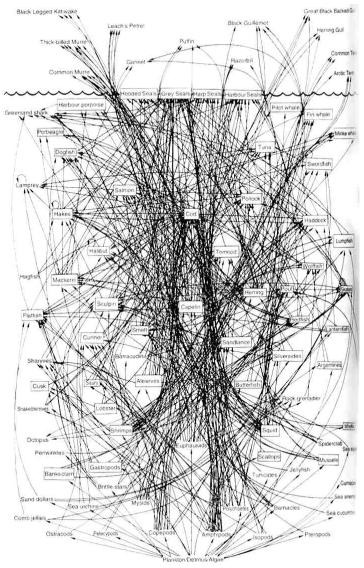
The Neo Database

This document is an introduction to the Neo database, a next-generation database which we believe addresses some of the shortcomings of the data management systems on the market today. The target audience is a technologically inclined person with a background in software development. This document gives an overview of what Neo does and the benefits and drawbacks of using Neo in your application development. It will also try to give a glimpse of how Neo works and how developing with Neo is different from developing with a relational database.

In a nut-shell, Neo is a *robust*, *scalable*, *high-performance* and *embedded* persistence engine for efficient management of *semi-structured* and *network-oriented* data. That's a lot of buzzwords! Let's have a closer look:

- Neo is a database designed for **network-oriented data**. This is data that is ordered in complex networks or deep trees. Where the relational model is based on tables, columns and rows, Neo's primitives are nodes, relationships and properties. Together, these form a large network of information that we call a *node space*.
- Additionally, Neo shines at handling **semi-structured data**. Semi-structured data is a research term that is quickly gaining ground outside of academia. Simply put, semi-structured data typically has few mandatory but many optional attributes. As a consequence, it usually has a very dynamic structure, sometimes to the point where it varies even between every single element. Data with that degree of variance is difficult to fit in a relational database schema but can be easily represented in the Neo model.
- Neo is an **embedded** persistence engine, which means that it's a small, light-weight and non-intrusive Java library that's easy to include in your development environment. Because it's embedded there's no overhead from serializing persistence requests to declarative queries.
- Neo has been written from scratch with **performance** and **scalability** in mind. It has been proven to handle large networks of data (100+ millions of nodes, relationships and properties).
- Finally, Neo is **robust**. It has full support for JTA and JTS, 2PC distributed ACID transactions, configurable isolation levels and battle-tested transaction recovery. These aren't just words: Neo has been in production for more than three years in a highly demanding 24/7 environment. It is mature, robust and ready to be deployed.



A network representing the food chain in the Atlantic Ocean (from imma.org)

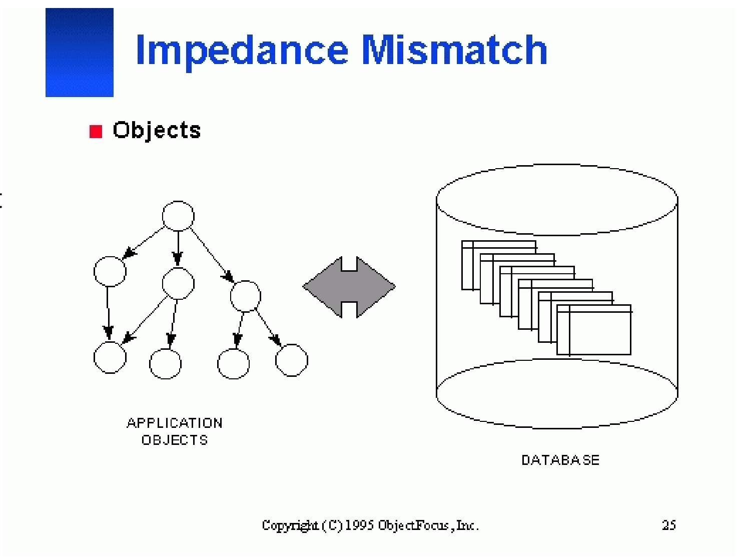
Before we elaborate more on the details of the Neo database and dig into an illuminating example, let's provide some background to help explain why we feel that there are some cases where current-generation databases fall short. This might take some time, but since it's important to know where Neo is coming from, please bear with us!

Background

The relational database represents one of the most important developments in the history of computer science. Upon its arrival some 30 years ago, it revolutionized the way the industry views data management and today it is practically ubiquitous.

However, in some cases the relational database technology shows its age. One issue that has been widely debated is the so-called *object-relational impedance mismatch*, which recognizes the fact that the relational data model is conceptually and practically incompatible with the object-oriented model that is so dominant in software engineering today. The popularization of object-oriented languages in the 90s forced this issue into the open.

By the turn of the century, the industry had responded to the impedance mismatch in two ways: a) with different flavors of O/R-mapping tools – software layers that allowed developers to work in the flexible object-oriented domain and let the tools map their OO designs to the relational world's tables and schemas. And b) with component frameworks like Java EE and .NET that promised that they would deal with persistence and allow the developer to “not care” about the relational database and focus on the business domain. Both of these solutions manage to shield the developer from the relational world, albeit with design time constraints, runtime constraints or both. But the core problem remains: the relational model is inherently inconsistent with the current software development paradigm.

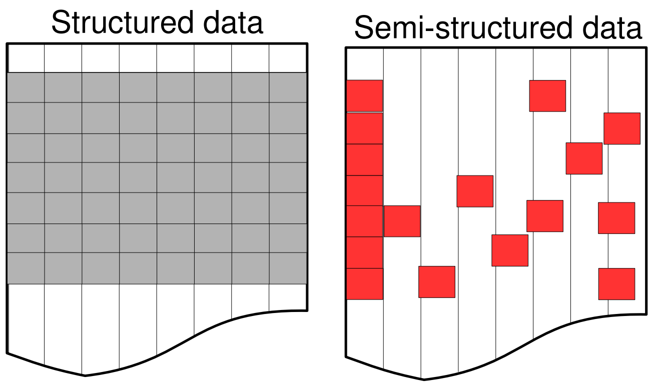


A 1995 slide describing the impedance mismatch

An issue that is similar to the impedance mismatch problem but in many ways harder to address is that of *schema evolution*. Many teams have produced a well-designed schema from a concept model or object-oriented design, only to realize as they put it into production how difficult it is to evolve that schema. One of the relational database's most powerful characteristics – that of strict enforcement of data conformity and referential integrity – is now working against the need to adapt to changing business requirements. Different teams have different ways to deal with schema evolution but most depend on ad-hoc scripting and manual intervention and none on good tool support. Just as the rise of object-oriented languages brought up the object-relational impedance mismatch in the mid 90s, the popularity of *agile methodologies* and their emphasis on evolutionary software design will put the problems of schema evolution at the top of the agenda in this decade.

Both the impedance mismatch and the schema evolution problem stem from the relational database's origin in the early 70s, at which time there were no object-oriented programming languages and the dominant methodology was waterfall. Both issues place constraints on the modern software development stack, at design-time and at development-time. But the relational database as it stands today also suffers from runtime problems in some important – and increasingly common – scenarios.

Firstly, a relational database shows very poor performance characteristics for *sparse tables*. Sparse tables are tables where many rows have no values for many columns. Sparse tables are a result of trying to squeeze in *semi-structured data* in a relational schema. Recall that *semi-structured data* is data that has few mandatory but many (maybe indefinitely many) optional attributes. The relational model is built to handle *structured* data – data with a defined and complete schema – and it does that very well. The problem is that, as Stefano Mazzocci of Apache and MIT research fame puts it, *“the great majority of the data out there is not structured and [there's] no way in the world you can force people to structure it.”*[1] Even the major database vendors acknowledge that the relational technology is insufficient when it comes to handling most of the data in the world.[2]

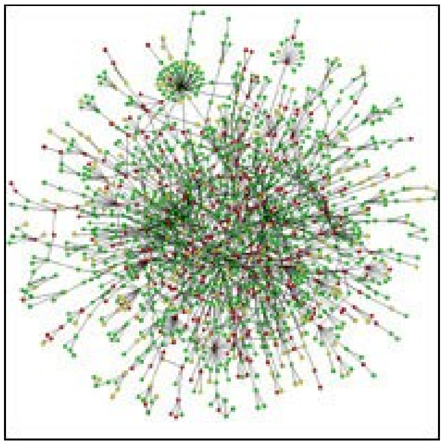


How semi-structured data is distributed in a relational database

1 “Stefano's Linotype”, http://www.betaversion.org/~stefano/linotype/news/46/, Feb 11, 2004

2 See Daniela Florescu's comments referenced in the conclusion on p.8

Secondly, there are many data sets that are naturally ordered in networks. The semantic web movement generates one such (gigantic) data set, and you can find many other examples from domains such as content management, bioinformatics, artificial intelligence, social networks, business intelligence and many more. The problem with such data sets is that processing them is by necessity translated to joins in the relational model – and a join is a very expensive operation. Every "jump" along an edge between two nodes in a network means one join. Anecdotal evidence suggests that most applications stop looking after three jumps because the exponential processing cost is too expensive. Obviously that is a problem if one wants to get any data out of the database in a timely and orderly fashion.


A visualization of a yeast protein (from jeffkennedyassociates.com)

So we have a situation where the relational database – while great at a number of tasks – has four major drawbacks:

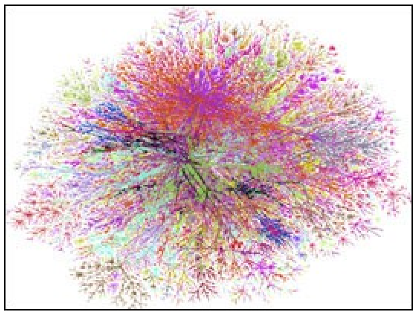
1. The **object-relational impedance mismatch** makes it unnecessarily difficult and time-consuming to squeeze an object-oriented "round object' into a relational "square table."

2. The static, rigid and inflexible nature of the relational model makes it difficult to **evolve schemas** in order to meet changing business requirements. For the same reasons, the database often holds a team back when they try to apply agile software development methodologies by rapidly evolving the object-oriented layer.

3. The relational model is exceptionally poor at capturing **semi-structured data**, a type of information that industry analysts and researchers alike agree will be the next "big thing" in information management.

4. A network is a very efficient data storage structure. It's not a coincidence that the human brain is one huge network or that the world wide web is structured as an ad-hoc network. The relational model can capture **network-oriented data**, but it is very weak when it comes to traversing that network in order to extract information.

Suffering from all these problems with its relational database backend, a crew from Windh Technologies – a software house specializing in development of high-end Enterprise Content Management services – set out in 2000 to create a transactional persistence engine with high performance, scalability and robustness but without the inherent disadvantages of the relational model. We dubbed the project and its resulting software Neo. And now, after describing the reasons that led to its conception, let's take a look at how it works!

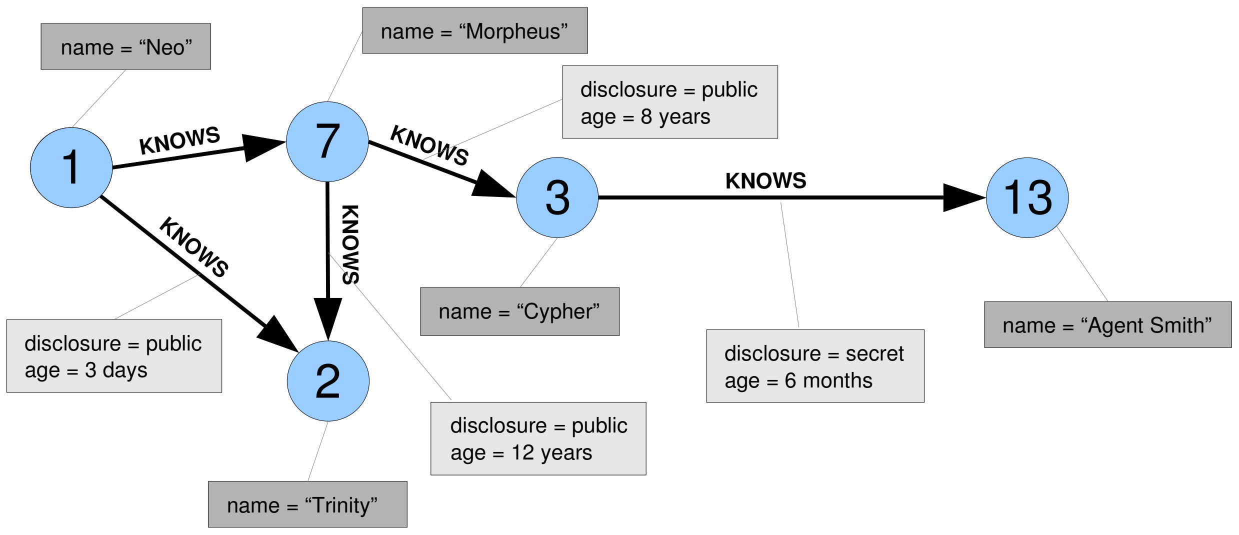

The internet, cirka 1998 (from Wired)

The Neo Model

Representation

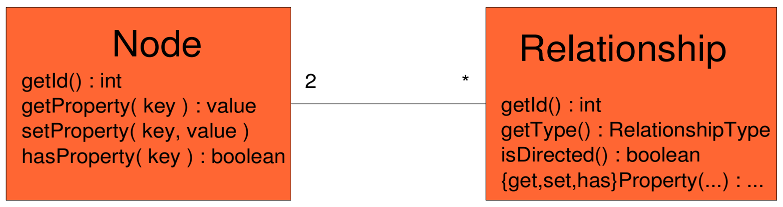
In the Neo model, everything is represented by nodes, relationships and properties. A relationship connects two nodes, has a well-defined, mandatory type and can be optionally directed. Properties are key-value pairs that are attached to both nodes and relationships. When you combine nodes, relationships between them and properties on both nodes and relationships they form a *node space* – a coherent network representing your business domain data.

Here's how a simple social network might be modelled:



Note how all nodes have ids and how all the relationships have a type (in this case, the only type is **KNOWS**, a directed relationship representing that one person knows of another). In this example, all nodes have a name property (seen in the dark gray boxes) and the relationships have properties describing how long the people have known each other and whether the acquaintance is secret (seen in the light gray boxes).

Working with nodes and relationships is easy. The basic operations are as follows:


Node
getId() : int
getProperty(key) : value
setProperty(key, value)
hasProperty(key) : boolean
2
*
Relationship
getId() : int
getType() : RelationshipType
isDirected() : boolean
{get,set,has}Property(...) : ...

This is an intuitive representation of a network and reasonably similar to the APIs of object-oriented math libraries that deal with discrete graphs, similar to W3C's DOM APIs and probably to many other implementations that want to represent a network of data in an object-oriented language. So far, nothing new under the sun. (Except, of course, that Neo handles at least an order of magnitude more nodes and relationships than any component on the market that we have found.)

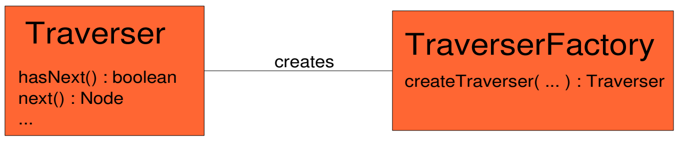
It's worth noting, however, that relationships in this model are full-blown objects and not just implicit associations between nodes. If you have another look at the social network example, you'll see that there's more information in the relationships between nodes than in the nodes themselves. The value of a network is in the connections between the nodes and Neo is one of the few models to capture that.

The fact that Neo is good at modeling network-oriented data structures should be clear by now. In fact, Neo's data model *is* a network. But how about semi-structured data? Continuing with our example, let's say that we wanted to feed our social network data from publicly available FOAF[1] files. Because of the nature of RDF this would yield a semi-structured data set. We know from the background section that modeling semi-structured data has traditionally been very difficult. In Neo, however, it's as easy as adding the properties to the relevant node: there's no need to worry about whether other nodes have the same property set and there's no requirement for conformance of data. This means that after parsing a few FOAF files, Morpheus from the example above might have a completely different set of properties than, say, Trinity. And that's fully supported and efficiently represented in the Neo model.

Traversal

Now that we can represent data in a flexible and intuitive manner, how do we get information out of it? Unlike a relational database, Neo does not support declarative queries. In database theory, Neo would be categorized as a *navigational database* – which means that you navigate from a (given or arbitrary) start node via relationships to the nodes that match your criterias. In principle, you can do this via the API outlined above (“`node.getRelationship( OWNER ).getEndNode()`”, etc) but that is laborous and requires a lot of boiler-plate code. Neo provides a powerful traversal framework that makes writing the most complex “queries” a trivial task.

This traverser framework is the key to effective development with Neo. A traverser is a tool that encapsulates a “query” such as *“give me all Morpheus' friends and his friends' friends”* or *“does Trinity know someone who is acquainted with an agent?”*. A traverser is actually a Java Iterator so working with traversers is trivial for anyone with a passing knowledge of Java.



The power of the traverser framework is how easy it is to express complex traversals. There are four basic components of a traversal: the starting node, the relationships that we wish to traverse, a stop criteria to know when we should stop traversing and a selection criteria to know which nodes to return as a result of the traversal. For the query about Morpheus' friends (*“give me all Morpheus' friends and his friends' friends”*) on the social network example above, a pseudo code traversal initialization would look like this:

1 FOAF is “Friend Of A Friend”, a standard for expressing social relations on the web. Since FOAF is an RDF flavor, a FOAF file can optionally include arbitrary statements from a schema completely out of the control of the FOAF authors. This freedom and decentralization leads to a semi-structured data set.

```
Traverser morpheusfriends = factory.createTraverser(
   Traverser.BREADTH_FIRST, // first return friends, then friends' friends
   morpheus,                // start with Morpheus
   KNOWS,                   // relationship types we wish to traverse
   new DepthLimitStopEvaluator( 2 ), // don't traverse deeper than 2 levels
   ReturnableEvaluator.RETURN_ALL_BUT_START_NODE ); // return all nodes visited in
                                                    // traversal except the first one
for ( Node friend : morpheusFriends )
...
```

This traverser will start with Morpheus and return all his friends and friends' friends. In the example above, that would be Trinity, Cypher and Agent Smith (probably not what he expected!). What's interesting in this example is the two lines in bold: the evaluators. It turns out that the core of the traversals – and a lot of the data-driven logic of a Neo application – is encapsulated in those two classes.

The example code above used two “pre-configured” evaluators. Neo ships with a set of evaluators that cover many cases such as “traverse to the end of the network”, “traverse to a certain depth”, “return all nodes”, and so on. But for more complex (case-specific) queries, the developer passes along a “closures-like” evaluator statement, usually in the form of an anonymous inner class. For example, say that we only wanted Morpheus' old-time buddies (friends for 10 or more years). Then creating the traverser would be identical to above, except that instead of the

```
ReturnableEvaluator.RETURN_ALL_BUT_START_NODE
```

we would pass along a:

```
new ReturnableEvaluator()
{
   public isReturnableNode( TraversalPosition position )
   {
      return position.getLastRelationshipTraversed().getProperty( age ) >= 10;
   }
}
```

Setting aside the Java cruft, we basically pass in a statement that checks the age property on the relationship that we traversed to get to the current node. If this statement is evaluated to “true” then the current node will be included in the set of nodes that that are returned from the traverser. The TraversalPosition instance that is passed to the evaluators encapsulates the current traversal state and is used to access information such as the current node, the current depth, total amount of nodes traversed, etc.

Writing evaluators is very easy. Even the most advanced applications we have developed with Neo, applications that traverse extremely large and complex networks are based on evaluators that are rarely more than a few lines of code.

Benefits

Using the Neo model rather than the relational model offers several benefits to an application developer. If the application domain contains data that is naturally ordered in networks then Neo's model is natively compatible with your data. You will find Neo's representation natural and intuitive and will be surprised by how easy it is to write complex, high-performance traversals over your domain data.

If the application domain contains semi-structured data, Neo captures that easily as well. Decentralization and peer empowerment leads to data sets where individual records have individual structure. The web is the prime force behind this trend, but semi-structured data exists already in numerous other domains and it's increasing by the day. All this semi-structured data, almost impossible to model in a relational database, can be easily and efficiently stored in Neo.

Once you have put your application in production, Neo's flexible and forgiving attitude towards properties and typing will allow you to evolve your schemas as quickly in your persistence layer as in your business layer. During development, this will encourage agile methodologies with rapid refactorings and short iterations. Combined, these advantages result in shorter development times, lower maintenance costs and higher performance.

This concludes our discussion of Neo's data and programming model. Hopefully, you have seen how easily it captures both semi-structured and network-oriented data sets while still allowing room for structured and tabular information.

Performance and scalability

No discussion about a new database technology is complete without talking about performance. It is difficult to make general statements about the performance and scalability of an arbitrary Neo application, since there isn't (as of now!) enough empirical evidence to draw intellectually satisfying conclusions from. Furthermore, as all benchmarking and testing institutes point out, every environment is different and no data sets are created equal.

Suffice to say that we have had a relentless commitment to performance and scalability during the design and implementation of the Neo kernel. We will later publish a set of reproducible benchmarks, comparing Neo with popular relational and non-relational persistence solutions.

As for anecdotal evidence, remember how every “hop” in a traversal is a join and that relational databases crawls to a halt on more than a handful (say, 3 or 4) joins with a non-trivial data set? We routinely perform traversals across several *hundreds of thousands* of relationships (yes, that's depth > 100 000) in a matter of seconds, on fairly modest hardware.[1] This is possible in part due to aggressive optimization and smart caching, but mostly because we have a model that is designed for large networks of data.

Drawbacks

Neo is not a silver bullet. It has been created to solve a certain type of problems and there are scenarios where other tools are a better choice. In particular:

- Neo may have a learning curve compared to a relational tool. SQL, the relational model and popular O/R mapping frameworks are widespread today. To our knowledge, there isn't any tools that does what Neo does out there. However, it's our experience that most developers find Neo's model intuitive.

- Consequently, while we are pushing to standardize Neo's model, the sheer lack of alternative implementations creates a potential lock-in situation.

- While we have developed some rudimentary tools for browsing and interacting with a node space, the tool support for relational databases are naturally much better.

- Relational databases are better at arbitrary queries on structured data. Neo can handle structured data as well as semi-structured, but since it uses a navigational model it is harder to execute arbitrary queries on that data. A relational database is very good at for example answering the question *“how many of my customers over age 25 and a last name that starts with an F have purchased items the last two months?”*. That query doesn't go along the relationships in the node space and is thus difficult to traverse. In order to alleviate this, Neo has integrated a search engine for arbitrary property lookups.

1 For example, one Neo application in production right now with a 5+GB data set runs on a Sun v20z with 2*2.4Ghz 64-bit CPUs and 8GB RAM of which 2GB is assigned to Neo's JVM. It commonly traverses across more than 100 000 relationships in less than 10 seconds. That same data set also loads and works well on a laptop with a 64 MB JVM.

Conclusion

The relational data model is now more than 30 years old. It's good for a number of scenarios and can handle certain types of data very well. But it isn't perfect. For data that is semi-structured and/or network-oriented, the relational database offers poor runtime characteristics. Furthermore, it forces a static development cycle and is of little help to those who have to live with a domain model that is constantly changing, even after deployment. This translates to wasted development time and money.

Simultaneously, we have a rise of data that is semi-structured and data that is naturally ordered in networks. Many organisations already have that kind of data, but the dominant role of the relational database may have led them to believe that the only way to model it is by forcing it into a tabular structure and then try to work around it in upper layers. Other organisations have simply not even tried to capture that data. As Oracle puts it: *"A large volume of information is still unavailable and unexploited because the existing data modeling and management tools are not adapted to the reality of such information."*[1]

Neo is not only adapted to that reality, it is designed and built from scratch for that environment. If your organisation has data that is naturally ordered in a network or data that is semi-structured, the Neo database offers an elegant and flexible alternative that is robust, fast and scalable.

1 Daniela Florescu, Oracle, in the "ACM Queue", Vol. 3, No.8, 2005