	[#28:107018, #28:176430, #28:61	[#26:137525, #26:176221, #26:32	TaTa	Description number 8826068	RESEARCH_GROUP
#21:9	20	RESEARCH_GROU	[#28:477131, #28:575934, #28:61	[#27:257131, #27:355934, #27:39	Daihatsu	This is a description 8822045	RESEARCH_GROUP
#21:10	22	RESEARCH_GROU	[#30:1275473, #37:176267, #37:5	[#26:106953, #26:396267, #28:41	Hyundai	This is a description 9439868	RESEARCH_GROUP
#21:11	29	RESEARCH_GROU	[#28:107019, #28:546927, #30:12	[#26:192807, #26:325018, #27:10	Citroën	This is a description 6951213	RESEARCH_GROUP

Page 1 of 0 View 1 - 20 of 20

Query executed in 0.113 sec. Returned 20 record(s)

***Figure F.2** Query page –query can be seen on class RESEARCH_GROUP (among fields it can be seen lists with outgoing (out) and incoming (in) edges), if we choose some row (root node) we can show the graph (see Figures F.3 – F.6)*



Database: EEG_ERP

@rid #21:11

Depth 5

☐ Blueprints Graph model

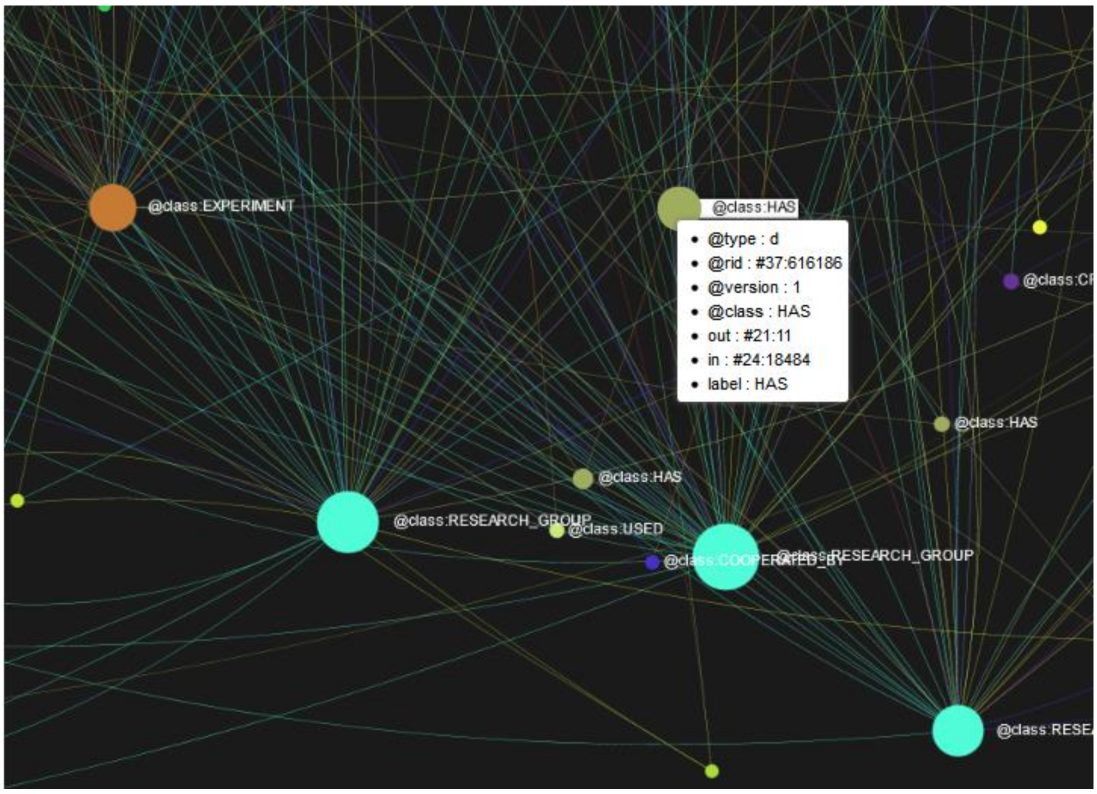
Edge type curve

☑ Auto Zoom

Fields

☐ @type	☐ @rid
☐ @version	☑ @class
☐ out	☐ in
☐ RESEARCH_GROUP_ID	☐ OWNER_ID
☐ TITLE	☐ DESCRIPTION
☐ TYPE	
☐ label	
☐ IS_DEFAULT	
☐ AUTHORITY	
☐ HARDWARE_ID	
☐ PERSON_ID	☐ GIVENNAME

Figure F.3 *Generated graph model from root of type RESEARCH_GROUP with Rid #21:11, graph is showed into the depth of connection 5 (in this case it is 683 vertices and 1423 edges)*



- @type : d
- @rid : #37:616186
- @version : 1
- @class : HAS
- out : #21:11
- in : #24:18484
- label : HAS

Figure F.4 *Detail of the graph – if we choose some node we can see its detailed description*

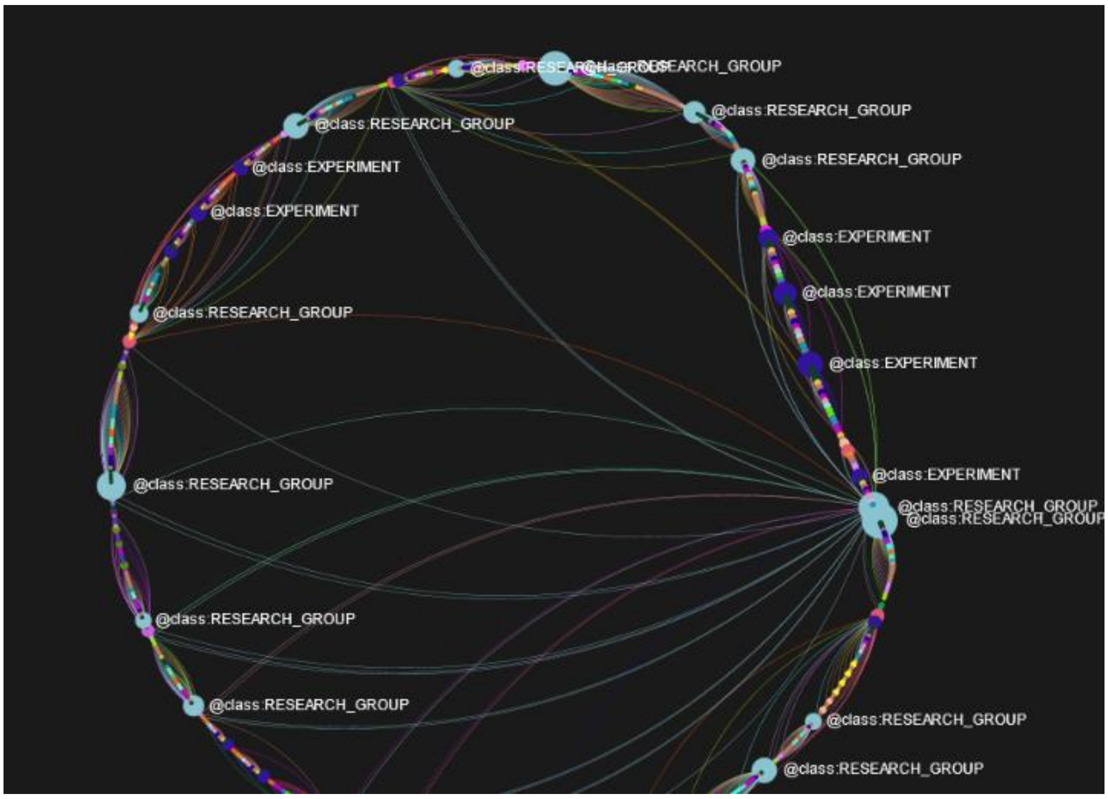


Figure F.5 *Circular layout*

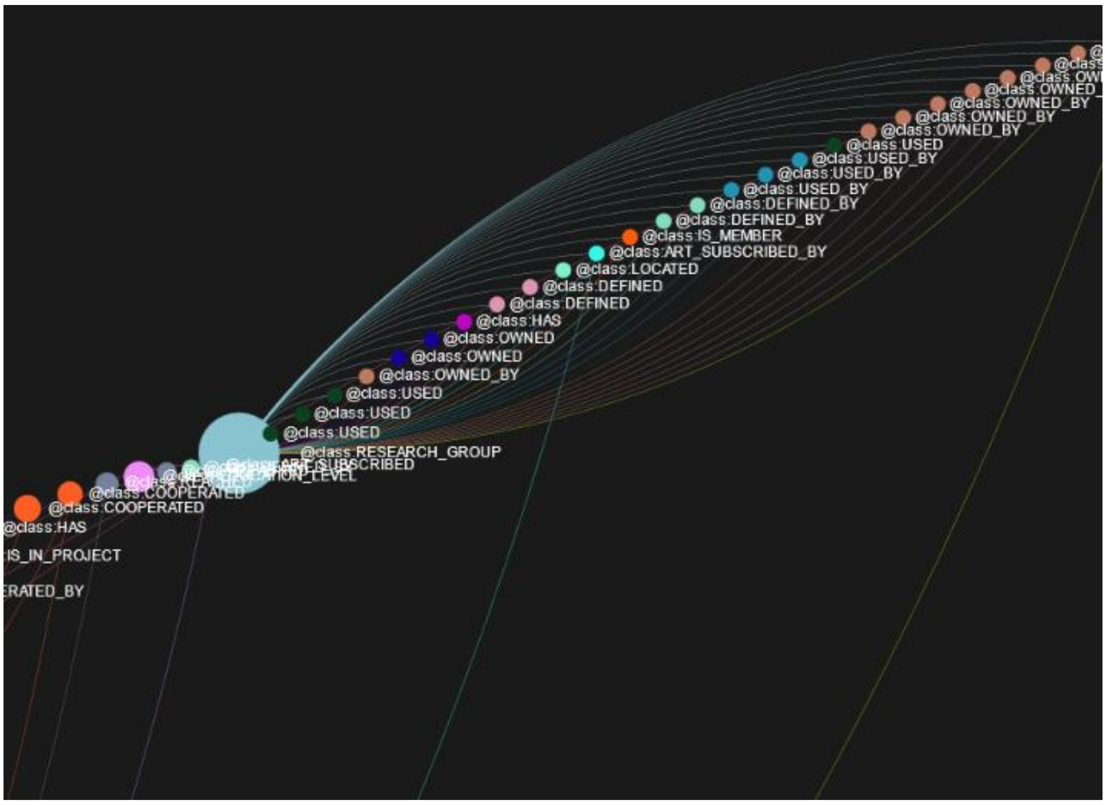


Figure F.6 *Detail of circular layout*

ATTACHMENT G

Tabular list of OrientDB properties and features from different viewpoints

Table G.1 *General information*

License	Apache
Level of documentation	Good (Wiki pages, Github)
Community	Active (Google group *Orient-db*, Google+)
Technical support	Mailing list, issue tracker, paid support
Free to use	Yes

Table G.2 *Database design*

Embeddable	Yes
Model	Document/Graph
Data storage model	Embedded, in-memory, remote (client/server), distributed
Import files	JSON, GraphML
Export files	JSON, GraphML
Native programming language	Java
Core language	Java
Object row storage	Cluster

Table G.3 *System requirements*

Operating system	All Linux distributions, Mac OS X, MS WIN 95/NT onward, Sun Solaris, HP-UX, IBM AIX
Java version	Java SE 6 and higher (64 bit JVM recommended)
Needs for additional software	No

Table G.4 *Integrity*

Model	ACID
Transactions	Yes
Referential integrity	Yes
Locks	Yes
Atomicity	Single document

Table G.5 *Indexes*

Automatic	Yes
Manual	Yes
Full-text	Yes
Other (structural, value, dictionary)	Yes

Table G.6 *Restrictions*

Maximal file size	2^78
Maximal number of records	2^63 per cluster
Maximal database size	2^78 (2^15 clusters per database)

Table G.7 *Features and deployment options*

Query language	Extended SQL, Gremlin
APIs/Clients	Java (native), JS, Scala, C, PHP, Ruby, .NET, Python, Clojure
RESTful	Yes
WebDAV	No
Scalable	Yes (partially horizontally)
Replication	Yes (multi-master)
Sharding	Yes (Since 2013)
Multi-user	Yes

Security model	Admin, writer, reader
Triggers	Yes (Hooks)
Visualization	Yes (OrientDB studio)
Extensions	TinkerPop stack
Spring support	Attempts (e.g. Orient-master project)
Data analysis	Gremlin (graph query language)
Standalone server	Yes
Web Application	Yes
Java library	Yes

ATTACHMENT H

OrientDB manual

Software requirements:

- Microsoft Windows/Linux/UNIX/Mac OS, (x86, x64)
- JVM 1.6 32/64bit and higher
- Java SE 6 for development of client applications

I provide here basic steps to run OrientDB Graph Edition1.3.0:

1) download OrientDB Graph Edition1.3.0 from the official website[3]
2) extract OrientDB zip archive into arbitrary directory on the disk
3) to start server, go to the *bin* folder and run script *server.sh* (Unix environment) or *server.bat* (Window environment)
4) to start console, go to the *bin* folder and run script *console.sh* or *console.bat*
5) put OrientDB *EEG_ERP database* from *attached DVD* into folder *databases*
6) now EEG_ERP database can be open from console, Gremlin console od OrientDB Studio (default login credentials are *admin* for username and *admin* for password in OrientDB Studio)

The way to connect, query and configure the OrientDB EEG_ERP database was described in *practical part* of this work. For more detailed information to work with OrientDB see the OrientDB's *reference documentation*[4].

[3] OrientDB Graph Edition stable release is available from:
<http://code.google.com/p/orient/downloads/list>
[4] OrientDB's reference documentation is available from:
<http://code.google.com/p/orient/wiki/Main?tm=6>

ATTACHMENT I

DVD content

DVD is attached to this work. There can be found following materials:

- SQL scripts for schema rebuilding of Oracle *test* database model
- SQL scripts for importing test data into Oracle *test* database model
- OrientDB 1.3.0 database
- created OrientDB database of the *whole* EEG/ERP portal test database (including BLOBs and CLOBs) which runs on the server students.kiv.zcu.cz
    - SQL scripts for creation of vertices and edges classes
    - SQL scripts for importing the data
    - SQL scripts for deletion of useless IDs fields from relational database
- created OrientDB database model from the *tested part* of EEG/ERP portal database model (without BLOBs and CLOBs)
    - SQL scripts for vertices and edges classes creation
    - SQL scripts for importing tested data
    - SQL scripts for deletion of useless IDs fields from relational database
- SQL, OrientDB SQL and Gremlin testing queries
- figures which are used in this work