Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Information Engineering (FEM)

Bachelor Thesis

Relational and post-relational databases

Elena Kulshetova

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

BACHELOR THESIS ASSIGNMENT

Elena Kulshetova

Informatics

Thesis title

Relational and post-relational databases

Objectives of thesis

This work focuses on the comparison of relational SQL and object-oriented Non-SQL databases on two concrete examples. Concept of object-oriented programming slightly dominates among the development, while data are still usually organized using relational structure. The main objective of this thesis is to analyse the individual type of databases, to reveal their advantages and disadvantages, to embrace the same data model, implement it and conclude on that basis.

Methodology

This work will consists of: 1) Comparing system performance of individual databases. 2) Compare database management options. 3) Comparing the ability to replicate data. 4) Development of relational and post-relational databases. In the literature review, definitions of basic concepts related to SQL and NO-SQL databases will be found as well as a detailed description of each type according to available information sources and knowledge bases incl. object modelling procedures. The practical part will include the development of database model in both relational and post-relational databases.

Declaration

I declare that I have worked on my bachelor thesis titled "Relational and Post-relational databases" by myself and I have used only the sources mentioned at the end of the thesis. As the author of the bachelor thesis, I declare that the thesis does not break copyrights of any their person.

In Prague on 15.03.2018 ____________________________

Acknowledgement

I would like to thank doc. Ing. Vojtěch Merunka, Ph.D. for their advice and support during my work on this thesis.

Relational and post-relational databases

Abstract

The main goal of the literature review of the thesis with topic „Relational and post-relational databases" is to describe a brief history of database, make an overview of existing database models, provide more detailed information on theory of relational database as well as post-relational or NoSQL alternative options. The main goal for practical part is to implement the same data model to both relational PostgreSQL and post-relational Ne04j database and to see the performance and the usability difference.

Keywords: database, relational database, post-relational database, SQL, NoSQL, database modelling, comparison

Relační a post-relační databáze

Abstrakt

Hlavním cílem této bakalářské práce na téma "Relační a post-relační databáze" je stručně popsat historii databáze, vytvořit přehled o existujících databázových modelech, poskytnout podrobnější informace o teorii relační databáze, stejně tak jako o alternativních možností post-relační nebo NoSQL databázi. Hlavním cílem praktické části je implementace stejného datového modelu do PostgreSQL a post-relační databázi Ne04j a zobrazení rozdílu výkonu a použitelnosti.

Klíčová slova: databáze, relační databáze, post-relační databáze, SQL, NoSQL, modelování databáze, porovnání

Table of content

List of figures

List of abbreviations

DB - database

DBMS – database management system

SQL – structured query language

NoSQL – not-only SQL

1 Introduction

People have had a tendency to save, record and forward information since many centuries ago. Data storage has a long history, starting with basic pictures depicting simple events to digital data of different types and forms. Since the 70's relational databases have become the golden standard of data warehousing.

Due to the exponential growth of the IT field, the need to store complex data has steadily increased and relational database might no longer be the best way to handle it. Modern databases that are not relational are described as post-relational; databases that are relational but do not use SQL are referred to as No-SQL.

This thesis compares relational databases to no-SQL databases; the theoretical part describes relational databases and various no-SQL databases, and the practical part is a comparison of PostgreSQL and Neo4J, particularly revealing the pros and cons of each option.

Objectives and Methodology

1.1 Objectives

This work focuses on the comparison of relational SQL and Non-SQL databases on two concrete examples. In the developing area the majority of used programming languages are object-oriented, but data are still usually organized using relational model. The main objective of this thesis is to analyse the individual type of databases, to reveal their advantages and disadvantages, to embrace the same data model, implement it and conclude on that basis.

1.2 Methodology

This work will consist of:

1) Comparing system performance of individual databases.
2) Compare database management options.
3) Comparing the ability to replicate data.
4) Development of relational and post-relational databases.

In the literature review, definitions of basic concepts related to SQL and NO-SQL databases will be found as well as a detailed description of each type according to available information sources and knowledge bases incl. object modelling procedures. The practical part will include the development of database model in both relational and post-relational databases.

2 Theoretical background of databases

2.1 Introduction to Databases

Database represents a software solution that serves to store data in a way that enables later analysis, eases the process of browsing and operating with data using script, and managed connections of related data. Data stored in the database is available to a large number of users and applications, thus increasing productivity, quality and accuracy of further search and analysis.

2.1.1 Brief history

I would like to start the thesis with a brief history according to the article "Histiry of databases" by Kristi L. Berg et. al the precursor of the database is paper filing cabinet, which has allowed to arrange data according to first and even second degree classification. All operations with the filing cabinet are manual. First example of machine data processing were electromechanical machines that used punch plates as the storage medium. The invention of computer technology changed the way of handling data since a large amount of data could be stored on a small physical item.

One of the greatest impulses for the development of databases was high cost of early computers by private companies in mid-60's. Two popular data models developed in this decade were: A network model called CODASYL (Conference on Data System Language) and a hierarchical model called IMS (Information Management System). *Hierarchical* database was represented as an upside-down tree and is still one of the most widespread on the mainframe MVS (Multiple Virtual Storage is an operating system from IBM). The first *network* DBMS was developed for mainframe computers. Nowadays it is barely used.

70's is the time of invention of the first relational database, looking at data as tables and the first version of SQL. This is also the period when such broadly known notions as a database schema, a definition language schematic, subschema etc have emerged. This development has brought power-efficient systems, which after further development overcame network and hierarchical databases. Newly appeared entity-relational model has allowed developers to focus on data application instead of logical table structure.

The next decade is associated with object-oriented databases, in which information is represented in form of objects as in object-oriented programming languages, and uses the exact same model as in OOP. To overcome the problems of OODBMS and take advantage of both object-oriented and relational models, the Object Relational Database Model was developed in the early 90’s.

By the end of the 20th century the IT industry already had very exponential growth. The last decade brought us several platforms for application development and efficient tools for personal use such as Microsoft Excel, XML format and the Internet and World Wide Web. Since personal computers were used not only at work but at homes as well, client-server models were used in both cases.

Databases had some advancement as well because of the rise of demand for database internet connectors and online transaction processing.Talking about database specifically, a new way of storing data has appeared – NoSQL. It has quickly gained popularity among leading companies. Only in the year 2011 UnQL (Unstructured Query Language) - a specific language for NoSQL - has started development.

Besides this, at the beginning of the 21st century a development line has appeared that forms the opposite of relational databases because its members are not following the rules of ACID (Atomicity, Consistency, Isolation, Durability) and simplified procedural languages. It goes without saying that existing solutions and technologies were expanded and improved.

Nowadays databases tend to have more and more complex logic. Cloud computing, virtual reality and other technological progress (especially medical devices) are on the way already and going to bring significant change to the database world.

2.1.2 DBMS

A well-known publisher Chris Date in his book “An Introduction to Database System (8th edition)”, 2004 on page 10, provided a good explanation of the difference between the terms "database" and "database management systems". During the time the databases are used, this exact label has slightly slowed down and under the term database can be understood the database management system as well as the database system. Although, the relation can be expressed by the equation:

DB + DBMS = DBS (Database + Database Management System = Database System)

DBMS stands for database management system and is determined by the design of the software equipment that ensures work with the database, i.e. creates the interface between application programs and saved data. Its task is to efficiently handle, store large amount of data and it support operations such as create, read, update, delete.

The DBMS is an integral part of most applications, especially those with multi-layer architecture. "Between the physical database itself – that is, the data as physically stored – and the user of the system is a layer of software, known variously as the database manager or database server, or, most commonly, the database management system (DBMS)." (Ramez, Shamkat, 2011). Role of DBMS increases with database size, number of users in different sites (typical for large multinational organisations and Internet portals). DBMS must offer several qualifications - integrity, in other words consistent data organisation, data accessibility and security.

2.2 Database models

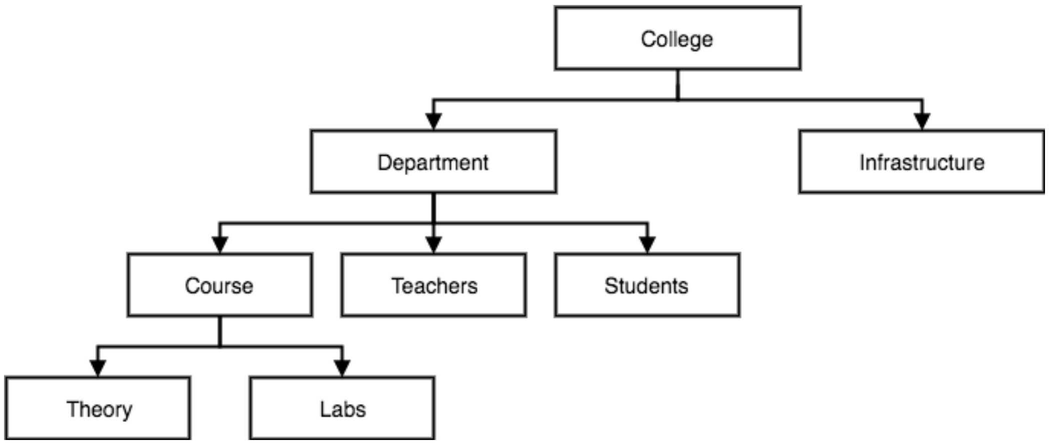
A database model is a set of rules for representing and logically organising data in database. Over the years, many database models have been deployed for data storage and management.
"**High-level** or **conceptual** data models provide concepts that are close to the way many users perceive data, whereas **low-level** or **physical** data models provide concepts that describe the details of how data is stored on the computer storage media, typically magnetic disks. Concepts provided by low-level data models are generally meant for computer specialists, not for end users." (Elmasri, Navathe, 2011).

Many data models have been proposed, which we can categorise according to the types of concepts they use to describe the database structure. In terms of the way we store data and the links between them, we can divide the databases into basic types:

2.2.1 Hierarchical model

Represents a special case of network model, having a multi-level structure that is similar to inverted tree. The parent table may belong to many child tables, but the child table always has only one parent. Hierarchical databases were used especially at the time of storing data on magnetic tapes since data access was strictly sequential. Today, the model is often considered to be inappropriate due to its inflexible structure and insufficient support for complex relationships of many applications. However, many implementations contain features that bypass these restrictions. You can read more information about hierarchical model in "SQL All-in-One For Dummies (2nd Edition)" by Allen G. Taylor.

Figure 1 Hierarchical model



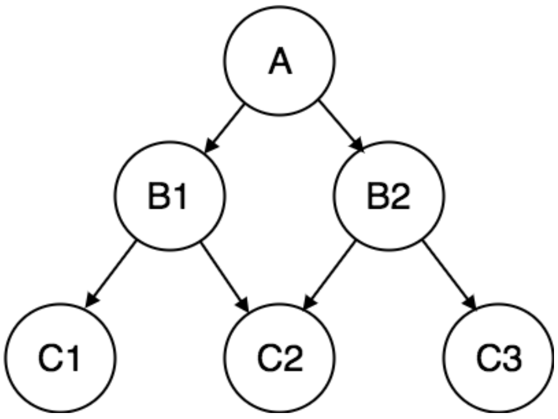
Source: https://www.studytonight.com/dbms/database-model.php, 2018

2.2.2 Network model

The database is represented by a graph where nodes are types of records, and arcs define the relationship between the data. The network model seeks to escape from some hierarchical constraints like permitting many-to-many relationships between linked records and involving multiple parent records. "Wanting to avoid the redundancy of the hierarchical model without sacrificing too much in the way of performance, the designers of the network model opted for an architecture that does not duplicate items, but instead increases the number of relationships associated with some items." (Taylor, 2011)
The structures are similar, but the tables are organised into groups that relate the pairs of tables to owners and members. Any table can be part of any group with other tables in a database. The database supports more complex queries than the hierarchical model does.

However, the network model also has its limitations. To work with groups, we need to know the database very well and it is difficult to change the structure without affecting it.

Figure 2 Network model



Source: https://www.studytonight.com/dbms/database-model.php, 2018

2.2.3 Object-oriented model

Allen G. Taylor mentions object-oriented database in his book “SQL ALL-in-One For Dummies”: Representing the data model using object-oriented programming languages where the data structure is closer to the real world. In this type of model, object contains both data and its relationships as in ER diagram and each object belongs to a class with similar structure (attributes) and behaviour (methods or simply procedures). Classes' are of hierarchical structure so one class cannot have more than one parent. Consequently, objects inherit attributes and methods from their parents.

An object-oriented database is organised around objects rather than actions, and data rather than logic. Since the database is integrated with the programming language, the programmer can maintain consistency within one environment, in this case both the database and the programming language will use the same representation of model.

He claims that such model was used mainly for storing graphical objects rather than text or numbers and that is because relational DB is not suitable for that data type. “Although object-oriented databases outperform relational databases for selected applications, they do not do as well in most mainstream applications and have not made much of a dent in the hegemony of the relational products. As a result, I will not be saying anything more about OODBMS products.” (Taylor, 2011)

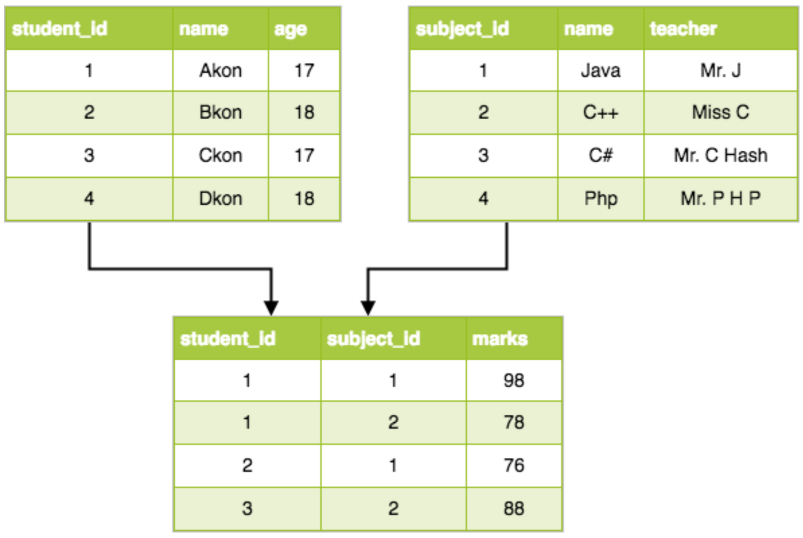
2.2.4 Object-relational model

This model attempts to unify the features of both relational and object databases. Permanently the information remains in tables, but some items may have a richer data structure called ADT, or abstract data types. These types are formed by combination of basic data types. In other words, it allows designers to incorporate objects into the familiar table structure. More specific derails you can find in the book of Ramez Elmasri and Shamkant B. Navathe, 2011, “Fundamentals of database systems (6th edition)”.

2.2.5 Relational model

The relational model is a strong competitor to many hierarchical or network system databases since it is the most commonly implemented model in modern databases. In comparison to other databases, the relational database is independent of the application - meaning a developer can change the structure of a database without affecting it. Relational model is the basis for SQL. The structure is based on relationships and tables allow the definition of complex relationships, which is the greatest and the weakest feature at the same time. I will talk about this model more in the next chapter.

Figure 3 Relational model



student_Id	name	age
1	Akon	17
2	Bkon	18
3	Ckon	17
4	Dkon	18

subject_Id	name	teacher
1	Java	Mr. J
2	C++	Miss C
3	C#	Mr. C Hash
4	Php	Mr. P H P

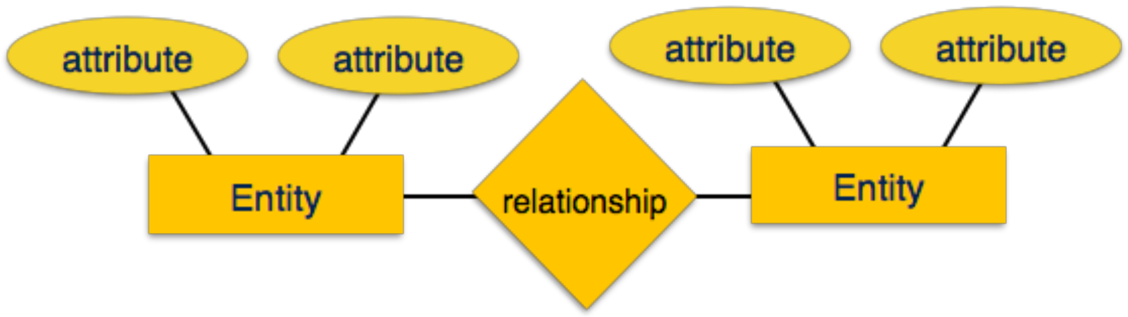
student_Id	subject_Id	marks
1	1	98
1	2	78
2	1	76
3	2	88

Source: https://www.studytonight.com/dbms/database-model.php, 2018

2.2.6 Entity-relationship model

I find the book "Fundamentals of Database Systems (6th edition)" 2011 by Elmasri, Ramez very well written and understandable. According to this book on pages 414-418 in Entity-Relationship model, relationships are created by storing an object of interest as entity and its characteristics as attributes while together they make up a domain. Different entities which are displayed in rectangular boxes are related using relationships of diamond-shaped boxes. Attributes, in the meantime, are shown in ovals. Mapping cardinalities can be: 1-1, 1-M, M-1, M-M. This way of organising data helps developers to easier understand what they're working with. This model is good to design a database which can be turned into tables in relational model. The star schema is a common form of the ER diagram in which a central fact table connects multiple dimensional tables.

Figure 4 Entity-relationship model



Source: https://www.tutorialspoint.com/dbms/dbms_data_models.htm, 2018

2.3 Relational database

Wladston Viana Ferreira Filho "Computer science distilled: learn the art of solving computational", explains the meaning of relational DB in a very simple way. A relational database is a relation and relationship based database. Often this concept refers not only to the database itself but also to its specific software solution. In relational database rows typically represent records and columns keep information about relationships between the records using foreign keys. Duplicated information is hard to manage and update. To avoid it, the relational model splits related information to different tables and makes connections between them. Data in relational database is structured and there is no redundancy as well.

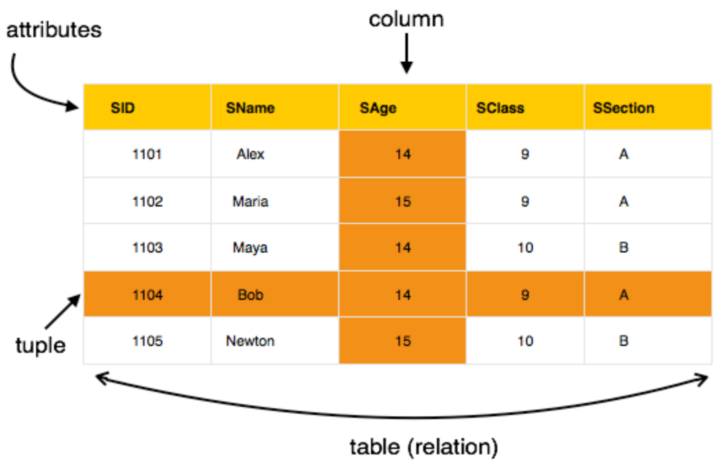
The **integrity** of data is very important, so RDBMS supports ACID transactions (Atomicity, Consistency, Isolation, and Durability). When a database is organised in a way that it is completely free of duplicate information, we say that the database is normalised. I will explain normalisation forms in the following chapter.

The main part of relational database management systems is accomplished using Structured Query Language to manipulate with data and in the meantime, it has positives such as stored procedures. That is simply SQL code that can be reused mainly to update or add new data.

2.3.1 Terminology

A new modern book of Wladston Viana Ferreira Filho "Computer science distilled: learn the art of solving computational", 2018 explains relational database in a very clear way. The basic constructor of relational databases are **relations** (database tables). The organisation of a database table is given by its fields and the restrictions they enforce. This combination of fields and restrictions is called **schema**. Relations have two-dimensional structure formed by the header and the body. The header represents attributes of the relation and data is stored in the body. Columns are called **fields** and they provide general description about the information that is stored in that specific column. One column has to contain data only of that same type (NUMBER, VARCHAR, BLOB etc) and domain, which is the set of all possible valid values. Rows of the table are the **records** or tuples. The row goes through the columns of the table and serves to save the data itself.

Figure 5 Components of relational model



SID	SName	SAge	SClass	SSection
1101	Alex	14	9	A
1102	Maria	15	9	A
1103	Maya	14	10	B
1104	Bob	14	9	A
1105	Newton	15	10	B

Source: https://www.tutorialspoint.com/dbms/dbms_data_models.htm, 2018

2.3.2 Relationships

The concept of relational database is known with tables and the connections between them. The connections are called relationships and they unite related data from different tables together. They are made using both primary and foreign keys. We will discuss them in detail in the following sub-chapters.

2.3.2.1 Primary key

Basing on the book “Relational database design and implementation: clearly explained”, 1998 by Jan L. Harrington, the row must also include one cell that is uniquely identifies the record and distinguish it from others. It may happen that two or more records would store the same information of a specific attribute, for example name, and while working with the data we would have problems searching or, what is more important, deleting the exact record we need. To overcome the inconvenience there is a rule that at least one column of the table must contain record’s unique identification where the content cannot be repeated through the whole column and that is called primary key.

The primary key is, as was said before, the identifier of the table row. Primary key may be a column or a combination of multiple columns, but with one compulsory condition - uniqueness. The primary key field must contain a value, i.e. cannot contain a blank NULL value. It is also possible to use an artificial key, which is a numeric or written identifier.

The aim of the primary key is to provide an efficient way of indexing data. New records are provided with different unique identifiers which distinguishes from all previous and next. Typically, an integer series is used: every new record gets the number one unit greater than the the last inserted id. As a best practice, primary keys are called id – short for identifier.

2.3.2.1 Foreign key

A foreign or a non-key is used to express relationships between tables. It is a field or a group of fields that allow you to identify the link between records from different tables to connect values within a database. The relationships are made by storing the primary key of the first table as a foreign key in the second table. By all means records of the second table we want to make a connection with also have to have their own primary keys. Allen G. Taylor in his book “SQL All-in-One For Dummies” says that it is the core of why the relational database is relational.

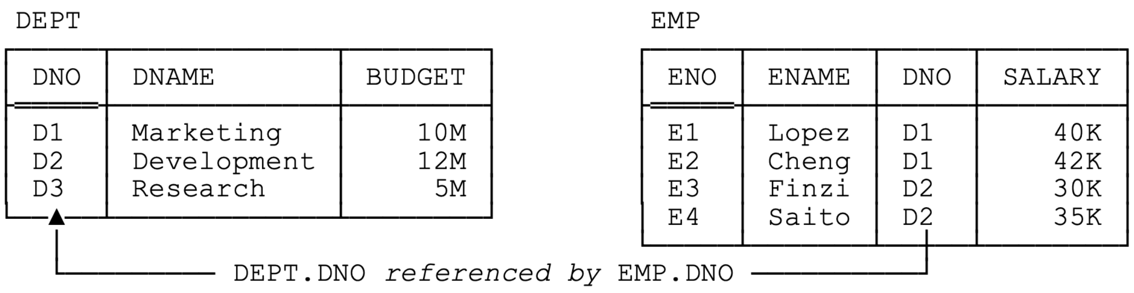
In contrary to primary keys, foreign keys can be duplicate - meaning the same primary key of a record from another table. Let’s have a look at an example: One customer bought 2 products. It means that one table of customers has 2 connections to the products table. This type of relationship is 1:N (one-to-many) and the foreign key is stored in the product table. In case of a M:N relationship we create a junction table which contains both ids of related tables. This abstraction allows us to create a wide range of relationships between tables.

To get the data from the more than one table in database we use command JOIN. This is a SQL keyword. We will discuss SQL in a further chapter.

2.3.2.2 Candidate key

“A candidate key is just a unique identifier; in other words, it’s a combination of attributes often but not always a “combination” consisting of just a single attribute such that every tuple in the relation has a unique value for the combination in question. In Fig. 1.1, for example, every department has a unique department number and every employee has a unique employee number, so we can say that {DNO} is a candidate key for DEPT and {ENO} is a candidate key for EMP. Note the braces, by the way; to repeat, candidate keys are always combinations, or sets, of attributes (even when the set in question contains just one attribute), and the conventional representation of a set on paper is as a comma list of elements enclosed in braces.” (Date, 2009)

Figure 6 Candidate key explanation



DEPT

DNO	DNAME	BUDGET
D1	Marketing	10M
D2	Development	12M
D3	Research	5M

EMP

ENO	ENAME	DNO	SALARY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K
E4	Saito	D2	35K

DEPT.DNO *referenced by* EMP.DNO

Source: C. J. Date “SQL and Relational Theory: How to Write Accurate SQL Code”, 2009

2.3.2.3 Superkey

“A superkey is a subset of the heading with the uniqueness property; a key is a superkey with the irreducibility property. All keys are superkeys, but “most” superkeys aren’t keys. The concept of a subkey can be useful in studying normalization. Here’s a definition: Let X be a subset of the heading of relvar R; then X is a subkey for R if and only if there exists some key K for R such that X is a subset of K. For example, the following are all of the subkeys for relvar SP: {SNO,PNO}, {SNO}, {PNO}, and {} (note that the empty set {} is necessarily a subkey for all possible relvars R). By way of illustration, here’s a definition of third normal form that makes use of the subkey concept: Relvar R is in third normal form, 3NF, if and only if, for every nontrivial functional dependency X Æ Y to which R is subject, X is a superkey or Y is a subkey. (A nontrivial functional dependency is one for which the right side isn’t a subset of the left side.) „ (Date, 2011)

2.3.3 Normalization

Again, I am referring to a great book “Fundamentals of database systems” 2011 by Ramez Elmasri and Shamkant B. Navathe. The process of transforming a database with duplicated data to one without is called **normalisation**. The steps of normalising a database’s structure are called normal forms and they have hierarchical structure. We can’t skip the steps or apply them chaotically - we must follow them in the given order.

2.3.3.1 First Normal From (1NF)

Each relation should have at least one primary key;
Repeating groups are not allowed;
It should only have atomic valued attributes/columns (cells have single value).

2.3.3.2 Second Normal Form (2NF)

It must already follow the rules of the First Normal form;
No partial functional dependency.

A functional dependency that holds in a relation is partial when removing one of the determining attributes gives a FD that holds in the relation. A FD that isn't partial it is full.

if {A, B} → {C} but also {A} → {C} then {C} is partially functionally dependent on {A,B}.

2.3.3.3 Third Normal Form (3NF)

It must already follow the rules of the Second Normal form;
No transitive dependency.

A transitive dependency is an indirect functional dependency, one in which {A→C} only due to {A→B} and {B→C}.

2.3.3.4 Boyce-Codd Normal Form (BCNF or 3.5NF)

It must already follow the rules of the Third Normal From;
No redundancy from any functional dependency.

For each functional dependency {A → B}, {A} should be a super Key.

2.3.3.5 Fourth Normal Form (4NF)

It must already follow the rules of the Boyce-Codd Normal Form;
No multi-valued dependency.

2.3.4 Integrity constraints

"An introduction to Database systems", 2003 by C.J. Date. tells us that integrity is a state when data stored in a database conforms to a set of specifically defined rules. Therefore, only data that matches these rules can be inserted into the database. Norms typically refer to the range of stored values, respect the data type set for that table column, or the links between the stored records. Integrity constraints serve to ensure integrity. This includes tools to prevent insertion of incorrect data or loss or damage of existing records during work with the database. For instance, it is possible to secure deletion of data that has already lost its meaning; for example, if we delete a user from the database, the rest of their records are deleted in other database tables as well.

The following constraints are commonly used:

NOT NULL - Ensures that a column cannot have a NULL value;

UNIQUE - Ensures that all values in a column are different;

PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table;

FOREIGN KEY - Uniquely identifies a row/record in another table;

CHECK - Ensures that all values in a column satisfies a specific condition;

DEFAULT - Sets a default value for a column when no value is specified;

INDEX - Used to create and retrieve data from the database very quickly.

Source: *SQL Constraints* [online]. [cit. 2018-11-21]. Dostupné z: https://www.w3schools.com/sql/sql_constraints.asp

2.3.5 Indexing

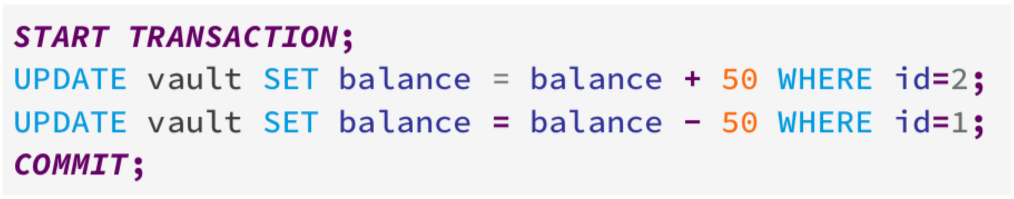
Index is a database object created to improve read performance. Tables in a database can have a large number of rows that are stored in arbitrary order, and their search by a given criterion by sequentially iterating the table row by row could take a long time. The index is formed from the values of one or several columns of the table and points to corresponding rows of the table and thus allows you to search for rows that meet the search criteria. DBMS builds an auxiliary index, mapping row IDs (mostly but not necessarily) to their respective addresses in memory. Acceleration of work with the use of indexes is achieved primarily due to the fact that the index has a structure optimised for searching - a balanced binary tree. A good and detailed explanation of b-tree, text and bitmap indexes and you can read in the book, “Learning SQL: Master SQL Fundamentals”2nd Edition, 2009 by Alan Beaulieu.

2.3.6 Transactions

A transaction generally represents any change in a database. Allen G. Taylor in his book "SQL All-in-One For Dummies.", 2011 explains why the transactions are needed. Performing multi-step updates without transactions eventually leads to wild, unexpected, and hidden inconsistencies in the data. Thus, main purposes are preventing from possible data inconsistency in case of system failure and providing reliable data keeping and correct backup.

The most common example of why transactions are needed is bank keeping. Imagine a situation when you want to send some money to your colleague's account. It would be performed in two actions: subtracting money from your account and adding to another. So is recorded in the database. What if someone queries the total balance of all accounts after a subtraction is recorded but before the corresponding addition is? Some money could be missing. Worse situation would be when the system loses power and shuts down between two operations. To solve these difficulties, transactions are made in a way that dependent actions are performed together.

Figure 7 Transacion explanation



```
START TRANSACTION;
UPDATE vault SET balance = balance + 50 WHERE id=2;
UPDATE vault SET balance = balance - 50 WHERE id=1;
COMMIT;
```

Source: Wladston Viana Ferreira Filho, "Computer science distilled: learn the art of solving computational problems", 2018

The acronym ACID describes properties for transactions and stands for:

Atomic: all changes are made (commit), or none (rollback).

Consistent: transaction won't violate declared system integrity constraints.

Isolated: results independent of concurrent transactions.

Durable: committed changes survive various classes of hardware failure.

2.4 SQL - Structured Query Language

C.J. Date is a great expert in relational database and in the 8th edition of his book "An Introduction to Database Systems", 2003 he describes the language to work with relational DBs. SQL is supported by just about every relational DBMS. According to the theory, simple operations (UNION, CROSS, PROJECT, SELECT, DIFFERENCE, and JOIN) perform all operations with data, other operations are merely combinations of these basic ones. SQL was originally intended to be a "data sublanguage" specifically. However, with the addition in 1996 of the SQL Persistent Stored Models (SQL/PSM, PSM for short), the standard became computationally complete – it now includes statements such as CALL, RETURN, SET, CASE, IF, LOOP, LEAVE, WHILE and REPEAT, as well as several related features such as the ability to declare variables and exception handlers.

A SQL query is a statement of what data should be retrieved from the database and this is a typical example of it:

Figure 8 SQL example

```sql
SELECT DISTINCT customers.name, customers.phone
FROM customers
JOIN orders ON orders.customer = customers.id
WHERE orders.amount > 100.00;
```

Source: Wladston Viana Ferreira Filho, "Computer science distilled: learn the art of solving computational problems", 2018

2.4.1 DDL – Data Definition Language

DDL's commands are used to create, edit, and delete database objects such as tables, views, schemas, domains, triggers, and stored procedures. The key words that fall into the DDL are CREATE, ALTER, and DROP.

2.4.2 DML – Data Manipulation Language

The language manipulation commands are used to view, add, edit, and delete data stored in database objects. Keywords belonging to DML are SELECT, INSERT, UPDATE, and DELETE.

2.4.3 DCL – Data Control Language

DCL's commands allow you to define access rights to database objects. The basic features include enabling and blocking access to GRANT and REVOKE commands. DCL also allows you to determine the level of access for individual users, such as rights to read or manipulate data.

2.5 Data replication

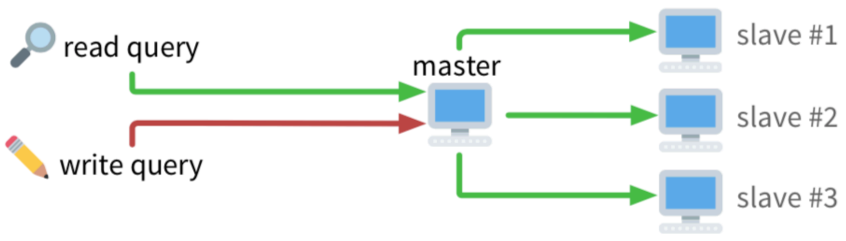
Data Replication is simply copying data from a database from one server to another so that all the users can share the same data without any inconsistency. As s result we get a distributed database where users can access data relevant to their tasks without interfering with the work of others.
Microsoft brings forward the term distributed query as "any SELECT, INSERT, UPDATE, or DELETE statement that references tables and rowsets from one or more external OLE DB data sources". (Microsoft, 2010)

Wladston Viana Ferreira Filho describes replication of data in his book "Computer science distilled: learn the art of solving computational problems", 2018. There are several cases when it can't be made without distributed database for instance database of hundreds of terabytes or system that processes thousands queries per second. Another good example that author gave is recording of altitude and speed of aircraft currently in a given airspace. Relying on a single computer might be too dangerous due to probability of crash the database becomes unavailable.

2.5.1 Single-Master replication

In Single-Master replication type only one main computer receives write queries to the database and then forwards them to connected so called slave computers which, at the meantime, have a replica of the database.

Figure 9 Single-Master Replication



Source: Wladston Viana Ferreira Filho, “Computer science distilled: learn the art of solving computational problems”, 2018

With this scheme, the master is able to proceed more read queries, because it can assign them to slaves. And the system becomes more reliable because if the master computer shuts down, the slaves can coordinate and elect a new master automatically, so the system doesn’t stop running.

2.5.2 Multi-Master

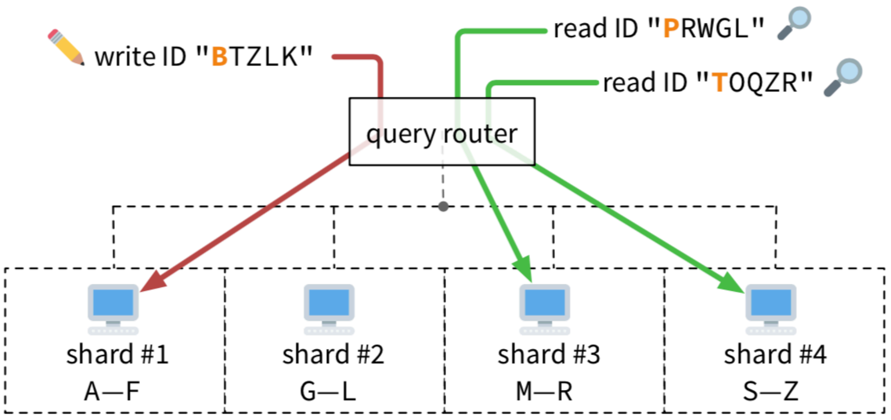
To handle a massive amount of concurrent write queries all computers in the cluster become masters. A load balancer distributes incoming read and write queries evenly among the machines. All the computers are connected to one another in the cluster so they stay synchronized and have a copy of the entire database.

2.5.3 Sharding

In case when a database receives a big number of write queries for large amounts of data, it’s getting problematic to synchronize the database everywhere in the cluster and some computers might simply not have enough storage space to accommodate everything. Partitioning the database among the computers is one of the solutions. Since each machine owns a part of the database, a query router forwards query to the relevant one:

This solution is good, but not perfect. If a machine in the cluster fails, the data it contains become unavailable. To mitigate that risk, sharding can be used with replication. It means that each of the computers have a master-slave cluster to prevent data loosing or system’s break downs.

Figure 10 Sharding setup



Source: Wladston Viana Ferreira Filho, “Computer science distilled: learn the art of solving computational problems”, 2018

Another common example is splitting a customer database geographically. Customers located on the East Coast can be placed on one server, while customers on the West Coast can be placed on a second server. Assuming there are no customers with multiple locations, the split is easy to maintain and build rules around.

2.6 NoSQL database

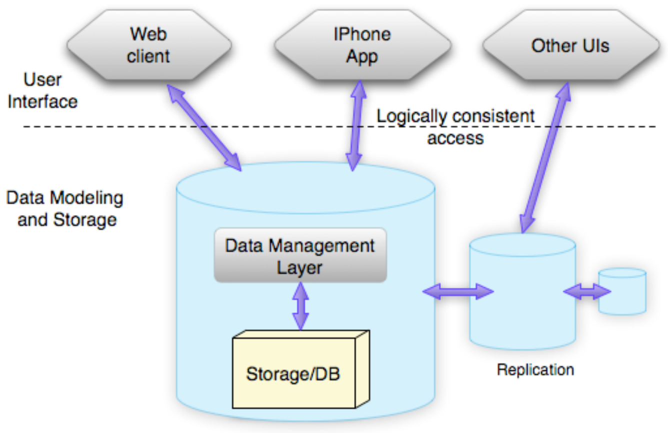
Daniel Power gives a good explanation to the term post-relational database in the article “What are post-relational databases?”. Post-relational databases implement a conceptual schema using a non-relational data model. A conceptual schema is a high-level description of a corporation's information needs. A conceptual schema can be mapped to various logical data models. When data is organised in a substantially different arrangement from the relational model of relations and primary and foreign keys, then the database can be classified as a post-relational database. Examples vary from multidimensional databases and object-relational databases to flat file databases. One of the advantages is that the databases of post-relational type are of the majority open sourced.

“NoSQLers came to share how they had overthrown the tyranny of slow, expensive relational databases in favour of more efficient and cheaper ways of managing data.”(Lai, 2009) Christof Strauch, a professor of Stuttgart Media University, wrote a study book for the students for the subject No-SQL databases in Media. According to his opinion, non-relational database does not chase table structure and, as it comes from its name, NoSQL database cover all the other types besides relational where obviously the language for manipulating with data is not SQL. The most popular NoSQL database systems include: MongoDB, Oracle NoSQL Database, CouchDB, Apache Cassandra, HBase, Amazon’s Dynamo, Project Voldemort and others. Google started developing BigTable in 2004. Facebook open sourced the Cassandra DB appeared in 2008. HBase is a BigTable clone for Hadoop. Even though it seems that NoSQL database gained its popularity in 2000’s, the origin of such model refers to 70’s.

According to Daniel Power in his article “What are post-relational databases?” tells that at first, the term itself was mostly associated with handling large data sets which are accessed and manipulated on the web vast. “Post-relational databases can store both "data in motion" and "data at rest", also called static data. Data in motion is collected and analysed in real-time. For example, data collected to analyse the quality of products during the manufacturing process is data in motion. An example of data at rest would be historical data about all aspects of the customer relationship, including sales transactions, social media data, and customer service interactions” (Hurwitz at.al, 2013). Post-relational model was grown to satisfy the needs of rapidly developing technologies of cloud computing enabling storing unstructured dynamic data, high volume, variety and velocity data (three V), collecting and processing in real time.

The NoSQL database approach is characterised by reduction of the complexity of SQL based servers. The logic is moved out from the database to a programmatic layer. It allows a developer to provide a stronger functionality of data management conveniently coded in a preferable powerful programming language instead of specific database's stored procedure language. NoSQL database focuses on higher performance and scalability, provides low-level access to a data management layer and frees from collection of complex APIs and languages in a SQL server (data column, queries, stored procedures, etc).

Figure 11 Architecture with NoSQL



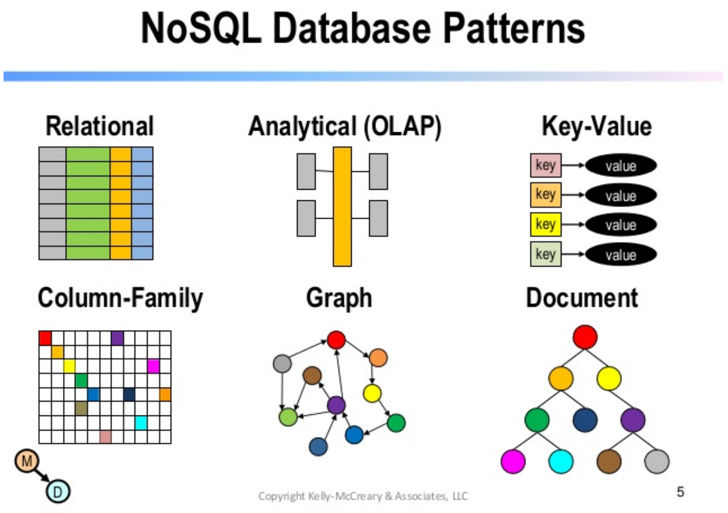
Source: https://www.sitepen.com/blog/nosql-architecture/, 2018

2.6.1 Types

No-SQL model defines a database as a collection of objects or reusable software elements, with related characteristics and methods. The object-oriented database model is the best known post-relational database model since it consolidates tables but isn't narrowed to them. Such models are known as hybrid database models as well.

There are several kinds of object-oriented databases: multimedia database absorbs media, such as for example images, that could not be stored in a relational database; hypertext database allows any object to link to any other object. It's useful for organizing lots of incommensurable data but is not very suitable for numerical analysis.

Figure 12 NoSQL database patterns



Source: https://nitendragautam.com/database/nosql-databases/, 2019

2.6.1.1 Key-Value store

Wladston Viana and Ferreira Filho included some information on key-value store in the book "Computer science distilled: learn the art of solving computational problems". The key-value database, also called key-value store and well-known as a dictionary or hash table. It is the least complex form of organised, constant data storage. The key in key/value pair is a unique value and can be easily searched for to access the data. There are two basic types of the database of this type – the first one keeps the data in memory and the second has the abilities of storing it to the disk.

One of the mainly spread use of key-value store is caching. When repeating a slow operation that always generates the same outcome, caching it might prove itself as highly useful. There is no compulsory need in using a key-value store, there are plenty other opportunities for storing the cache in alternative types of databases. Nevertheless, exclusively in case the cache is very frequently retrieved, the efficiency of key-value store systems becomes relevant. The main goal is reducing the input and output overhead. Typically, this data snapshot has limited durability, and when not needed, the cache will eventually expire.

"For example, when a user requests a specific web page to a server, the server must fetch the web page's data from the database and use it to render the HTML it will send to the user. In high-traffic websites, with thou- sands of concurrent accesses, doing that becomes impossible.

To solve this problem, we use a key-value store as a caching mechanism. The key is the requested URL, and the value is the final HTML of the corresponding web page. The next time someone asks for the same URL, the already generated HTML is simply retrieved from the key-value store using the URL as key." (Viana and Filho, 2018)

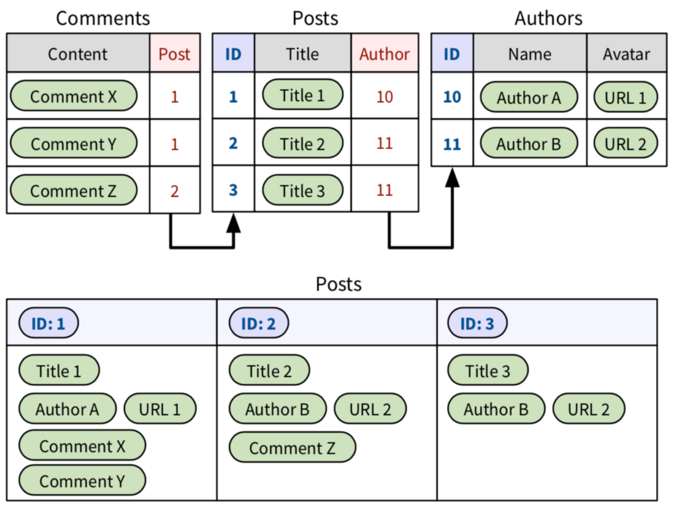
2.6.1.2 Document store

Redmond and Wilson in a year 2012 wrote a wonderful book called "Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement". As it comes from its name, authors describe different non-relational databases one by one. Speaking of document store we can say that it is one of the most popular version of NoSQL database is Document store. Data are stored according to the need of an application. It has good compatibility with data of JSON or XML formats. The architecture is easier to understand on the provided example of blog bellow. Data entries are called documents and remind relational raw and related documents together make up a collection. Each document must have unique primary key, as a result it is possible to make relationships between the documents, but JOINs are not optimal. However, NoSQL databases support indexes for primary keys.

Another difference is that it does not need to follow the structure of schema as in relational one, so it is very suitable for storing inconsistent data. On the picture we cannot but notice data duplicity which relational database seeks to avoid. Contra wise document store expects us to do so in every place where it is necessary. On the one hand, duplicated data is hard to update and keep consistent, while on the other hand, document store database proves to be very flexible.

Document Store database supports dynamic queries and defined indexes and also has good performance on big databases. It can easily find use in storing product, customer and order information, log data or user generated content for instance comments, ratings, chats.

Figure 13 Document store



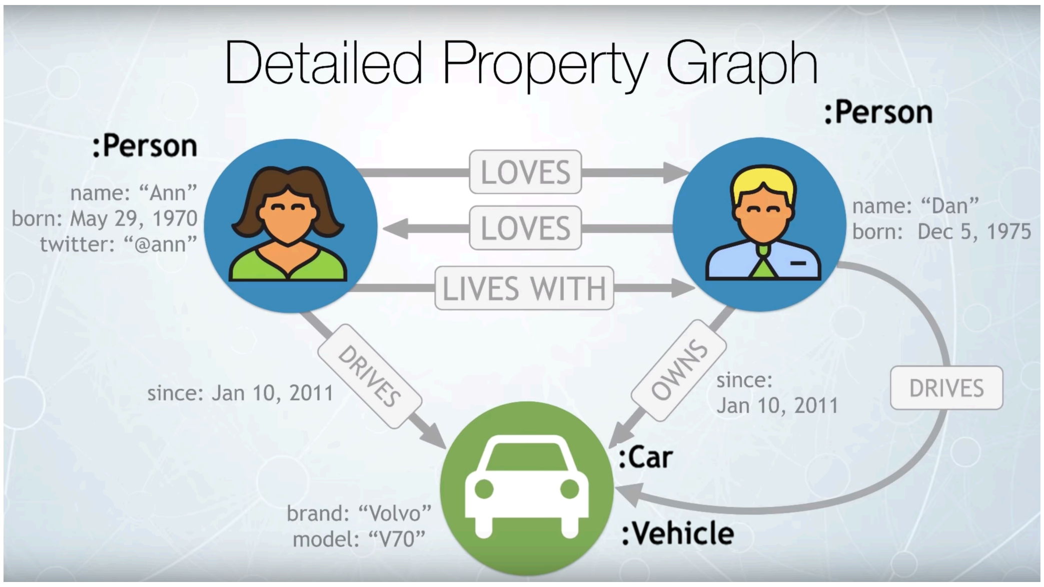
Source: Source: Wladston Viana Ferreira Filho, “Computer science distilled: learn the art of solving computational problems”, 2018

2.6.1.3 Graph

A graph database, also called a graph-oriented database, is a type of NoSQL database that stores, maps and queries relationships in form of graph. A graph database is essentially a set of nodes and edges. Every node represents an entity and edge represents a relationship among the nodes. Both node and edge have a unique identifier and a set of properties expressed as key/value pairs. At the meantime, edges have a starting-place and ending-place node, while nodes a have a list of outgoing and incoming edges.

The easiest way to understand the graph database theory is to imagine a map of public transport where nodes represent the stops and consequently edges or relationships between them are buses’ routes. This concept allows flexible data storage without being fixed or limited to schema. Moreover, using graph model makes it easier find the best connection between two stops even for people of not IT sphere.

Figure 14 Graph Neo4j model



Source: https://www.youtube.com/watch?v=NH6WoJHN4UA, 2019

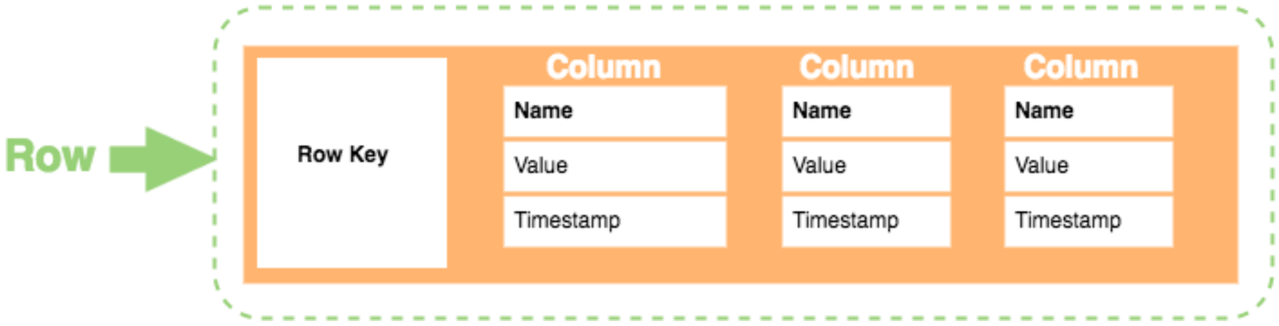
Graph databases proves itself to be sufficient for examining and evaluating interconnections, which leads to be an appropriate way for analysing data from social and physical networks, logistic and transportation. "Graph databases are also useful for working with data in business disciplines that involve complex relationships and dynamic schema, such as supply chain management, identifying the source of an IP telephony issue and creating "customers who bought this also looked at..." recommendations." (Rouse, 2016)

Speaking of last, advertisement platforms usually do not store relations between specific customers, instead they apply data mining approaches and choose the content for popping up advertisement with suggestions based on typical purchase sequences and associations, or in even more advanced tracking systems of user's browser history statistics.

2.6.1.4 Column Family

In the article "Bigtable: A Distributed Storage System for Structured Data" Fay Chang and others touched the topic of column family database. The database of column-family type stores and processes data by columns instead of rows. Speaking more detailed, a key identifies a row where data is stored in one or more column families (they remind relational table), which, in the meantime, can consists of several columns so it means that the columns in one row don't have to match columns in another row. Each value in each column has its own timestamp as sort of column's own identifier. It is very suitable for storing for example historical data. The model generally supports sparse column allocation since row IDs without certain columns do not need an explicit NULL value for those columns. On the other hand, columns which have few or no NULL values must still store the column key in each row, which leads to greater space consumption. However, querying is extremely fast and gives the opportunity to work with the results straight forward.

Figure 15 Column Family row example



Source: https://database.guide/what-is-a-column-store-database/, 2019

Christopher Strauch in his book "NoSQL databases" also gives a very good explanation of column family database. The main influence on column-family database has Google's BigTable project. Hbase and Cassandra are modelled after BigTable but Hbase is made specially for Hadoop. Another very popular column database is Amazon's Dynamo. Speaking of the origin for column database we can say that BigTable is "a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers" (Fay, 2009). A lot of Google's products such as Google Earth, Google Analytics Google Docs and others are using Bigtable as their database.

2.6.1.5 XML database

Ramez Elmasri and Shamkant B. Navathe talk about XML database in their book "Fundamentals of database systems (6th edition). The first reference to XML takes us to 70's-80's which also happens to be the rise of the Internet. Since the first database model was relational, huge amount of information in the XML format was also converted to the appropriate format and stored in relational database. Nevertheless, it would make more sense to store XML as XML format database directly and now it is possible using XML-enabled and Native XML (NXD). The data stored in the database are queried using XQuery. The XML language is in itself an object-based language that allows users to take advantage of inherent XML organizational structures. IT professionals may refer to an XML database as a "native XML database" if it provides direct XML storage.

As it was mentioned before, there are two major types of XML databases − XML- enabled Native XML (NXD). XML - Enabled Database XML is nothing but the extension provided for the conversion of XML document. This is a classic relational database, where data is stored in tables consisting of rows and columns. Native XML database is based on the container rather than table format. It can store large amount of XML document and data. Native XML database is queried by the XPath.

3 Practical Part

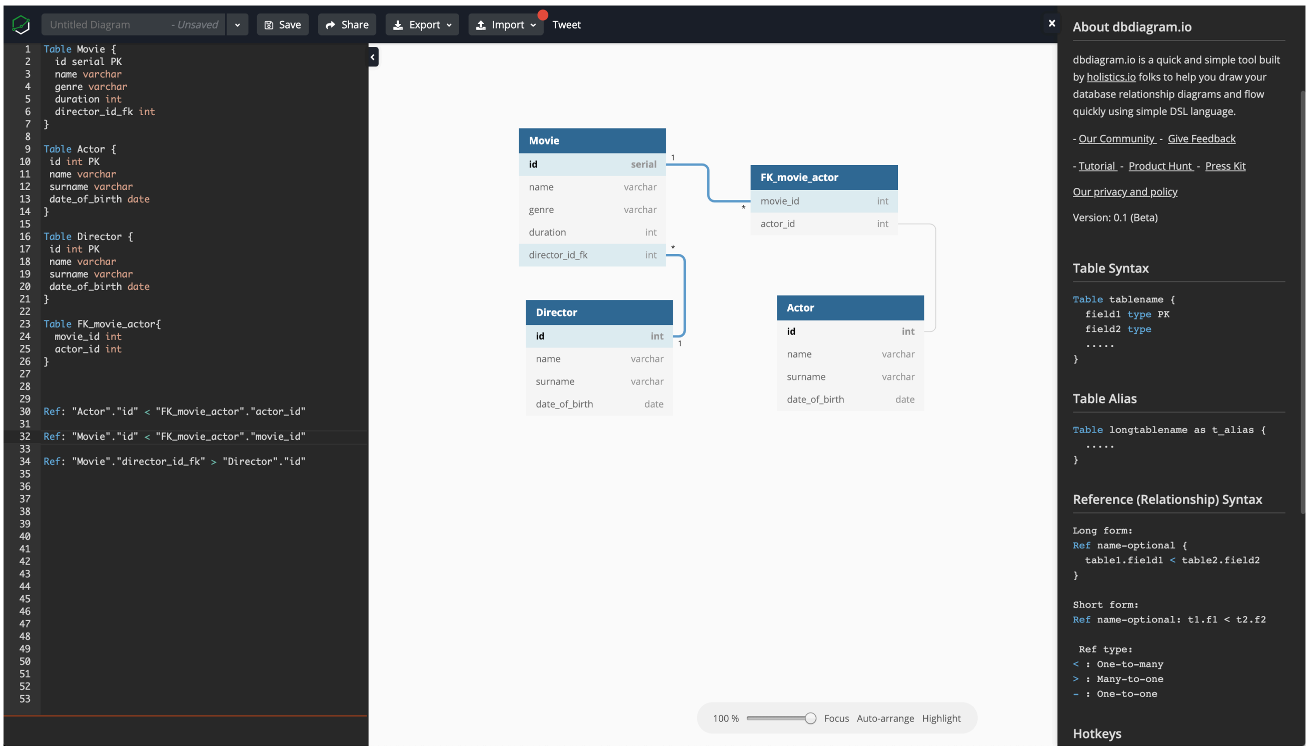
3.1 Implementation of the database

I decided to create a small well-arranged transparent database to easily see any differences between the two models and their implementation using different appropriate language for that purpose. I would like to compare not only the database structure but the model as well.

3.1.1 Database model

I created the diagram using a smooth tool dbdiagram.io. They have a simple language for creating tables and referencing one to another.

Figure 16 Database model for practical part in application (own work)



```
Table Movie {
  id serial PK
  name varchar
  genre varchar
  duration int
  director_id_fk int
}

Table Actor {
 id int PK
 name varchar
 surname varchar
 date_of_birth date
}

Table Director {
 id int PK
 name varchar
 surname varchar
 date_of_birth date
}

Table FK_movie_actor{
  movie_id int
  actor_id int
}



Ref: "Actor"."id" < "FK_movie_actor"."actor_id"

Ref: "Movie"."id" < "FK_movie_actor"."movie_id"

Ref: "Movie"."director_id_fk" > "Director"."id"
```

Movie	
id	serial
name	varchar
genre	varchar
duration	int
director_id_fk	int

FK_movie_actor	
movie_id	int
actor_id	int

Director	
id	int
name	varchar
surname	varchar
date_of_birth	date

Actor	
id	int
name	varchar
surname	varchar
date_of_birth	date

About dbdiagram.io

dbdiagram.io is a quick and simple tool built by holistics.io folks to help you draw your database relationship diagrams and flow quickly using simple DSL language.

- Our Community - Give Feedback

- Tutorial - Product Hunt - Press Kit

Our privacy and policy

Version: 0.1 (Beta)

Table Syntax

```
Table tablename {
  field1 type PK
  field2 type
  .....
}
```

Table Alias

```
Table longtablename as t_alias {
  .....
}
```

Reference (Relationship) Syntax

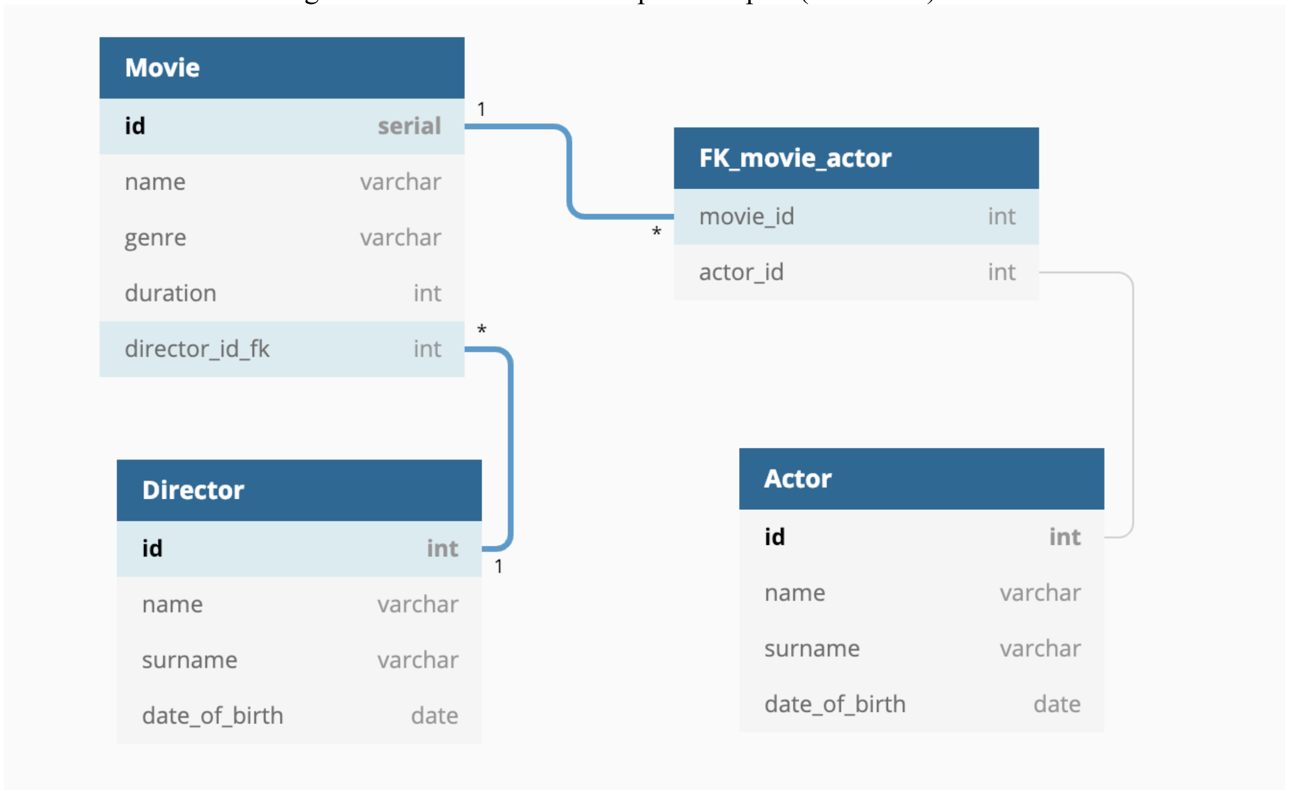
```
Long form:
Ref name-optional {
  table1.field1 < table2.field2
}

Short form:
Ref name-optional: t1.f1 < t2.f2

 Ref type:
< : One-to-many
> : Many-to-one
- : One-to-one
```

Hotkeys

Source: https://dbdiagram.io/d/5c842976f7c5bb70c72f3ef6, 2019

Moving to the model itself we can see that is crated for RDBMS. It consists of 3 main relations: Movie, Director and Actor and 1 relation that serves as junction relation for storing primary keys of related tables of Many-to-Many type. Each of them has unique identifier of the numeric type and some basic attributes describing the table. The structure of the tables Director and Actor is very similar since it describes the same object – person, but of different role. The main relation is Movie and it contains the column for primary key of related table Director in One-to-Many relationship type. But we cannot see the same otherwise: Director table does not have the column with Movie's id – in One-to Many relationships we store only one foreign key on the "many side".

Figure 17 Database model for practical part (own work)

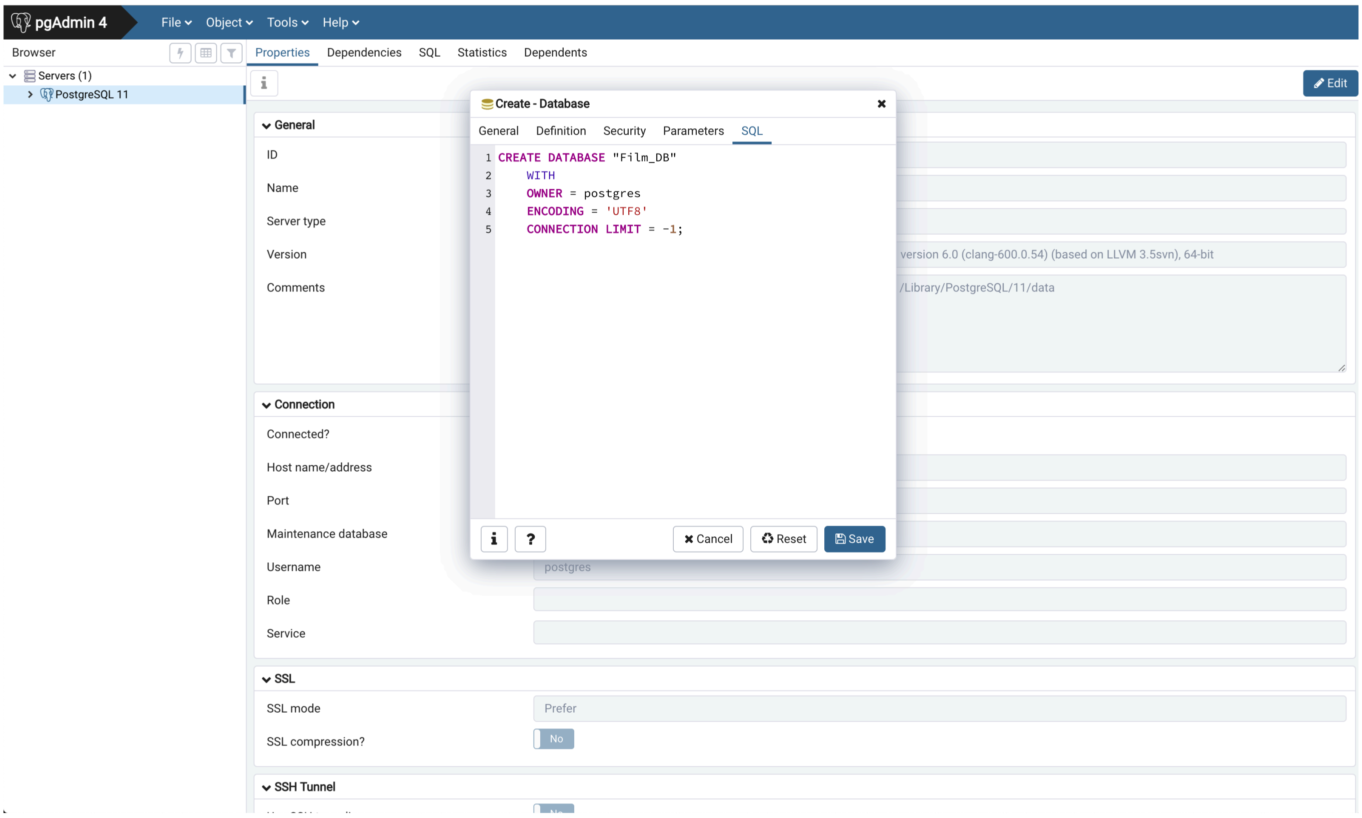


Source: own work, 2019

3.1.2 Implementation to the object-relational PostgreSQL database

For relational database I decided to use open source PostgreSQL. Even though it claims to be object-relational database, we can still use it just as classic relational DB. PostgreSQL uses and extends the SQL combined with many features that store and scale the most complicated data workloads. It has a huge range of data types such as primitives: Integer, Numeric, String, Boolean; structured: Date/Time, Array, Range, UUID; document: JSON/JSONB, XML, Key-value (Hstore); geometry: Point, Line, Circle, Polygon; customizations: Composite, Custom Types and, in addition, there is an opportunity to define own custom data type. Regarding the procedural language or SQL, PostgreSQL tries to conform with the SQL standard. Many of the features required by the SQL standard are supported, though sometimes with slightly differing syntax or function. As of the version 11 release in October 2018, PostgreSQL conforms to at least 160 of the 179 mandatory features for SQL:2011 Core conformance.

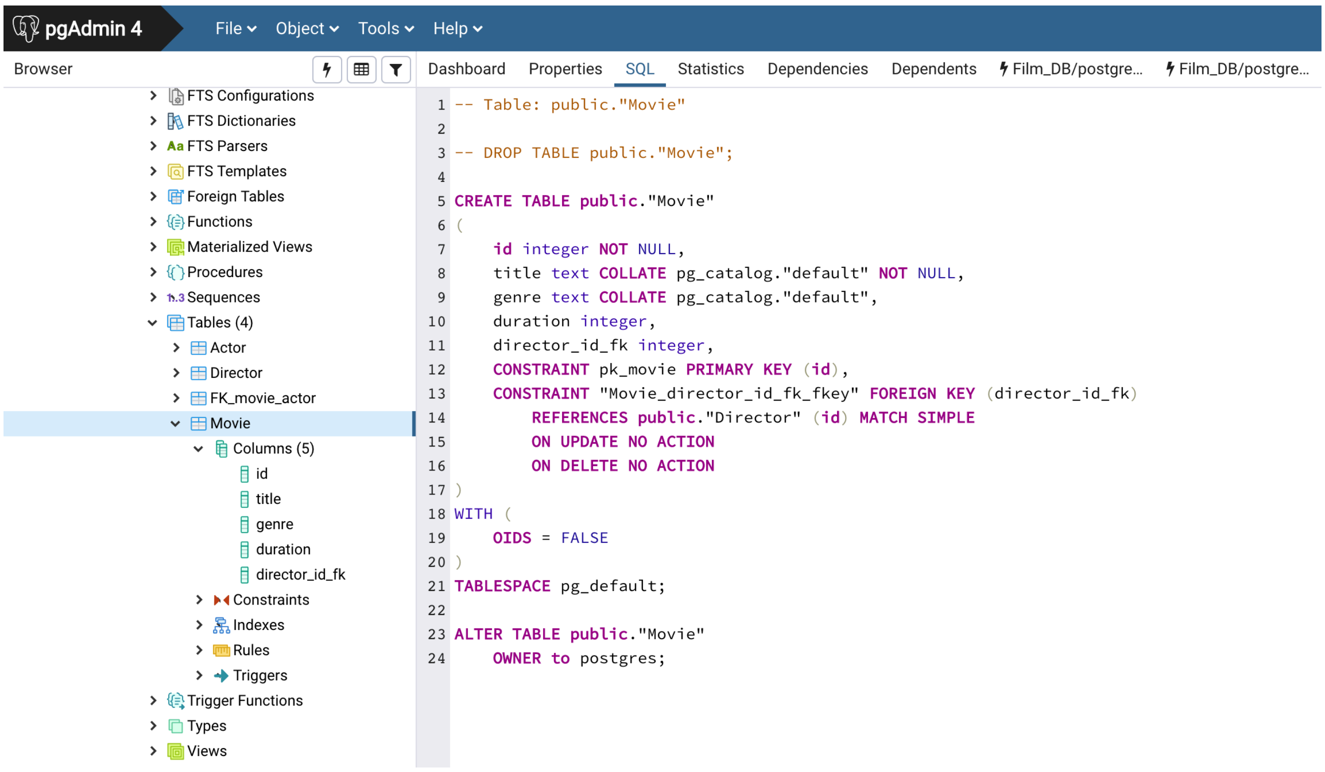
First, we create a database. I used GUI but, of course, there is a shall version as well.



```
CREATE DATABASE "Film_DB"
    WITH
    OWNER = postgres
    ENCODING = 'UTF8'
    CONNECTION LIMIT = -1;
```

ID	
Name	
Server type	
Version	version 6.0 (clang-600.0.54) (based on LLVM 3.5svn), 64-bit
Comments	/Library/PostgreSQL/11/data
Connected?	
Host name/address	
Port	
Maintenance database	
Username	postgres
Role	
Service	
SSL mode	Prefer
SSL compression?	No

This is how it looks when you already created a table and just see the SQL code it. I created the "Movie" table using GUI.

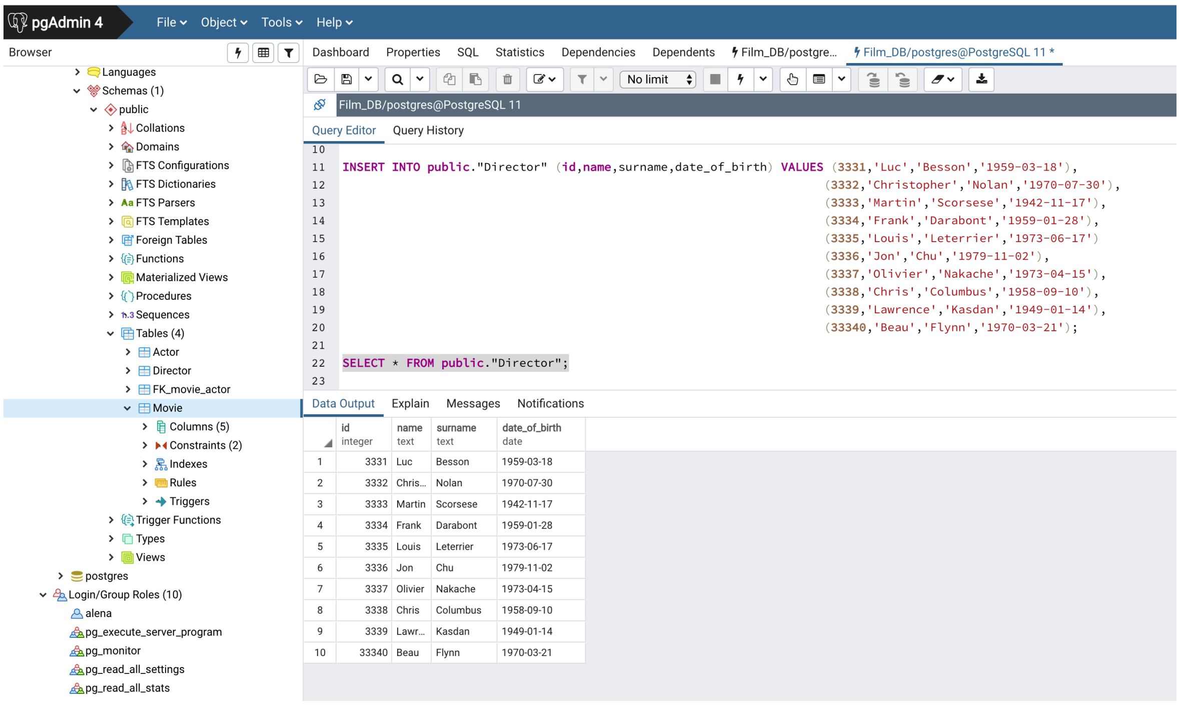


```
-- Table: public."Movie"

-- DROP TABLE public."Movie";

CREATE TABLE public."Movie"
(
    id integer NOT NULL,
    title text COLLATE pg_catalog."default" NOT NULL,
    genre text COLLATE pg_catalog."default",
    duration integer,
    director_id_fk integer,
    CONSTRAINT pk_movie PRIMARY KEY (id),
    CONSTRAINT "Movie_director_id_fk_fkey" FOREIGN KEY (director_id_fk)
        REFERENCES public."Director" (id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
WITH (
    OIDS = FALSE
)
TABLESPACE pg_default;

ALTER TABLE public."Movie"
    OWNER to postgres;
```

Other tables I created manually. We can see the difference that the name of the relation is captured in quotes. I assume it is due to the fact that it is OBJECT-relational DBS. There is no mentions concerning that in the internet though.

```sql
CREATE TABLE public."Director"
(
    id integer NOT NULL,
    name text COLLATE pg_catalog."default" NOT NULL,
    surname text COLLATE pg_catalog."default" NOT NULL,
    date_of_birth date,
    CONSTRAINT "Director_pkey" PRIMARY KEY (id)
);

CREATE TABLE public."Actor"
(
    id integer NOT NULL,
    name text COLLATE pg_catalog."default" NOT NULL,
    surname text COLLATE pg_catalog."default" NOT NULL,
    date_of_birth date,
    CONSTRAINT prim_key_actor PRIMARY KEY (id)
);

CREATE TABLE "FK_movie_actor" (
  "movie_id" integer REFERENCES "Movie"(id),
  "actor_id" integer REFERENCES "Actor"(id),
  CONSTRAINT id_relation_table PRIMARY KEY (movie_id, actor_id)
);
```

Then I inserted some data into the database.



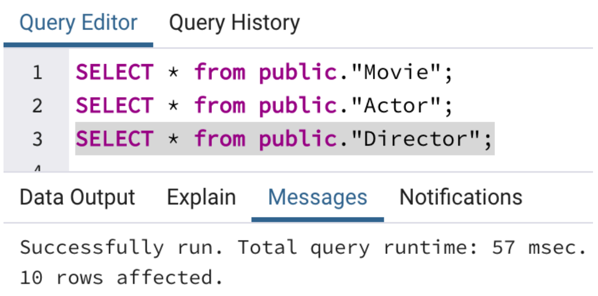
```sql
INSERT INTO public."Director" (id,name,surname,date_of_birth) VALUES (3331,'Luc','Besson','1959-03-18'),
                                                                     (3332,'Christopher','Nolan','1970-07-30'),
                                                                     (3333,'Martin','Scorsese','1942-11-17'),
                                                                     (3334,'Frank','Darabont','1959-01-28'),
                                                                     (3335,'Louis','Leterrier','1973-06-17')
                                                                     (3336,'Jon','Chu','1979-11-02'),
                                                                     (3337,'Olivier','Nakache','1973-04-15'),
                                                                     (3338,'Chris','Columbus','1958-09-10'),
                                                                     (3339,'Lawrence','Kasdan','1949-01-14'),
                                                                     (33340,'Beau','Flynn','1970-03-21');

SELECT * FROM public."Director";
```

	id integer	name text	surname text	date_of_birth date
1	3331	Luc	Besson	1959-03-18
2	3332	Chris...	Nolan	1970-07-30
3	3333	Martin	Scorsese	1942-11-17
4	3334	Frank	Darabont	1959-01-28
5	3335	Louis	Leterrier	1973-06-17
6	3336	Jon	Chu	1979-11-02
7	3337	Olivier	Nakache	1973-04-15
8	3338	Chris	Columbus	1958-09-10
9	3339	Lawr...	Kasdan	1949-01-14
10	33340	Beau	Flynn	1970-03-21

```sql
INSERT INTO public."Movie" VALUES (110,'Leon','thriller',133,3331),
                                  (111,'The Shawshank Redemption','drama',142,3334),
                                  (112,'Shutter Island','detective',138,3333),
                                  (113,'The Green Mile','drama',189,3334),
                                  (114,'Interstellar','fantasy',169,3332),
                                  (115,'Inception','fantasy',148,3332),
                                  (116,'The 5th Element','fantasy',126,3331),
                                  (117,'Now you see me','criminal',115,3335),
                                  (118,'Now you see me 2','comedy',129,3336),
                                  (119,'Intouchables','melodrama',112,3337);

INSERT INTO public."Movie" VALUES (120,'Fight club','thriller',131),
                                  (121,'The notebook','drama',124),
                                  (122,'Crazy stupid love','melodrama',118),
                                  (123,'Home Alone','family',103),
                                  (124,'The social network','biography',120),
                                  (125,'Harry Potter and the Prisoner of Azkaban','fantasy',142);


INSERT INTO public."Actor" VALUES (2221,'Jean','Reno','1948-07-30'),
                                  (2222,'Gary','Oldman','1958-03-21'),
                                  (2223,'Natalie','Portman','1981-07-09'),
                                  (2224,'Bruce','Willis','1955-03-19'),
                                  (2226,'Leonardo','DiCaprio','1974-11-11'),
                                  (2227,'Joseph','Gordon-Levitt','1981-12-17'),
                                  (2228,'Matthew','McConaughey','1969-11-04'),
                                  (2229,'Michael','Caine','1933-03-14'),
                                  (2230,'Mark','Ruffalo','1967-11-22'),
                                  (2231,'Tim','Robbins','1958-10-16'),
                                  (2232,'Tom','Hanks','1956-07-09'),
                                  (2233,'Morgan','Freeman','1937-07-01'),
                                  (2234,'Jesse','Eisenberg','1983-10-05'),
                                  (2235,'Daniel','Radcliffe','1989-07-23'),
                                  (2236,'Edward','Norton','1969-08-18'),
                                  (2237,'Brad','Pitt','1963-12-18');


INSERT INTO public."FK_movie_actor" VALUES (110,2221), (110,2222),
                                           (110,2223), (111,2231),
                                           (111,2233), (112,2230),
                                           (112,2226), (113,2232),
                                           (114,2228), (114,2229),
                                           (115,2226), (115,2227),
                                           (116,2222), (116,2224),
                                           (117,2233), (117,2229),
                                           (118,2234), (118,2235),
                                           (120,2236), (120,2237),
                                           (124,2234), (125,2235);
```

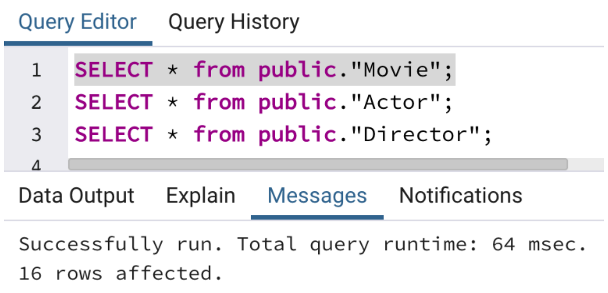
3.1.2.1 Simple SELECT runtime

In the following steps I will run similar SELECTs but for different relations: Movie-Director is 1-N and Movie-Actor is M-N. and we will see the outcomes and compare running time. First example shows us the content of just one table and the results are very similar. The Movie relation contain a bit more rows than Director one so it explains a difference. For higher accuracy I ran the SELECT several times and chose the estimate mean of all.



```sql
SELECT * from public."Movie";
SELECT * from public."Actor";
SELECT * from public."Director";
```

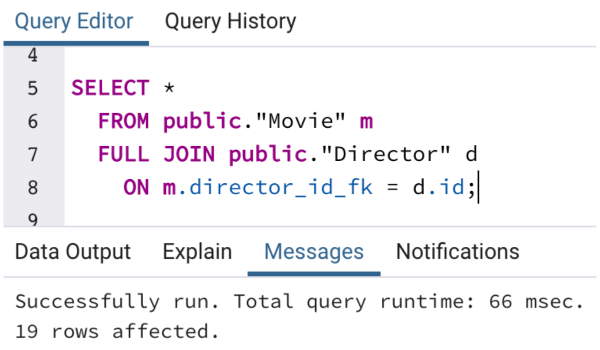
```
Successfully run. Total query runtime: 57 msec.
10 rows affected.
```



```sql
SELECT * from public."Movie";
SELECT * from public."Actor";
SELECT * from public."Director";
```

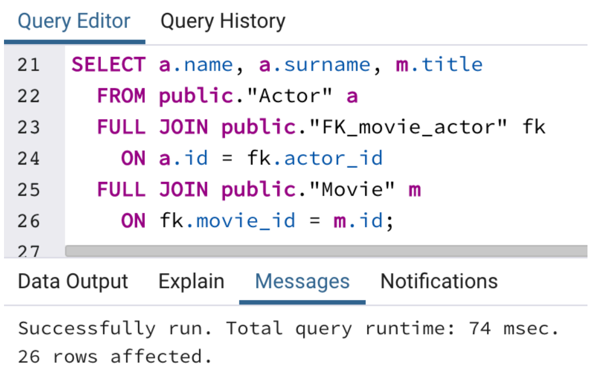
```
Successfully run. Total query runtime: 64 msec.
16 rows affected.
```

Here we can notice that I used FULL JOIN. It includes all the records from both of the tables not only matching records. On the left-hand side there are performed operations with Movie-Director 1-M and on the right-hand side Movie-Actor M-N. We cannot but notice that runtime is greater by a long chalk. But let's have a look on the next results.



```sql
SELECT *
  FROM public."Movie" m
  FULL JOIN public."Director" d
    ON m.director_id_fk = d.id;
```

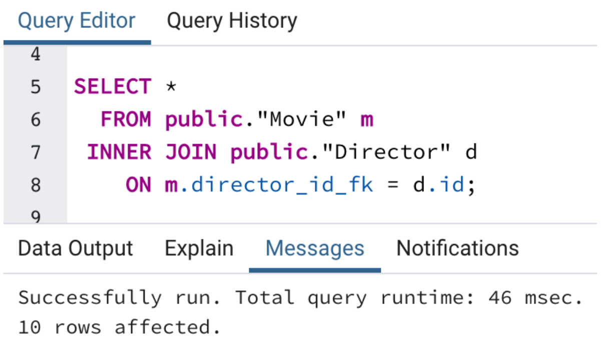
```
Successfully run. Total query runtime: 66 msec.
19 rows affected.
```



```sql
SELECT a.name, a.surname, m.title
  FROM public."Actor" a
  FULL JOIN public."FK_movie_actor" fk
    ON a.id = fk.actor_id
  FULL JOIN public."Movie" m
    ON fk.movie_id = m.id;
```

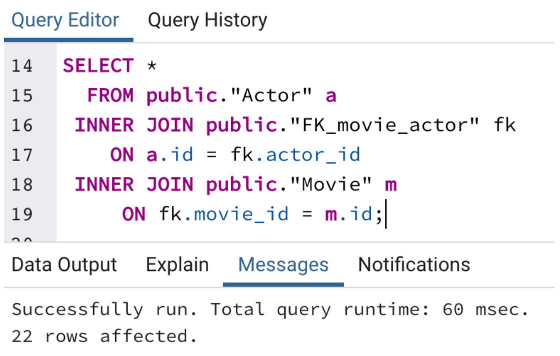
```
Successfully run. Total query runtime: 74 msec.
26 rows affected.
```

For the next query I decided to use INNER JOIN. This gets us all records that are common between both tables based on the foreign key. Even though I only put data that are matching, the results are not identical in comparison to other queries. The runtime is less than the runtime of retrieving data from a not joined table.



```
SELECT *
  FROM public."Movie" m
 INNER JOIN public."Director" d
    ON m.director_id_fk = d.id;
```

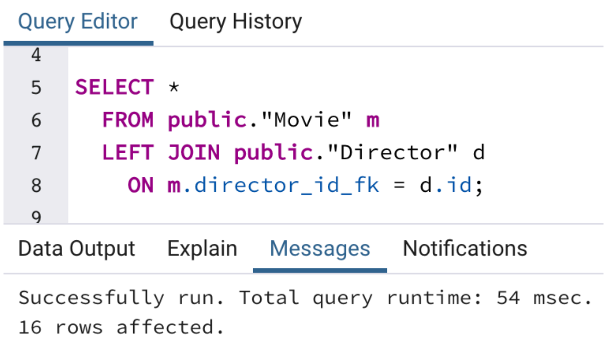
```
Successfully run. Total query runtime: 46 msec.
10 rows affected.
```



```
SELECT *
  FROM public."Actor" a
 INNER JOIN public."FK_movie_actor" fk
    ON a.id = fk.actor_id
 INNER JOIN public."Movie" m
    ON fk.movie_id = m.id;
```

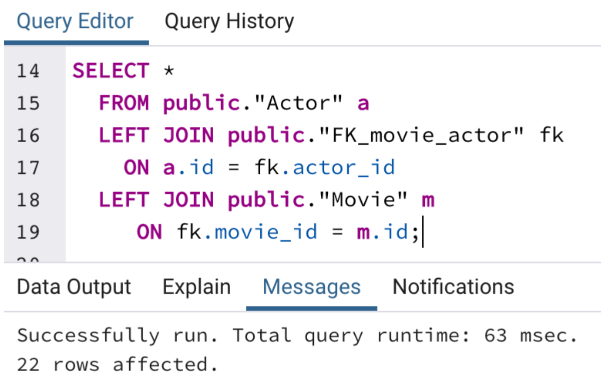
```
Successfully run. Total query runtime: 60 msec.
22 rows affected.
```

It’s time for LEFT JOIN. It gives us all the record from the main table plus common selected rows in the second table. As we can see, LEFT JOIN needs more time because it has to do all the work PLUS the extra work of null - extending the results. It would also be expected to return more rows, further increasing the total execution time simply due to the larger size of the result set.



```
SELECT *
  FROM public."Movie" m
  LEFT JOIN public."Director" d
    ON m.director_id_fk = d.id;
```

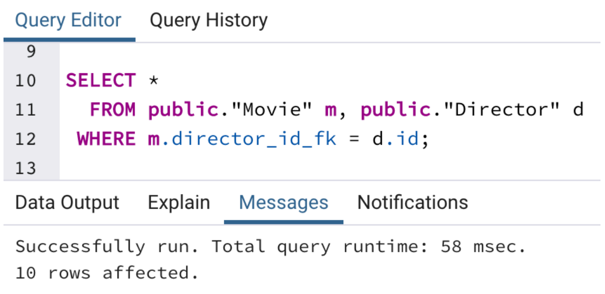
```
Successfully run. Total query runtime: 54 msec.
16 rows affected.
```



```
SELECT *
  FROM public."Actor" a
  LEFT JOIN public."FK_movie_actor" fk
    ON a.id = fk.actor_id
  LEFT JOIN public."Movie" m
    ON fk.movie_id = m.id;
```

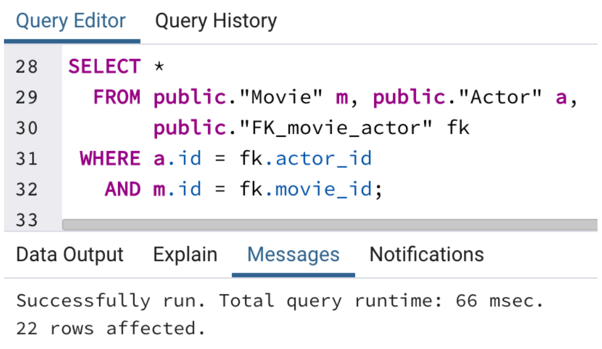
```
Successfully run. Total query runtime: 63 msec.
22 rows affected.
```

Last but not least, I tried to retrieve data instead of joining table just using the condition clause and equating the foreign keys. As expected, the runtime was higher.



```
SELECT *
  FROM public."Movie" m, public."Director" d
 WHERE m.director_id_fk = d.id;
```

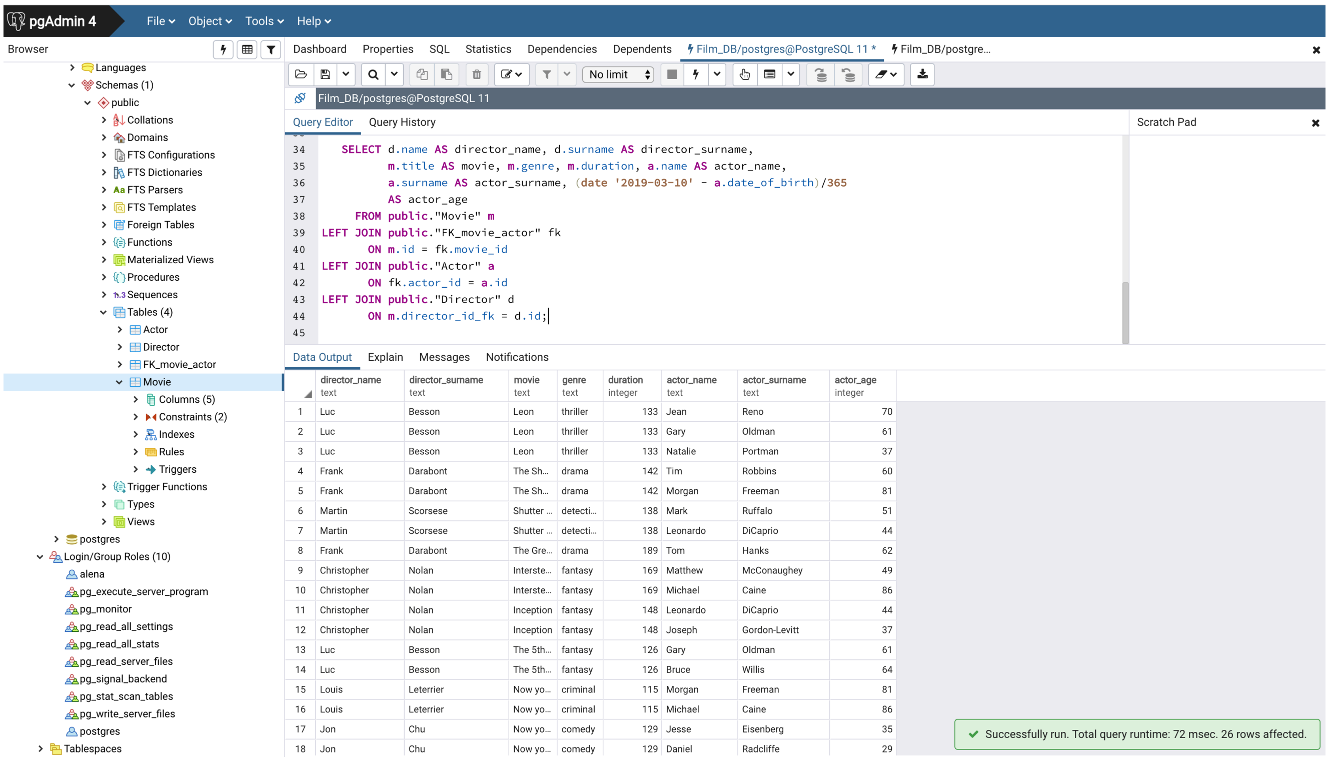
```
Successfully run. Total query runtime: 58 msec.
10 rows affected.
```



```
SELECT *
  FROM public."Movie" m, public."Actor" a,
       public."FK_movie_actor" fk
 WHERE a.id = fk.actor_id
   AND m.id = fk.movie_id;
```

```
Successfully run. Total query runtime: 66 msec.
22 rows affected.
```

On the screenshot bellow you can see the results of a simple query with some calculations. The runtime is 72 msec.



```
   SELECT d.name AS director_name, d.surname AS director_surname,
          m.title AS movie, m.genre, m.duration, a.name AS actor_name,
          a.surname AS actor_surname, (date '2019-03-10' - a.date_of_birth)/365
          AS actor_age
     FROM public."Movie" m
LEFT JOIN public."FK_movie_actor" fk
       ON m.id = fk.movie_id
LEFT JOIN public."Actor" a
       ON fk.actor_id = a.id
LEFT JOIN public."Director" d
       ON m.director_id_fk = d.id;
```

	director_name text	director_surname text	movie text	genre text	duration integer	actor_name text	actor_surname text	actor_age integer
1	Luc	Besson	Leon	thriller	133	Jean	Reno	70
2	Luc	Besson	Leon	thriller	133	Gary	Oldman	61
3	Luc	Besson	Leon	thriller	133	Natalie	Portman	37
4	Frank	Darabont	The Sh...	drama	142	Tim	Robbins	60
5	Frank	Darabont	The Sh...	drama	142	Morgan	Freeman	81
6	Martin	Scorsese	Shutter ...	detecti...	138	Mark	Ruffalo	51
7	Martin	Scorsese	Shutter ...	detecti...	138	Leonardo	DiCaprio	44
8	Frank	Darabont	The Gre...	drama	189	Tom	Hanks	62
9	Christopher	Nolan	Interste...	fantasy	169	Matthew	McConaughey	49
10	Christopher	Nolan	Interste...	fantasy	169	Michael	Caine	86
11	Christopher	Nolan	Inception	fantasy	148	Leonardo	DiCaprio	44
12	Christopher	Nolan	Inception	fantasy	148	Joseph	Gordon-Levitt	37
13	Luc	Besson	The 5th...	fantasy	126	Gary	Oldman	61
14	Luc	Besson	The 5th...	fantasy	126	Bruce	Willis	64
15	Louis	Leterrier	Now yo...	criminal	115	Morgan	Freeman	81
16	Louis	Leterrier	Now yo...	criminal	115	Michael	Caine	86
17	Jon	Chu	Now yo...	comedy	129	Jesse	Eisenberg	35
18	Jon	Chu	Now yo...	comedy	129	Daniel	Radcliffe	29

```
Successfully run. Total query runtime: 72 msec. 26 rows affected.
```

3.1.3 Implementation to the Neo4j graph database

For post-relational databased I chose Neo4j Graph Database (pronounced neo-four-j). Developers claim it to be TRULY relational DB. This database does not need to have predefined model – the data are stored the same way as you would draw on the paper, thus it results to an extremely easy modeling and storing the relationships. Only one database can run at a time. Even though it is NoSQL database, it provides an ACID-compliant transactional backend for applications, which is not very common in non-relational databases. Performance of relationship intersections remains constant with grow in data size due to using so called index-free adjacency. Another very useful feature is that we can add any additional property to anything (even relationships) and it does not require migration. Neo4j uses its own free Cypher project as its graph native language. Some of the further graph database use Cypher as its query language as well. Shortly, Cypher is a replicant od SQL but costumed for graph database. It goes without mentioning that work with graph DB is very flexible and highly convenient allowing to freely work with data with no limitations and at the meantime the data are very well organized.

Moving on to the implementation of the same model to Neo4j. I implement the exact same model as I had for relational database not taking advantage of interesting opportunities of the DBS. First there are performed some simple CREATE statement for one record already filled with the data:

```
$ CREATE (:Movie {title:"Home Alone", genre:"family", duration:103})
```

Then here we can notice that I created two instances with attributes and the relationship with direction between them in one step at the same time:

```
$ CREATE (:Director {name:'Oliver', surname:'Nakache',
  date_of_birth:'date('1979-01-01')'}) -[:controlls]-> (:Movie
  {title:'Intouchables', genre:'melodrama', duration:112})
```

Creating the whole "record" at once:

```
$ CREATE (:Director {name:"Luc", surname:"Besson",
  date_of_birth:"date('1959-03181')"}) -[:controlls]-> (:Movie
  {title:"Leon", genre:"thriller", duration:'133'}) <-[:plays_in]- (:Actor
  {name:"Jean", surname:"Reno", date_of_birth:"date('1948-0730')"})
```

```
Added 3 labels, created 3 nodes, set 9 properties, created 2 relationships, completed after 2 ms.
```

Here is an interesting example of creating one Movie instance and straight relating 2 Actor instances in just one step. If there exist a director for that film a will make a relationship between them separately.

```
$ CREATE (:Actor {name:"Edward", surname:"Norton",
  date_of_birth:1969-8-18})-[:plays_in]->(:Movie {title:'Fight club',
  genre:'thriller', duration:'131'})<-[:plays_in]-(:Actor
  {name:'Brad', surname:'Pitt', date_of_birth:1963-12-18})
```

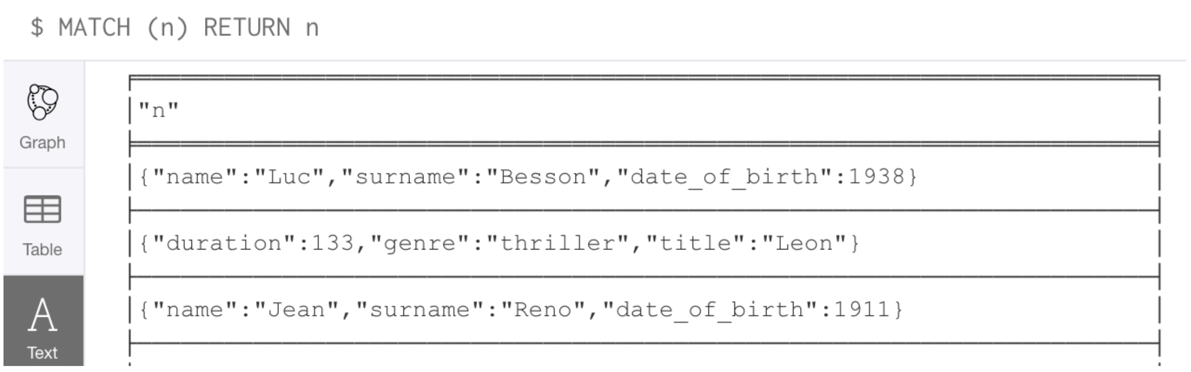
The following examples shows only creating the relationships between already existing instance and creating the connected instance at the same time. First, we have to find the needed instance using MATCH and set it to variable and then use it to store the relationship. There are shown two ways of matching the desired instance with defining the properties directly and using WHERE clause.

```
MATCH (m :Movie{title:'Leon'})
 CREATE (:Actor {name:"Gary", surname:"Oldman", date_of_birth:1958-
03-21})-[:plays_in]->(m)
```

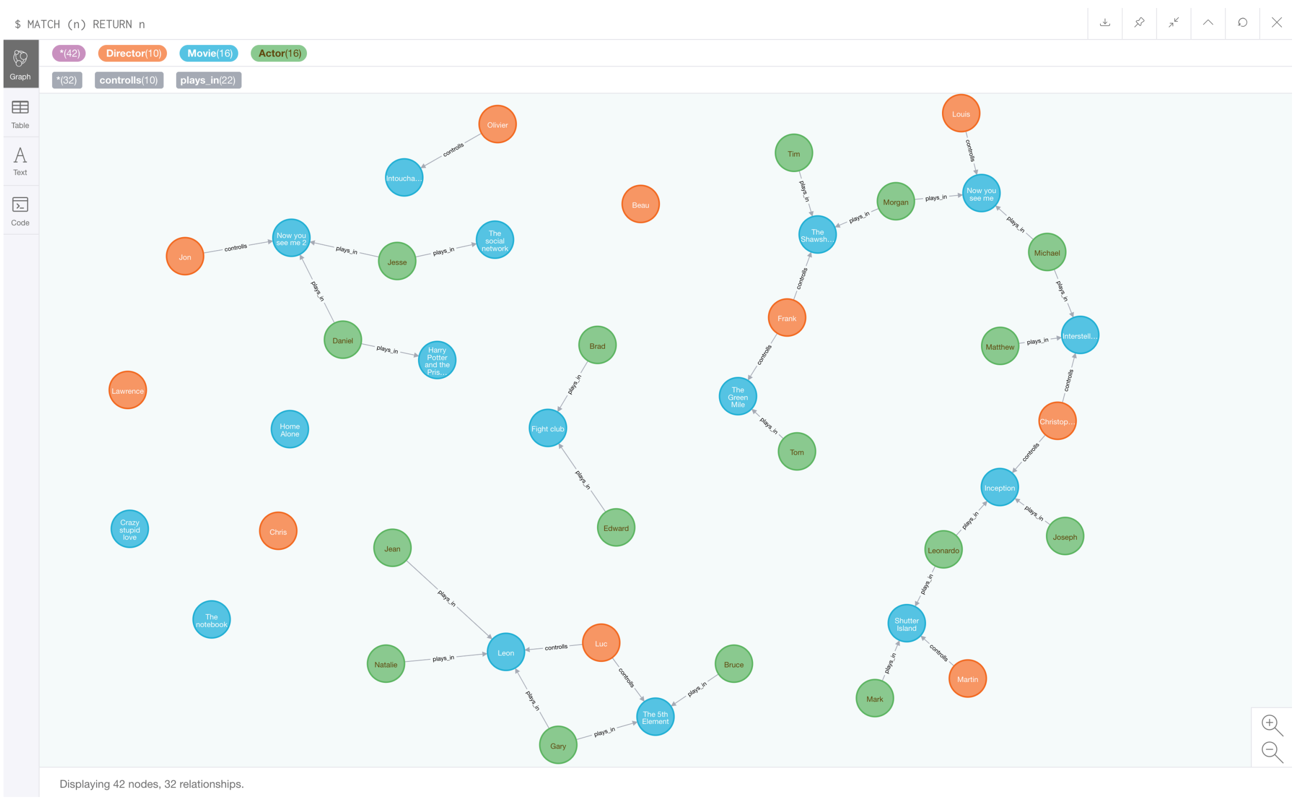
```
MATCH (a :Actor {name:'Daniel'})
 CREATE (a) -[:plays_in]-> (:Movie {title:"Harry Potter and the
Prisoner of Azkaban", genre:"fantasy", duration:142})
```

```
MATCH (a:Actor)
WHERE a.surname = 'Freeman'
CREATE (:Director {name:'Louis', surname:'Leterrier',
date_of_birth:1973-06-17})-[:controlls]->(:Movie {title:'Now you
see me', genre:'criminal', duration:115})<-[:plays_in]-(a)
```

This are some examples of the way how I created and implemented the whole model. Only after these steps we can see our data model. At the beginning we see all the instances distributed chaotically. We always can relocate them the way we want it to be. The label Movie is defined with a blue colour, Directors with green and Actors have orange colour. To see the properties, we just click on the instance. Even though it is a graph database, there is an opportunity to display the data in the table and text format.



```
$ MATCH (n) RETURN n

╒══════════════════════════════════════════════════════════╕
│"n"                                                       │
╞══════════════════════════════════════════════════════════╡
│{"name":"Luc","surname":"Besson","date_of_birth":1938}    │
├──────────────────────────────────────────────────────────┤
│{"duration":133,"genre":"thriller","title":"Leon"}        │
├──────────────────────────────────────────────────────────┤
│{"name":"Jean","surname":"Reno","date_of_birth":1911}     │
├──────────────────────────────────────────────────────────┤
```

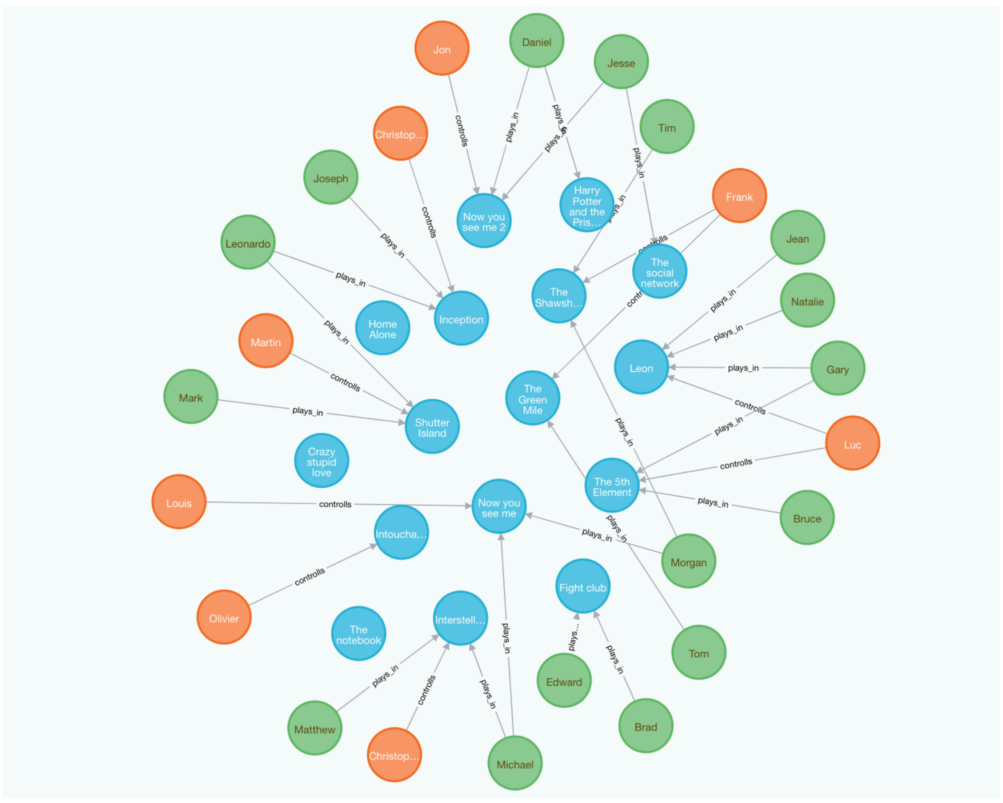


```
$ MATCH (n) RETURN n
```

*(42) Director(10) Movie(16) Actor(16)

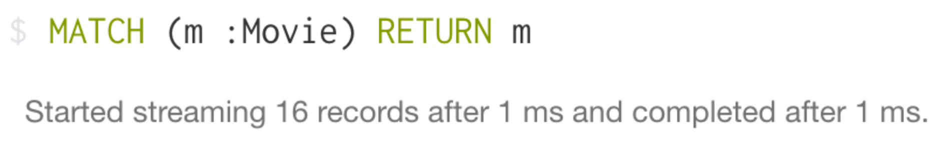
*(32) controlls(10) plays_in(22)

Displaying 42 nodes, 32 relationships.



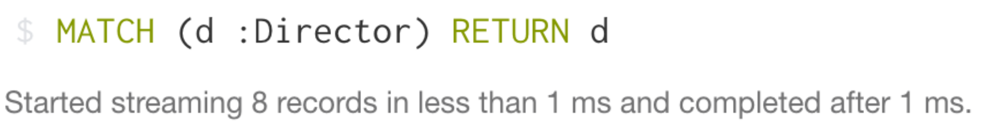
3.1.3.1 Simple MATCH runtime

I tried to retrieve the same data as I did in Postgres: Movie, Directors, Movies and Directors, Movies and Actors. In comparison to previous outcomes are overwhelming. As simple as that I put the labels and got the results without any difficulties and unnecessary complexity. For higher accuracy I ran the SELECT several times and chose the estimate mean of all.



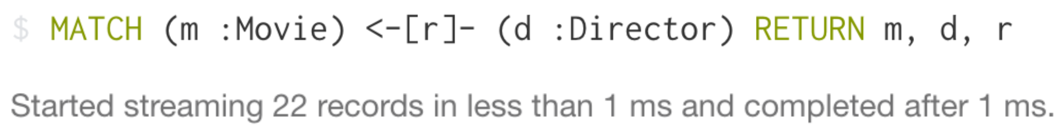
```
$ MATCH (m :Movie) RETURN m
```

Started streaming 16 records after 1 ms and completed after 1 ms.



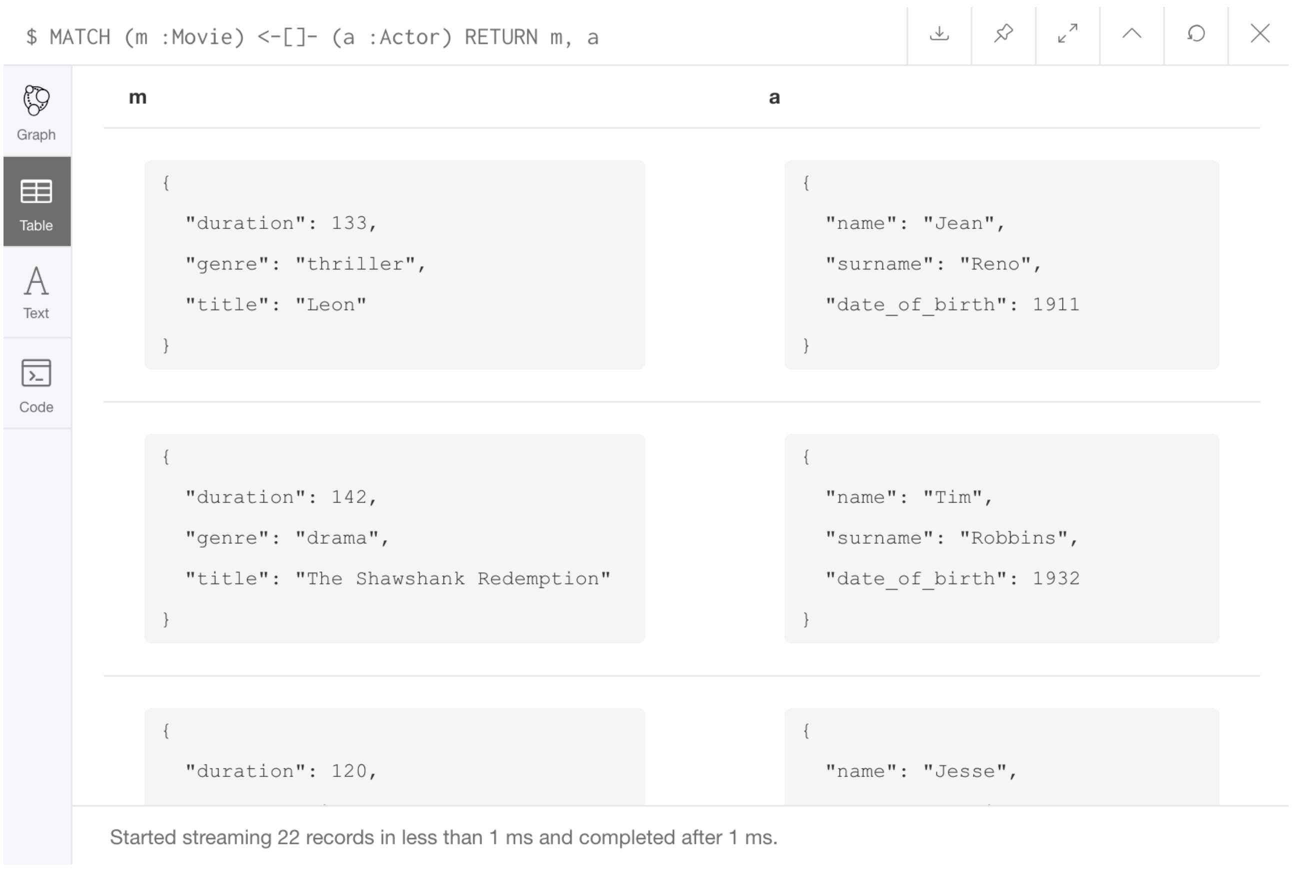
```
$ MATCH (d :Director) RETURN d
```

Started streaming 8 records in less than 1 ms and completed after 1 ms.



```
$ MATCH (m :Movie) <-[r]- (d :Director) RETURN m, d, r
```

Started streaming 22 records in less than 1 ms and completed after 1 ms.



```
$ MATCH (m :Movie) <-[]- (a :Actor) RETURN m, a
```

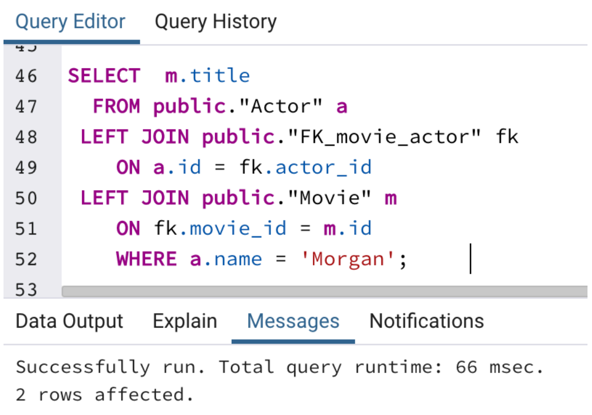
m	a
{ "duration": 133, "genre": "thriller", "title": "Leon" }	{ "name": "Jean", "surname": "Reno", "date_of_birth": 1911 }
{ "duration": 142, "genre": "drama", "title": "The Shawshank Redemption" }	{ "name": "Tim", "surname": "Robbins", "date_of_birth": 1932 }
{ "duration": 120,	{ "name": "Jesse",

Started streaming 22 records in less than 1 ms and completed after 1 ms.

3.1.4 Queries

In this chapter I would retrieve the same data to compare the runtime and the query itself.

1) First, I will select the movie title where plays an actor with the name Morgan.

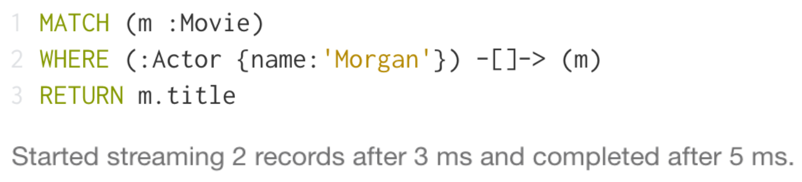

Query Editor   Query History

```sql
SELECT  m.title
  FROM public."Actor" a
 LEFT JOIN public."FK_movie_actor" fk
    ON a.id = fk.actor_id
 LEFT JOIN public."Movie" m
    ON fk.movie_id = m.id
    WHERE a.name = 'Morgan';
```

Data Output   Explain   Messages   Notifications

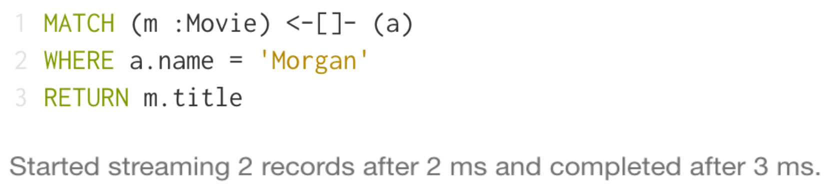
```
Successfully run. Total query runtime: 66 msec.
2 rows affected.
```

Interesting to note that Cypher does not have such strict rules for coding, there usually are a couple of choices how to express the idea.



```
MATCH (m :Movie)
WHERE (:Actor {name:'Morgan'}) -[]-> (m)
RETURN m.title
```

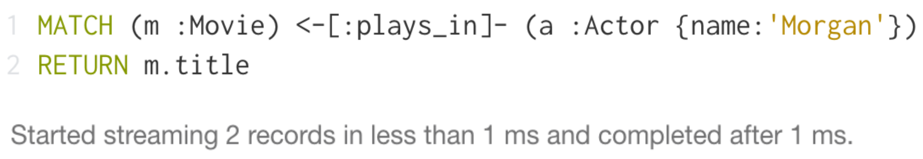
Started streaming 2 records after 3 ms and completed after 5 ms.



```
MATCH (m :Movie) <-[]- (a)
WHERE a.name = 'Morgan'
RETURN m.title
```

Started streaming 2 records after 2 ms and completed after 3 ms.

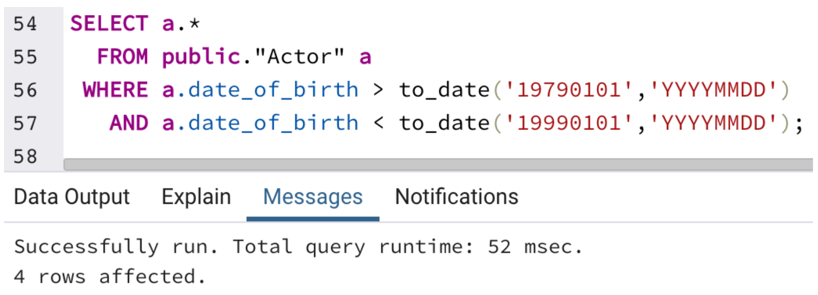
We do not necessarily need to write WHERE clause at all.



```
MATCH (m :Movie) <-[:plays_in]- (a :Actor {name:'Morgan'})
RETURN m.title
```

Started streaming 2 records in less than 1 ms and completed after 1 ms.

2) All columns of Actor table with a birth date between 1979-01-01 and 1999-01-01.

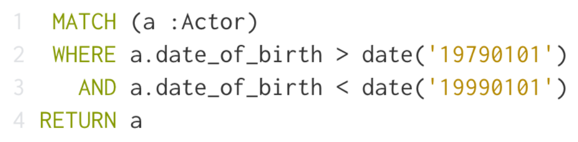


```sql
SELECT a.*
  FROM public."Actor" a
 WHERE a.date_of_birth > to_date('19790101','YYYYMMDD')
   AND a.date_of_birth < to_date('19990101','YYYYMMDD');
```

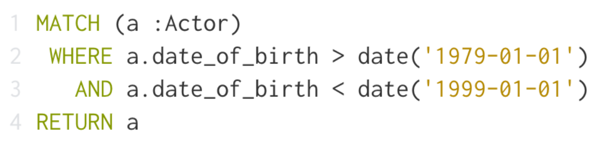
Data Output   Explain   Messages   Notifications

```
Successfully run. Total query runtime: 52 msec.
4 rows affected.
```

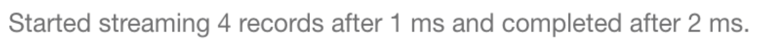
Again, Cypher proves itself to be a very flexible language.



```
MATCH (a :Actor)
WHERE a.date_of_birth > date('19790101')
  AND a.date_of_birth < date('19990101')
RETURN a
```

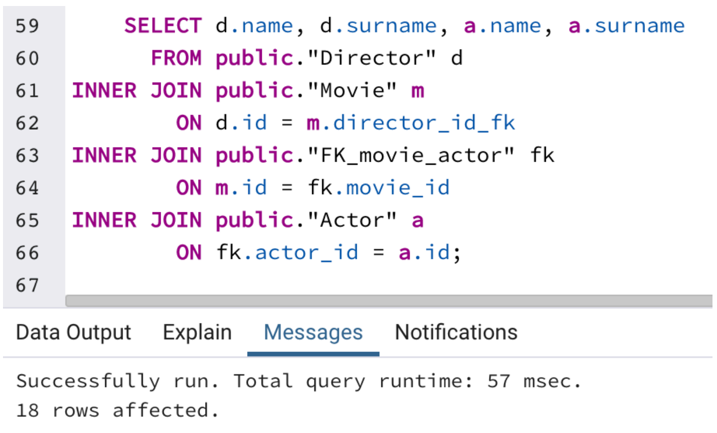


```
MATCH (a :Actor)
 WHERE a.date_of_birth > date('1979-01-01')
   AND a.date_of_birth < date('1999-01-01')
RETURN a
```



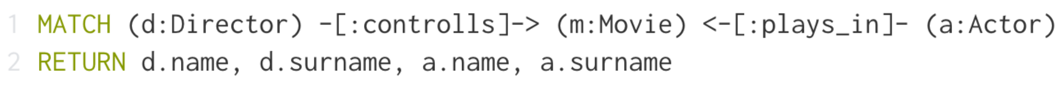
```
Started streaming 4 records after 1 ms and completed after 2 ms.
```

3) Retrieving actors and directors have relationships to the same movie.

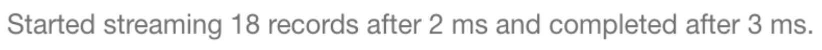


```
    SELECT d.name, d.surname, a.name, a.surname
      FROM public."Director" d
INNER JOIN public."Movie" m
        ON d.id = m.director_id_fk
INNER JOIN public."FK_movie_actor" fk
        ON m.id = fk.movie_id
INNER JOIN public."Actor" a
        ON fk.actor_id = a.id;
```

```
Successfully run. Total query runtime: 57 msec.
18 rows affected.
```

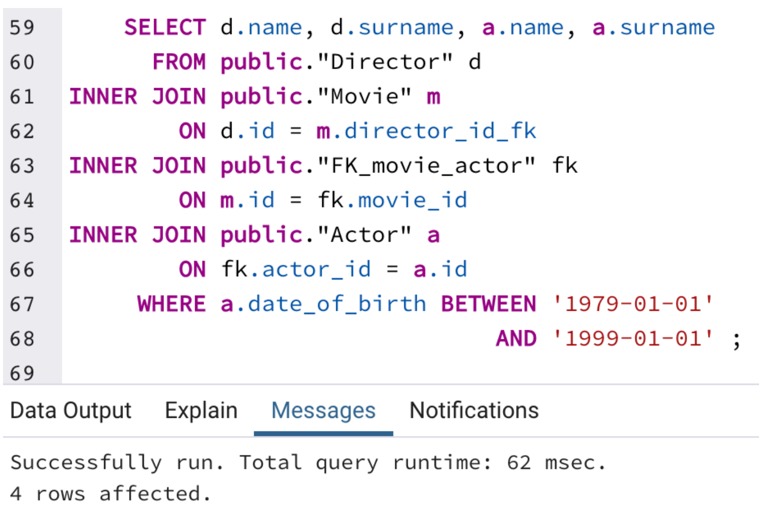


```
MATCH (d:Director) -[:controlls]-> (m:Movie) <-[:plays_in]- (a:Actor)
RETURN d.name, d.surname, a.name, a.surname
```



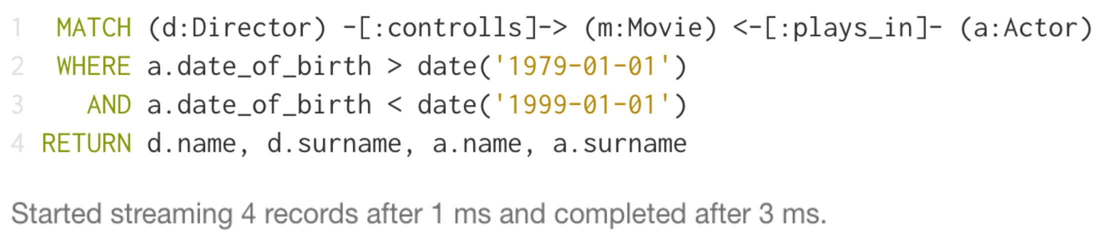
```
Started streaming 18 records after 2 ms and completed after 3 ms.
```

4) And then upgraded a bit SQL query and mix together two previous queries.



```
    SELECT d.name, d.surname, a.name, a.surname
      FROM public."Director" d
INNER JOIN public."Movie" m
        ON d.id = m.director_id_fk
INNER JOIN public."FK_movie_actor" fk
        ON m.id = fk.movie_id
INNER JOIN public."Actor" a
        ON fk.actor_id = a.id
     WHERE a.date_of_birth BETWEEN '1979-01-01'
                               AND '1999-01-01' ;
```

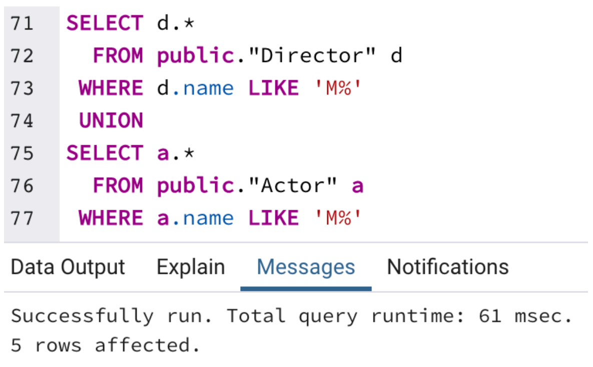
```
Successfully run. Total query runtime: 62 msec.
4 rows affected.
```



```
MATCH (d:Director) -[:controlls]-> (m:Movie) <-[:plays_in]- (a:Actor)
WHERE a.date_of_birth > date('1979-01-01')
  AND a.date_of_birth < date('1999-01-01')
RETURN d.name, d.surname, a.name, a.surname
```

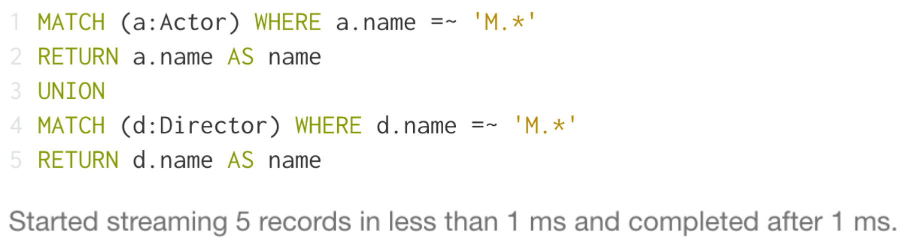
```
Started streaming 4 records after 1 ms and completed after 3 ms.
```

5) I also tried UNION command.



```
SELECT d.*
  FROM public."Director" d
 WHERE d.name LIKE 'M%'
 UNION
SELECT a.*
  FROM public."Actor" a
 WHERE a.name LIKE 'M%'
```

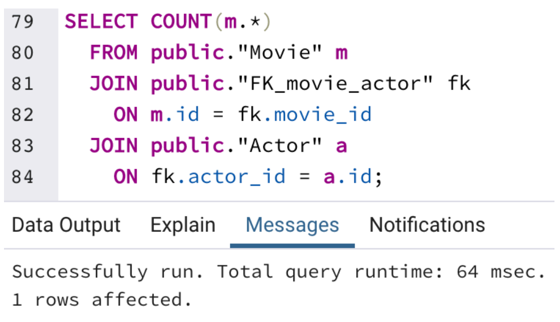
```
Successfully run. Total query runtime: 61 msec.
5 rows affected.
```



```
MATCH (a:Actor) WHERE a.name =~ 'M.*'
RETURN a.name AS name
UNION
MATCH (d:Director) WHERE d.name =~ 'M.*'
RETURN d.name AS name
```

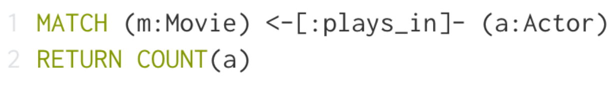
```
Started streaming 5 records in less than 1 ms and completed after 1 ms.
```

6) COUNT function



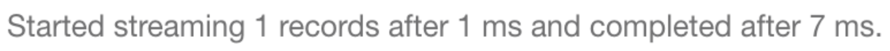
```
SELECT COUNT(m.*)
  FROM public."Movie" m
  JOIN public."FK_movie_actor" fk
    ON m.id = fk.movie_id
  JOIN public."Actor" a
    ON fk.actor_id = a.id;
```

```
Successfully run. Total query runtime: 64 msec.
1 rows affected.
```



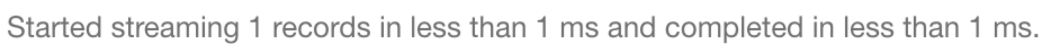
```
MATCH (m:Movie) <-[:plays_in]- (a:Actor)
RETURN COUNT(a)
```

When match for the first time:



```
Started streaming 1 records after 1 ms and completed after 7 ms.
```

When match EVERY next time:



```
Started streaming 1 records in less than 1 ms and completed in less than 1 ms.
```

3.1.5 Data replication

Streaming replication in PostgreSQL (with local storage) is the most common approach. In this approach, you can use local disks or attach persistent volumes to your instances. A primary node has the tables' data and write-ahead logs (WAL). This Postgres WAL log is then streamed to a secondary node. The new secondary node needs to replay the entire state from the primary node. The replay operation may then introduce a significant load on the primary node.

The second approach relies on disk mirroring (also called volume replication.) In this approach, changes are written to a persistent volume. This volume then is mirrored to another volume at the same time. It works for all relational databases. You can use it for MySQL, PostgreSQL, or SQL Server. However, the disk mirroring approach to replication in Postgres also requires that you replicate both table and WAL log data. Further, each write to the database now needs to synchronously go over the network. You can't miss any data because that could leave your database in a corrupt state.

The third approach turns the replication and disaster recovery process around. Writes are sent to the primary node that makes a backup every day, and incremental backups every 60 seconds. That makes up to a cloud-native design.

Neo4j's main responsibility is to safeguard data, which is done by Core Server. The Core Servers replicate data using the Raft protocol. Raft helps to confirm that the data is consistent before applying commits. This means that after the majority of Core Servers in a cluster has accepted the transaction then it is then safe to work with the commit in the end application.

Read Replicas' responsibility is to help with scaling workloads (Cypher queries, procedures, and so on). Read Replicas act like cache data that the Core Servers safeguard, but they are not simple key-value caches. In fact Read Replicas are fully-fledged Neo4j databases capable of executing arbitrary read graph queries and procedures. Read Replicas are asynchronously replicated from Core Servers using incremental log. Periodically (usually in the ms range) a Read Replica will ask a Core Server about new transactions it has processed since the last poll, and the Core Server will send these transactions to the Read Replica. Many Read Replicas can be fed data from a relatively small number of Core Servers, allowing for a large scale.

4 Results and Discussion

4.1 Getting started

To start with any relational database a person needs to have strong theoretical background. Even after that the following process of designing a DB a developer has to follow several rules such as Normal Forms so that data is consistent. DB designers have to think through the structure of the tables including all the attributes and connections. All the entities have to have the exact same created structure and if trying to change it a programmer would face certain difficulties. Despite the data are stored logically, when half of the columns is incomprehensible numbers or arrays which are foreign key's identifiers, we cannot fully say we easily understand and see the data like it is in the compared graph database.

On behalf of post-relational database, we have graph database Neo4j. The simplest model and implementation save some struggles of not only skilled developers but beginners as well. The developer can get straight to work without wasting time on designing the model and then creating the carcass for inserting data writing hundreds of lines of code. An interesting difference is that every part can be added with a property and if having the property in one place does not mean there is an obligation to include the same property to other records as well. It is a perfect elementary way to store additional or more detailed information when needed. Obviously, the nodes have relationship and they have a direction from which node to which the connection goes. In contrary to the relational DB, if there is a relationship it can work both way. It concerns M-N type specifically. In the example (:Actor) –[:plays_in] –> (:Movie) there is no necessity to store the relationship backwards like (:Actor) <– [:is_played_by] – (:Movie). The relationship work both ways not affecting to performance. When doing MATCH command, we do not have to specify the direction of relationship. However, if we put the wrong direction it will not give us any result. Note that we always have to specific relationship direction in CREATE.

These two databases have different approaches to unique identifier. PostgreSQL has a classic primary key id and there is a specific data type which is very suitable for id. Serial data type creates a sequence and sets the next value generated by the sequence as the default value for the column. In other words, it is an AUTO_INCREMENT function. The value that was set be the sequence becomes default value of the column. It also adds NOT NULL constraint to the column because a sequence always generates value. The owner of the sequence is assigned to the id column and, as a result, the sequence is deleted when the id column or table is dropped.

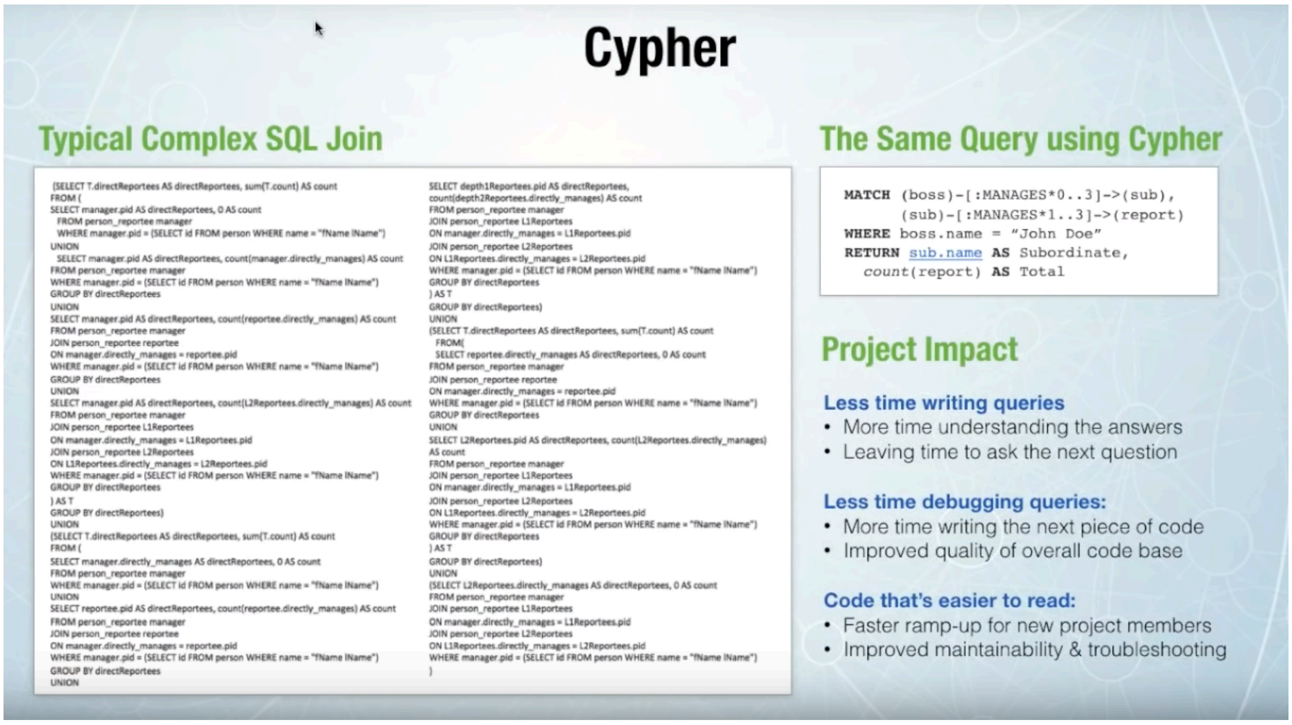
Concerning Neo4j’s id, as you might notice, I did not set any id for any node, property nor relationship. Nevertheless, each of them has its unique id. When clicking on node, at the bottom will be displayed it properties and among them there is an autogenerated self-adding id. It is usually used for deleting, because when querying we usually do not need object’s id.

4.2 Languages

It is not a secret that SQL is a special language for relational databases. After understanding the relational theory, we are able to use SQL masterfully. However, graph native Cypher proves us to be way more understandable even for non-IT specialists. The language allows the developer not to be exactly that precise in some aspects when SQL would cause a syntax error. Cypher also use something between alias and variable, if you will. It goes without saying that the size of the query and the runtime is immoderately shorter, which provides a better performance.

Speaking of Cypher specifically, I cannot but mention the impressive comparison of the real-life example of the SQL and Cypher query. We can notice similar reducing Cypher’s query size in my previous running queries.

Figure 18 Comparison SQL code to Cypher's



Cypher

Typical Complex SQL Join

```
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
FROM (
SELECT manager.pid AS directReportees, 0 AS count
  FROM person_reportee manager
  WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
UNION
  SELECT manager.pid AS directReportees, count(manager.directly_manages) AS count
FROM person_reportee manager
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT manager.pid AS directReportees, count(reportee.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT manager.pid AS directReportees, count(L2Reportees.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
FROM (
SELECT manager.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
UNION
SELECT reportee.pid AS directReportees, count(reportee.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT depth1Reportees.pid AS directReportees,
count(depth2Reportees.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
  FROM(
  SELECT reportee.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT L2Reportees.pid AS directReportees, count(L2Reportees.directly_manages)
AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT L2Reportees.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
)
```

The Same Query using Cypher

```
MATCH (boss)-[:MANAGES*0..3]->(sub),
      (sub)-[:MANAGES*1..3]->(report)
WHERE boss.name = “John Doe”
RETURN sub.name AS Subordinate,
  count(report) AS Total
```

Project Impact

Less time writing queries
- More time understanding the answers
- Leaving time to ask the next question

Less time debugging queries:
- More time writing the next piece of code
- Improved quality of overall code base

Code that’s easier to read:
- Faster ramp-up for new project members
- Improved maintainability & troubleshooting

Source: https://www.youtube.com/watch?v=lb90EBfAj0o, 2019

4.3 Retrieving data

Relational database while searching for the related data from another table goes through every row and compares its id to the content of the stored foreign key. Neo4j stores relationship directly to the nodes and in case of retrieving data does not search through the whole records but matches straight the needed node. This is the reason why the graph DB with no doubt showed much better performance. Both of databases have indexes though. Applying the same queries to both of the databases the runtime differs by not just milliseconds but dozens of them. The outcomes exceeded all the expectations in a good way.

After the query is finished and run, the results in the relational database a performed with no surprises in a form of table. However, graph Neo4j can provide requested data not only in the form of graph, but also in the text form or table of tables. I was pleasantly amazed to see in the interface of the browser application for Neo4j a button to export the matched data to CSV file directly.

5 Conclusion

In this work I made an overview of existing database types and models, explained the basis of relational database and concepts of several major types of NoSQL DBs. I made a small transparent data model of movies, directors and actors and then implemented it to the chosen presentative PostgreSQL and Neo4j following the same steps and compared the results.

NoSQL (or Not-Only SQL or Non-relational) databases appeared to overcome some limitations of databases of relational type. As it was mentioned before, RDBMS's JOINs are its best and worst feature at the same time. When the database grows in its size the number of relations is increasing. So is the number of JOINS, which at the meantime extremely increase the cost of the query and runtime.

According to my practice with Neo4j, I can say that basing on what I have experienced so far, graph DB Neo4j can be used instead PostgreSQL and give much higher performance for their clients. It is a brilliant combination of best features of two contrary databases and can be a good alternative not only for relational, but also non-relational databases. The browser application was smoother, the language more logical and simpler and the basic logic of the graph DB makes more sense. Even though the majority is used to the relational concept, the enormous results of speed speaks for itself saving plenty of time not only when retrieving data, but also when writing the code which is also very important.

To conclude I would like to add, that even after being familiar with relational DB before writing the thesis, it was easier for me to work with graph database instead of already well-known one. I hope that my research and comparison would encourage developers to pay more attention and choose carefully the proper database for their purpose and data and my work would help them with that.

6 References

Berg, Kristi et al. (2012). “History Of Databases”. International Journal of Management & Information Systems (IJMIS). 17. 29. 10.19030/ijmis.v17i1.7587.

DATE, Christopher J. “An Introduction to Database System”. 8th edition. 2004, s. 10. ISBN 0-321-18956-6.

ELMASRI, Ramez and Shamkant B NAVATHE. “Fundamentals of database systems”. 6th edition. 2011. ISBN 13: 978-0-136-08620-8.

Wladston Viana Ferreira FILHO “Computer science distilled: learn the art of solving computational problems / Wladston Viana Ferreira Filho”. — 1st ed.”. 2018. ISBN 978-0-9973160-0-1

HARRINGTON, Jan L. “Relational database design and implementation: clearly explained”. 3rd edition. 1998. ISBN 978-0-12-374730-3.

DATE, Christopher J. “SQL and Relational Theory: How to Write Accurate SQL Code”. 2nd edition. 2011. ISBN 978-1-449-31640-2.

TAYLOR, Allen G. “SQL All-in-One For Dummies”. 2nd Edition. 2011. ISBN 978-0-470-92996-4.

BEAULIEU, “Alan. Learning SQL: Master SQL Fundamentals”. 2nd Edition. 2009. ISBN 978-0-596-52083-0.

“SQL Constraints”. [online]. [cit. 2018-11-21]. Available at: https://www.w3schools.com/sql/sql_constraints.asp

Power, D. J., "Decision Support Systems Vendor List", DSSResources.COM, URL http://DSSResources.COM/vendorlist, accessed "2019-01-14", 2019.

LAI, Eric. “No to SQL? Anti-database movement gains steam”: NEWS ANALYSIS. Computerworld [online]. 2009 [cit. 2019-01-14]. Available at: http://www.computerworld.com/s/article/9135086/No_to_SQL_Anti_database_-

STRAUCH, Christof. “NoSQL Database”. 2009. Available at: http://www.christof-strauch.de/nosqldbs.pdf

HURWITZ, Judith et al. “Big Data For Dummies. A Wiley Brand”. 2013. ISBN 978-1-118-50422-2.

REDMOND, Eric and Jim WILSON. “Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement”. 2012. ISBN 9781934356920.

CHANG, Fay et al. “Bigtable: A Distributed Storage System for Structured Data” [online]. Google, 2009 [cit. 2019-03-03]. Available at: https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf

ROUSE, Margaret. “An architect's guide: How to use big data: graph database” [online]. 2016 [cit. 2019-03-03]. Available at: https://whatis.techtarget.com/definition/graph-database

Glossary [online]. Microsoft, 2010 [cit. 2019-03-11]. Available at: https://docs.microsoft.com/en-us/previous-versions//cc966484(v=technet.10)

Clustering: Neo4j [online]. [cit. 2019-03-11]. Dostupné z: https://neo4j.com/docs/operations-manual/current/clustering/introduction/

ERDOGAN, Ozgun. Three Approaches to PostgreSQL Replication and Backup [online]. [cit. 2019-03-11]. Dostupné z: https://www.citusdata.com/blog/2018/02/21/three-approaches-to-postgresql-replication/

7 Appendix

List of Supplements…