Neo4j IN ACTION

Jonas Partner
Aleksa Vukotic

MANNING

MEAP Edition
Manning Early Access Program
Neo4j in Action MEAP version 3

Copyright 2012 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=812

brief contents

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=812

1 A Case for Neo4j Database

Computer science is closely related to mathematics, with a lot of its concepts coming originally from the philosophy of mathematics. Algorithms, cryptography, computation and automation, and even basic theories of mathematical logic and Boolean algebra are all mathematical concepts that closely couple these two disciplines'. Another mathematical topic can often be found in the computer science books and articles – **graph theory**. In computer science, graphs are used to represent specific data structures, for example organization hierarchy, social network, processing flow. Typically, during software design phase, the structures, flows and algorithms are described as graph diagrams on the whiteboard. The object-oriented structure of the computer system is modeled as graph as well, with inheritance, composition and object members.

However, while the graphs are used extensively during software development process, when it comes to data persistence, we usually forget about graphs. We try to fit our data to relational tables and columns, normalize and renormalize its structure until it looks completely different to what it is trying to represent. Take one example – access control list, problem solved over and over again in many enterprise applications. We would typically have tables for users, roles and resources. Then we would have many-to-many tables to map users to roles, and roles to resources. In the end we would have at least five relational tables to represent rather simple data structure, which is actually a graph. Then we would use an Object Relational Mapping tool to map this data to our object model, which is also actually a graph.

Wouldn't it be nice if we could represent the data in its natural form, so that we can make mappings more intuitive, and skip the repeating process of "translating" our data to and from storage engine? Thanks to graph databases, we can. Graph databases use the graph model to store data as graph, with structure consisting of vertices and edges, the two entities used to model any graph.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=812

In addition, we can use all algorithms from the long history of graph theory, to solve graph problems more efficiently and in less time then using relational database queries.

Once you have read this book you will be familiar with Neo4j, one of the most prominent graph databases currently available. You will learn how the Neo4j graph database helps you model and solve graph problems in a better performing and more elegant way, even when working with large data sets.

1.1 Why Neo4j

Why would we use graph database, or more specifically Neo4j, as database of choice? To try to answer that question, let's take a look at en example of a graph domain problem and compare the Neo4j-based solution to the traditional usage of relational database as data storage technology.

The example we are going to explore is a social network – a set of users that can be friends with each other. Figure 1.1 illustrates the social network where users connected with arrows are representing friends.

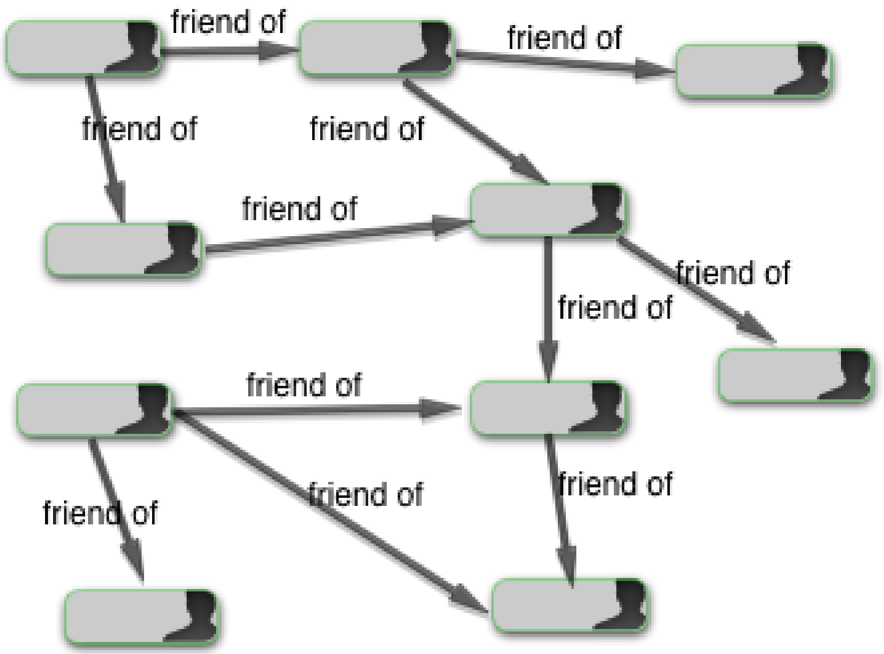


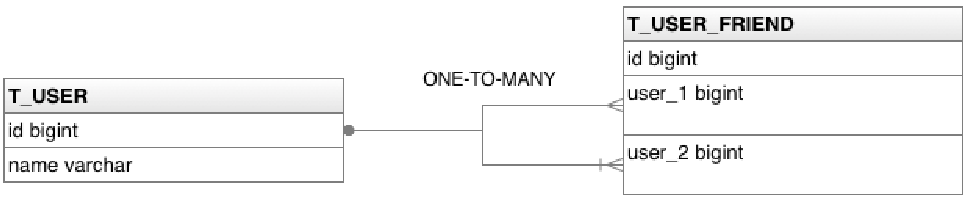
Figure 1.1 User and their friends represented as graph data structure

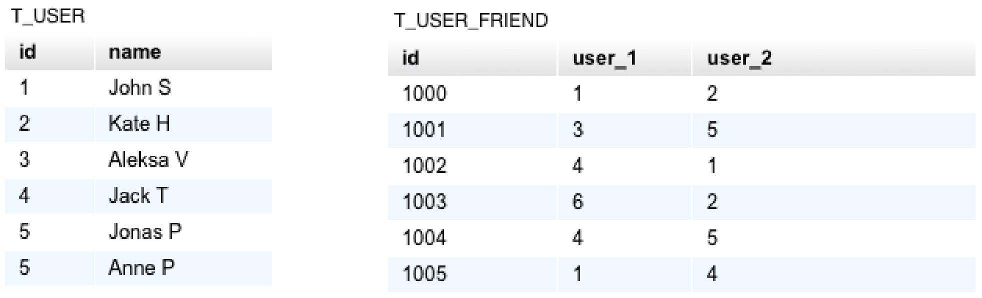
©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=812

Let’s take a look at the relational model that would use to store data about users and their friends.

1.2 Graph Data in a Relational Database

In a relational database, we would typically have two relational tables for storing social network data - one for storing user information, and another one that stores the relationships between users (see relational diagram on figure 1.2).





T_USER

id	name
1	John S
2	Kate H
3	Aleksa V
4	Jack T
5	Jonas P
5	Anne P

T_USER_FRIEND

id	user_1	user_2
1000	1	2
1001	3	5
1002	4	1
1003	6	2
1004	4	5
1005	1	4

Figure 1.1 SQL diagram of tables representing user and friends data

Listing 1.1 shows the actual SQL script for the creating tables using MySQL database.

Listing 1.1 SQL script defining tables for social network data

```
create table t_user (                                                      #A
  id bigint not null,
  name varchar(255) not null,
  primary key (id)
);
create table t_user_friend (                                               #B
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=812

```
    id bigint not null,
    user_1 bigint  not null,
    user_2 bigint  not null,
    primary key (id)
  );
  alter table t_user_friend
      add index FK416055ABC6132571 (user_1),
      add constraint FK416055ABC6132571
          foreign key (user_1) references t_user (id);                    #C
  alter table t_user_friend
      add index FK416055ABC6132572 (user_2),
      add constraint FK416055ABC6132572
          foreign key (user_2) references t_user (id);
```

#A Table definition for storing user information

#B Table definition for storing friendship relations

#C Foreign key constraints

Table t_user contains columns with user information, while table t_user_friend simply has two columns referencing table t_user using foreign key relation. The primary key and foreign key columns have indexes for quicker lookup operations, a typical strategy when modelling relational databases.

1.2.1 Querying Graph Data using SQL

Now, how would we go about querying our relational data? Getting the direct friends of a particular user is quite straightforward select query statement, for example:

```
select distinct uf.* from t_user_friend uf where uf.user_1 = ?
```

How about finding all friends of a user's friends? This time we would typically join t_user_friends table with itself before querying:

```
select distinct uf2.* from t_user_friend uf1[CA] inner joint t_user_friend
uf2 on uf1.user_1 = uf2.user_2[CA] where uf1.user_1 = ?
```

Popular social networks usually have a feature where they suggest people from your friendship network as potential friends or contacts up to the certain depth. If we wanted to do something similar to find friends of friends of friends of the user, we would need another join operation:

```
select distinct uf3.* from t_user_friend uf1[CA] inner joint t_user_friend
uf2 on uf1.user_1 = uf2.user_2[CA] inner joint t_user_friend uf3 on
uf2.user_1 = uf3.user_2[CA]  where uf1.user_1 = ?
```

Similarly, to iterate through forth level of friendship, we would need four joins. To get all connections for the famous six degrees of separation problem, six joins would be required.

While there is nothing unusual about this approach, there is one potential problem: although we are only interested in friends of friends of a single user, we have to perform a join of all data in t_user_friend table, and then discard all rows that we're not interested in. On a small data set this wouldn't be big concern, but if our social network grows large, we

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=812

could have a performance problem. As we will see, this can make a huge strain our relational database engine.

To illustrate the performance of such queries, we have run the friends of friends query on the small data set of 1000 users, where each user has on average 50 friends. This means that table t_user contains 1000 records, while table t_user_friend contains 1000 X 50 = 50,000 records.

No additional database performance tuning has been performed apart form column indices defined in the SQL script from Listing 1-1. We run the friends of friends query for depths two, three and four and five. We measured the execution time, repeating toe query ten times in order to warm up any caches available that would improve performance. Table 1.1 shows the results of our experiment.

NOTE All experiments we executed on the Intel i7 powered commodity laptop with 8GB of RAM, same one that is used to write this book.

Table 1.1 Execution times for multiple join queries using MySQL database engine on data set of 1,000 users

Depth	**Execution time (seconds) 1,000 users**	**Records returned**
2	0.028	~900
3	0.213	~999
4	10.273	~999
5	92,613.150	~999

NOTE Note that with depth 3, 4 and 5 we return 999 records. Due to the small data set, any user in our database is connected to all others, instead himself.

As we can see, MySQL handles queries with depths 2 and 3 quite well. That's not unexpected – join operations are common in the relational world, so most databases engines are designed and tuned with this in mind. In addition, we used databases indexes on relevant columns to increase performance of join queries.

With depths 4 and 5, however, we see significant degradation of performance: query involving four joins takes over 10 seconds to executes, while depth 5 execution takes way too long – over minute and a half, although the number of rows returned does not change. This just illustrates the limitation of MySQL when modelling graph data – deep graphs require multiple joins, which relational databases typically don't handle too well.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=812

Inefficiency of SQL Joins

In order to find all user's friends on depth 5, relational database engine needs to generate Cartesian product of t_user_friend table five times, With 50,000 record in the table, the resulting set will have 50,000^5 rows (102,4 X 10^{21}), which takes quite a lot of time and computing power to calculate. Then, we discard more the 99% of that, to return just under 1,000 records we're interested in!

So, let's take a look how does Neo4j perform with the same data set. Instead of tables, columns and foreign keys, we are going to model users as nodes, and friendships as relationships between nodes.

1.3 Graph Data in Neo4j

Neo4j stores data as vertices and edges, or, in Neo4j terminology, nodes and relationships. Users will be represented as nodes, and friendships will be represented as relationships between user nodes. If you take another look at our social network on Figure 1-1, you will see that it represents nothing more then a graph, with users as nodes and friendship arrows as relationships.

There is one key difference between relational database and Neo4j, which we will come across straight away – data querying. We don't have table and columns in Neo4j, no SQL with select and join commands. How do we query graph database? And no, the answer is not 'write a distributed map-reduce function'. Neo4j, like all graph databases, takes a powerful mathematical concept from graph theory and uses it as powerful and efficient engine for querying data. This concept is **graph traversal**, and it's one of the main tools that make Neo4j so powerful it dealing with large-scale graph data.

1.3.1 Traversing the Graph

The traversal is the operation of visiting a set of nodes in the graph by moving between nodes connected with relationships. It's a fundamental operation for data retrieval in a graph, and as such it is unique to the graph model. The key concept of traversals is that they are localized – querying the data using traversal only takes into account the data which is required, without the need to perform expensive grouping operations on the entire data set, like we did with join operations on relational data.

Neo4j provides rich traversal API, which you can employ to navigate through the graph. In addition, you can use REST API, or Neo4j query languages to traverse your data. We will dedicate much of this book to teach you the principles and best practices when traversing data with Neo4j.

In order to get all friends of user's friends we will run the code in Listing 3.2

Listing 3-2 Neo4j Traversal API code for finding all friends on depth two.

```
TraversalDescription traversalDescription =
[CA]Traversal.description()[CA].relationships("IS_FRIEND_OF",
Direction.OUTGOING)  [CA].evaluator(Evaluators.atDepth(2))
[CA].uniqueness(Uniqueness.NODE_GLOBAL);
Iterable<Node> nodes = traversalDescription.traverse(nodeById).nodes();
```

Don't worry if you don't understand the syntax of the above code snippet – everything will be explained slowly and thoroughly in the next few chapters. Figure 1.3 illustrates the traversal of our social network graph, based on the traversal description above.

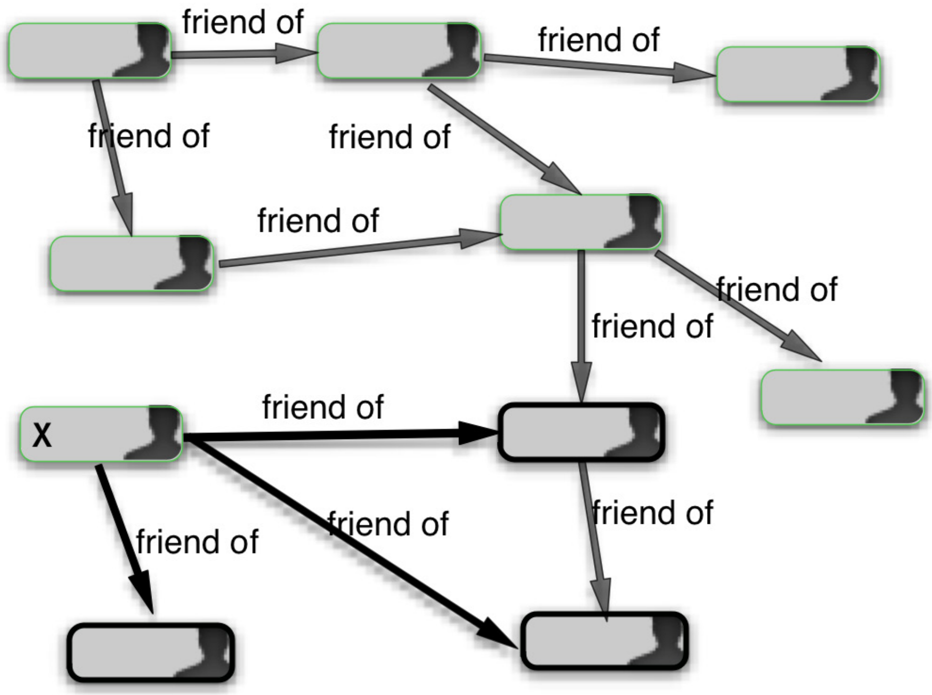


Figure 1.3 Traversing the social network graph data

Before the traversal starts, we select the starting node from which the traversal will start (Node X of Figure 1.3). Then we follow all friendship relationships (arrows) and collect the visited nodes as result. While traversal are satisfied, the traversal continues it's journey from one node to another, via relationships that connect them. When the rules stop applying, the traversal stops. For example, the rule can be visit only nodes that are on depth one from the starting node, in which case once all nodes on depth one are visited, the traversal stops.

Traversal can also be configured to stop once it visits all nodes or all relationships in a graph. Our friends traversal example will finish when all nodes in the graph have been visited at least once.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=812

Table 1.2 shows the performance metrics for functionally same queries we executed on MySQL database, with data set of 1,000 users and 50 friends per user.

Table 1.2 Execution times for graph traversal using Neo4j on data set of 1,000 users

Depth	Execution time (seconds) 1,000 users	Records returned
2	0.04	~900
3	0.06	~999
4	0.07	~999
5	0.07	~999

NOTE Similarly to MySQL setup, no additional performance tuning has been performed on Neo4j instance. Neo4j was running in embedded mode, with default configuration.

First thing to notice is that Neo4j performance is significantly better for all queries, except the simplest one. Only when looking for friends of friends the MySQL performance is comparable to the performance of Neo4j traversals. Traversal of friends on depth 3 is four times faster then the SQL counterpart. When performing traversal on depth 4, the results are 5 orders of magnitude times better! Finally, the depth 5 results are ten million times faster with Neo4j traversal compared to SQL queries!

Another conclusion that can be made from the results from table 1.2 is that performance of the query does degrade with the depth of the traversal when the number of nodes returned remains the same. MySQL query performance degrades with the depth of the query because of the Cartesian product operations that are executed before most of the results are discarded. Neo4j, on the other hand, keeps track of nodes visited, so it can skip the nodes already visited, and therefore significantly improve performance.

To find all friends on depth 5, MySQL will perform Cartesian product on t_user_friend table 5 times, resulting in $50,000^5$ records, out of which all but 1,000 are discarded. Neo4j, on the other hand, will simply visit nodes in the database, and when there is no more nodes to visit, it will stop the traversal. That is why Neo4j can keep constant performance as long as the number of nodes returned remains the same, opposed to significant degradation in performance when using MySQL queries.

NOTE Graph traversals perform significantly better then the equivalent SQL queries (thousands times better with traversal depth of 4 and 5!). At the same time, the performance does not decrease dramatically with the depth – the traversal on depth 5 is

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=812

only 0.03 seconds slower then the traversal on depth 2. The perfoemance of the most complex SQL queries is more then 10,000 times slower comparing to the simple ones.

But how does this approach scale? To get the answer to that question, let's repeat the experiment with the data set of one million users.

1.4 SQL Joins vs Graph Traversal on Large Scale

We are going to use exactly the same data structures as before, the only difference will be the amount of data.

In MySQL we will have 1,000,000 records in t_user table, and approximately 1,000,000 X 50 = 50,000,000 records in t_user_friend table. We have run the same four queries against this data set (friends on depth 2,3,4 and 5). Table 1.3 shows the collected results for the performance of SQL queries in this case.

Table 1.3 Execution times for multiple join queries using MySQL database engine on data set of 1,000,000 users

Depth	**Execution time (seconds) 1,000 users**	**Records returned**
2	0.016	~2,500
3	30.267	~125,000
4	1,543.505	~600,000
5	NOT FINISHED	-

Comparing to the SQL results for a 1,000 users data set, we can see that the performance of depth-two query has stayed the same, which can be explained by the design of the MySQL engine to handle table joins efficiently using indexes. Queries with depth 3 and 4 (which use 3 and 4 multiple join operations. respectively) demonstrate much worse results, by at least two orders of magnitude. SQL query for all friends on depth 5 did not finish in one hour we have been running the script!

These results clearly show that the MySQL relational database is optimized for a single join queries, even on the large data sets. However, the performance of multiple join queries on large data sets degrades significantly, to the point that some queries are not even executable (for example friends on depth 5 on one million users data set)

Why are are relational database queries so slow?

The results in the Table 3.3 are somewhat expected given the way joins work. As we discussed earlier, each join creates a Cartesian product of all potential combinations of rows, and then filters out those that are not matching the where clause. With one million users, the Cartesian product of 5 joins (equivalent of query on depth 5) contains huge

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=812

number of rows – billion billion billions – way to many zeros to be readable. Filtering out all the records that don't match the query is too expensive – such that the SQL query on depth 5 never finishes in a reasonable time.

Let's now repeat the same experiment with Neo4j traversals. We will have one million nodes representing users and approximately 50 million relationships stored in Neo4j. We run exactly the same four traversals as in the previous example, and collected the performance results in Table 1.4.

Table 1-4 Execution times for graph traversal using Neo4j on data set of 1,000,000 users

Depth	Execution time (seconds) 1,000 users	Records returned
2	0.010	~2,500
3	0.168	~110,000
4	1.359	~600,000
5	2.132	~800,000

As you can see from results in table 1.4, the increase in data by thousand times did not affect Neo4j performance significantly. The traversal get slower as we increase the depth, however the main reason for that is the increased number of results that are returned. The performance is slowing linearly with the increase of the result size, and is predictable even with larger data set and depth level. In addition, this is at least hundred times better then the corresponding MySQL performance. Main reason for this predictability of Neo4j is the localized nature of the graph traversal – irrespective how many nodes and relationships there are in the graph, traversal will only visit ones that are connected to the starting node, according to the traversal rules. Remember, the relational join operations will compute Cartesian product before discarding irrelevant results, affecting performance exponentially with the growth of the data set. Neo4j, however, visits only nodes that are relevant to the traversal, so it can keep predictable performance irrespective of the total data set. The more nodes traversal has to visit, the slower the traversal, as we have seen while increasing the traversal depth. However, this increase is linear and still doesn't depend on the total graph size.

What is the secret of Neo4j's speed?

No, Neo4j developers haven't invented super-fast algorithm for the military. Nor is the Neo4j speed product of fantastic speed of the technologies it relies on – it's implemented in Java after all!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=812

The secret is in the data structure – the localized nature of graphs makes it very fast for this type of traversals. Imagine yourself cheering your team on the a football stadium. If someone asks you how many people is sitting five meters around you, you will get up and count them. If the stadium is half empty, you will count people around you as fast as you can count. If the stadium is packed, you will still do it in a similar time! Yes, it may be slightly slower, but only because you have to count more people because of the higher density. We can say that, irrespective of how many people are on the stadium, you will be able to count the people around you at predictable speed – as you're only interested in people near you, you won't be worried about packed seats on the other end of the stadium for example.

This is exactly how Neo4j engine works in our example – it counts nodes connected to the starting node, at the predictable speed. Even when the number of nodes in the whole graph increases (given similar node density), the performance can remain predictably fast.

If you apply same football analogy to the relational database queries, we would count all people in the stadium and then remove those that not around us – not the most efficient strategy given the interconnectivity of the data.

These experiments demonstrated that Neo4j graph database is significantly faster in querying graph data then relational database. In addition, a single Neo4j instance can handle three orders of magnitude larger data sets without performance penalties. Independence of the traversal performance on the graph size is one of the key aspects that make Neo4j ideal candidate for solving graph problems, even when data sets are very large.

In the next section we'll try to answer the question what graph data actually is, and how can Neo4j help us model our data models as natural graph structure.

1.5 Graphs Around Us

Graphs are considered the most ubiquitous natural structures. The first scientific paper on graph theory is considered to be a solution of the historical Seven Bridges of Königsberg problem, written by Swiss mathematician Leohnard Euler in 1774. The seven bridges problem started what is now known as graph theory, with its model and algorithms applied across wide scientific and engineering disciplines.

NOTE You can read more about Seven Bridges problem and history of graph theory in Appendix A.

There are a lot of examples of graphs usage in modern science. For example, graphs are used to describe the relative positions of the atoms and molecules in chemical compounds. The laws of atomic connections in physics are also described using graphs. In biology, the evolution tree is actually a special form of graph. In linguistics, graphs are used to model

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=812

semantic rules and syntax trees. Network analysis is based on graph theory as well, with applications in traffic modeling, telecommunications and topology.

In computer science a lot of problems are modeled as graphs: social networks, access control lists, network topologies, taxonomy hierarchies, recommendations engines, etc. Traditionally, all of these problems would use graph object models (implemented by the development team), and store the data in the normalized form in a set of tables with foreign key associations.

In the previous section we have seen how social network is stored in the relational model using two tables and a foreign key constraint. Using Neo4j graph database, we have preserved the natural graph structure of the data, without compromising performance or storage space – improving the query performance significantly.

When using Neo4j graph database to solve graph problems the amount of work for the developer is reduced, as the graph structures are already available via the database APIs. The data is stored in its natural format – as graph nodes and relationships, so the effects of object-relational mismatch are minimized. And, the graph query operations are executed in an efficient and performant way by using Neo4j's traversal APIs.

Throughout this book we will demonstrate the power of Neo4j by solving problems such as access control lists and social networks with recommendation engines.

If you are not familiar with graph theory, don't worry; we will introduce some of the graph theory concepts and algorithms, as well as their applicability to graph storage engines such as Neo4j when it becomes relevant in the later chapters.

In the next section we are going to the discuss the reasons why have graph databases (and other NoSQL technologies) started to gain popularity on the computer world. We will also discuss different categories of NoSQL systems with the focus on their differences and applicability.

1.6 Neo4j in NoSQL space

Since the beginning of computer software, the data that the applications had to deal with has grown in complexity enormously. Complexity of the data included not only its size, but also the inter-connectedness of the data, it's ever-changing structure and concurrent access to the data.

With all these aspects of changing data, it has been recognized that relational databases, which have for a long time been de-facto standard for data storage, are not the best fit for all problems that increasingly complex data required. From that recognition, a number of new storage technologies have been created, with one common goal to solve the problems relational databases were not good at. All these new storage technologies are known under umbrella term NoSQL.

NOTE While the NoSQL name stuck, it doesn't reflect accurately the nature of the movement, giving the (wrong) impression that it's against SQL as a concept. The better name would probably be non-relational databases, as the relational/non-relational paradigm was the subject of discussion, where SQL is just a language used with relational technologies.

The NoSQL movement was born as the acknowledgment that new technologies were required to cope with the data changes. Neo4j, and graph databases in general, are part of the NoSQL movement, together with a lot of other more or less related storage technologies.

With rapid developments in NoSQL space, its growing popularity and a lot of different solutions and technologies to choose from, anyone new coming into the NoSQL world faces a lot of confusion when selecting the right technology. That's why in this section we will try to clarify categorization of NoSQL technologies, with the special view on the applicability of each category. In addition, we will make sure to explain the place of graph databases and Neo4j within NoSQL at the same time.

KEY-VALUE STORES

Key-value stores represent the simplest, yet very powerful category, designed from the need of handling high volume concurrent access to the data. Caching is a typical use case key-value based technology. Key-value stores allow storing data in very simple structures, often in-memory, for very fast access even in highly concurrent environment.

The data is stored in a huge hash table and is accessible by its key. The data structure is in the form of key/value pairs, and the operations are mostly limited to simple put (write) and get (read) operations. The values support only simple data structures like text or binary content, although some more recent key-value stores support a limited set of complex data structures (Redis for example supports lists and maps as values).

Key-value stores are the simplest NoSQL technologies. All other NoSQL categories build on the simplicity, performance and scalability of key-value stores to fit some specific use cases better.

COLUMN-FAMILY STORES

The need for some sort of data structure, while keeping the distributed key-value model, which scales very well, brought the column-family store category to the NoSQL scene. The idea was to group similar values (or columns) together, by keeping them in the same column-family (for example, user data, or books information). The column families are stored in a single file enabling better reads and writes performance on related data. The main goal of this approach was high performance and high availability when working with big data – therefore it's no surprise that the leading technologies in this space are Google's BigTable and Cassandra, originally developed by Facebook.

DOCUMENT-ORIENTED DATABASES

A lot of the real problems required the data structure that looks like a document (content management systems, user registration data, CRM data…), and document-oriented databases used that structure to describe it in a simple, yet efficient schema-less manner.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=812

The document model adds additional complexity to the data structure, by adding the self-contained documents and associative relationships to the document data. This makes it easier to model the data for the common software problems, however it comes at the expense of slightly lower performance and scalability compared to key-value and column-family stores. But the convenience of the object model built into the storage system is usually a good trade-off for all but massively concurrent use cases (most of us are not trying to build another Google or Facebook after all).

GRAPH DATABASES

The graph databases were designed with the view that a lot of the time, we build graph-like structures in our applications, but still store the data in the un-natural way – either in tables and columns of relational databases, or even in other NoSQL storage systems. As we mentioned before, problems like ACL lists, social networks, or indeed any kind of networks are natural graph problems. The idea of the graph data model is at the core of graph databases – we are finally able to store our object model which represents graph data – as a persisted graph!

The data model that can naturally represent a lot of complex software requirements, and the efficiency and performance of the graph traversal querying are main strength of graph databases.

Table 1.4 illustrates the use cases and representative technologies for each of the NoSQL categories.

Table 1.4 Overview of NoSQL categories

NoSQL Category	Typical Use Cases	Best-Known Technologies
Key-Value Stores	- Caches - Simple domain with fast read access - Massively concurrent systems	- Redis - Memcached - Tokyo Cabinet
Column-Family Stores	- Write on a big scale - Co-located data access (for reading and writing)	- Cassandra - Google BigTable - Apache HBase
Document-Oriented Databases	- When domain model is document by nature - To simplify development using natural document data structures	- MongoDb - CounchDb

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=812

	- Highly scalable systems (although on the lower level then the key-value and column-family stores)	
Graph Databases	- With inteconnected data - Domain can be represented with nodes and relationships naturally - Social Networks - Recommendation engines - Access Control Lists	- Neo4j - AllegroGraph - OrientDB

NOTE You can read about NoSQL movement, and Neo4j's place in it, in more detail in Appendix A

So far in this chapter we've seen some example of efficient usage of Neo4j to solve graph-related problems, illustrated how common real world problems can be naturally modeled as graphs, and learned where graph databases and Neo4j in particular sit within the wider NoSQL space.

There is one key aspect of Neo4j that none of the other NoSQL stores have, and which is very important when it comes in adoption of new storage technologies in the enterprise world – its transactional behavior.

1.7 Neo4j – the ACID-compliant database

Transaction management has been the talking point of NoSQL technologies since they started to gain popularity. Trade-off of transactional attributes for performance and scalability has been the common theme in non-relational technologies that targeted big data. For example, some (BigTable, Cassandra, CouchDB) opted to trade off consistency, allowing the clients to read stale data in some cases in distributed system (eventual consistency). Or, in key-value stores that concentrated on read performance, durability of the data wasn't of too much interest (Memcached). Or, atomicity on a single operation level, without possibility to wrap multiple database operations within single transaction, typical for document oriented databases.

While each of approaches mentioned are valid in specific use cases (for example caching, high data read volumes, high load and concurrency), they are usually the first hurdles when it comes to introducing non-relational databases to any enterprise or corporate environment.

Although devised a long time ago for the relational databases, transaction attributes are still important in the most practical use cases. Neo4j has taken a different approach here. Neo4j's goal is to be a graph database, with the emphasis on database. This means that you will get full ACID support from the Neo4j database:

- atomicity (A), where you can wrap multiple database operations within the single transaction, and make sure they are all executed atomically – if one of the operations fails, the entire transaction will be rolled back
- consistency (C), when you write data to the Neo4j, you can be sure that every client accessing the databases afterwards will read the latest updated data
- Isolation (I), where you can be sure that operations within the single transaction will be isolated one from another, so that writes in one transaction, won't affect reads in another transaction
- durability (D), so you are certain that the data you write to Neo4j will be written to disk and available after database restart or server crash

The ACID transactional support provides seamless transition to Neo4j for anyone used to relational databases, and offers safety and convenience in working with graph data. Transactional support is one of the strong points of Neo4j, which differentiates it from the majority of NoSQL solutions, and makes it a good option not only for NoSQL enthusiasts, but in enterprise environments as well.

1.8 Summary

In this chapter we learned how much more efficient and performant can Neo4j be compared to relational databases to solve specific problems, like social network in our example. We illustrated performance and scalability benefits of graph databases represented by Neo4j, showing clear advantage of Neo4j to relational database when dealing with graph related problems.

We also learned about some of the key characteristics of Neo4j, as well as discussed its place within the larger NoSQL movement of related data storage technologies.

In the next chapter we are going to move straight to the hands-on, practical usage of Neo4j, and start modeling a specific graph problem with Neo4j Java API.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=812