
Graph Database Internals

In this chapter, we peek under the hood and discuss the implementation of graph databases, showing how they differ from other means of storing and querying complex, variably-structured, densely connected data. Although it is true that no single universal architecture pattern exists, even among graph databases, this chapter describes the most common architecture patterns and components you can expect to find inside a graph database.

We illustrate the discussion in this chapter using the Neo4j graph database, for several reasons. Neo4j is a graph database with native processing capabilities as well as native graph storage (see [Chapter 1](#) for a discussion of native graph processing and storage). In addition to being the most common graph database in use at the time of writing, it has the transparency advantage of being open source, making it easy for the adventuresome reader to go a level deeper and inspect the code. Finally, it is a database the authors know well.

Native Graph Processing

We've discussed the property graph model several times throughout this book. By now you should be familiar with its notion of nodes connected by way of named and directed relationships, with both the nodes and relationships serving as containers for properties. Although the model itself is reasonably consistent across graph database implementations, there are numerous ways to encode and represent the graph in the database engine's main memory. Of the many different engine architectures, we say that a graph database has native processing capabilities if it exhibits a property called *index-free adjacency*.

A database engine that utilizes index-free adjacency is one in which each node maintains direct references to its adjacent nodes. Each node, therefore, acts as a micro-

index of other nearby nodes, which is much cheaper than using global indexes. It means that query times are independent of the total size of the graph, and are instead simply proportional to the amount of the graph searched.

A nonnative graph database engine, in contrast, uses (global) indexes to link nodes together, as shown in **Figure 6-1**. These indexes add a layer of indirection to each traversal, thereby incurring greater computational cost. Proponents for native graph processing argue that index-free adjacency is crucial for fast, efficient graph traversals.



To understand why native graph processing is so much more efficient than graphs based on heavy indexing, consider the following. Depending on the implementation, index lookups could be $O(\log n)$ in algorithmic complexity versus $O(1)$ for looking up immediate relationships. To traverse a network of m steps, the cost of the indexed approach, at $O(m \log n)$, dwarfs the cost of $O(m)$ for an implementation that uses index-free adjacency.

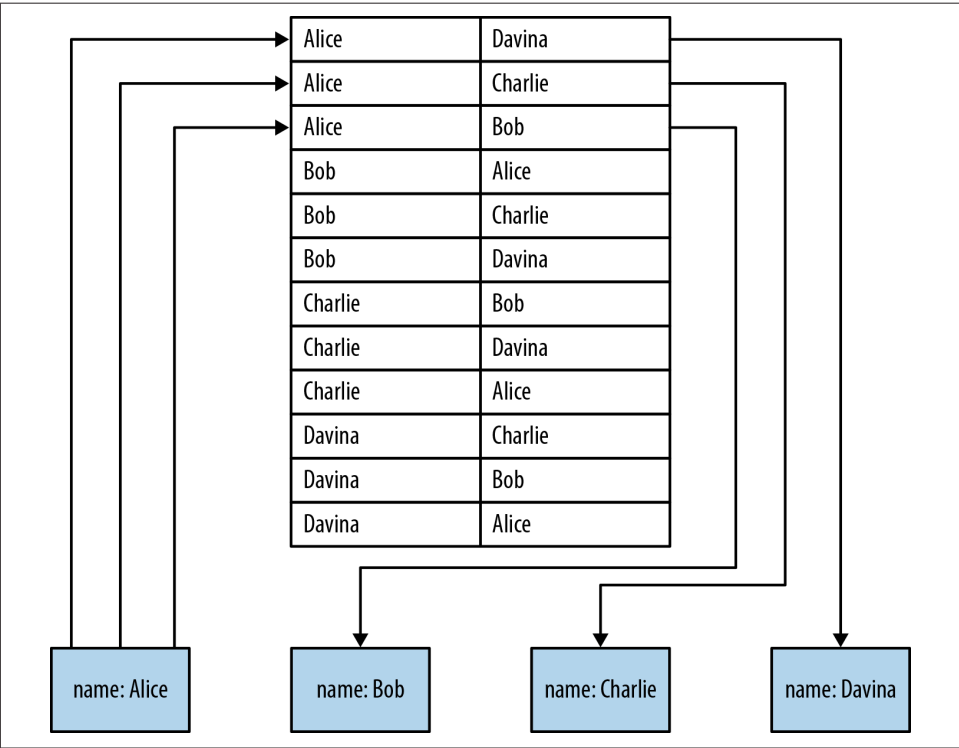


Figure 6-1. Nonnative graph processing engines use indexing to traverse between nodes

Figure 6-1 shows how a nonnative approach to graph processing works. To find Alice's friends we have first to perform an index lookup, at cost $O(\log n)$. This may be acceptable for occasional or shallow lookups, but it quickly becomes expensive when we reverse the direction of the traversal. If, instead of finding Alice's friends, we wanted to find out who is friends with Alice, we would have to perform multiple index lookups, one for each node that is potentially friends with Alice. This makes the cost far more onerous. Whereas it's $O(\log n)$ cost to find out who are Alice's friends, it's $O(m \log n)$ to find out who is friends with Alice.

Index-Free Adjacency Leads to Low-Cost “Joins”

With index-free adjacency, bidirectional joins are effectively precomputed and stored in the database as relationships. In contrast, when using indexes to simulate connections between records, there is no *actual* relationship stored in the database. From this, two problems arise:

Firstly, using a global index lookup is typically far more expensive algorithmically than traversing a physical relationship. Indexes typically cost $O(\log(n))$ in time, whereas—at least in Neo4j—traversing a relationship is $O(1)$ in time. In theory, for even modest values of n , the logarithmic costs can be many times more expensive than constant time. In practice, the performance can be even worse, as a result of the graph and its global indexes contending for resources like caches and I/O (e.g., when page contention occurs between index and graph data).

Secondly, using indexes to simulate connections becomes problematic when we try to traverse in the “opposite” direction from the one for which the index was constructed. Now we are faced with the choice of creating a reverse-lookup index for each traversal scenario, or we have to perform a brute-force search through the original index, which is an $O(n)$ operation. Given the poor algorithmic performance in this situation, joins like this are simply too costly to be of any practical use for online systems.

Index lookups can work for small networks, such as the one in Figure 6-1, but are far too costly for queries over larger graphs. Instead of using index lookups to fulfill the role of relationships at query time, graph databases with native graph processing capabilities use index-free adjacency to ensure high-performance traversals. Figure 6-2 shows how relationships eliminate the need for index lookups.

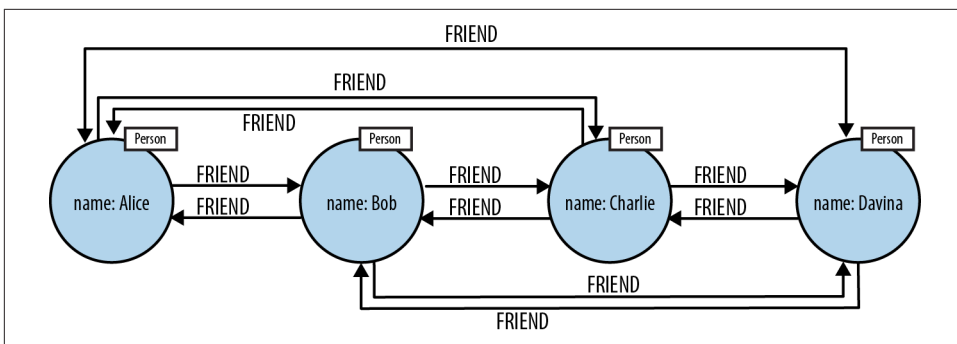


Figure 6-2. Neo4j uses relationships, not indexes, for fast traversals

Recall that in a general-purpose graph database, relationships can be traversed in either direction (tail to head, or head to tail) extremely cheaply. As we see in [Figure 6-2](#), to find Alice’s friends using a graph, we simply follow her outgoing FRIEND relationships, at $O(1)$ cost each. To find who is friends with Alice, we simply follow all of Alice’s incoming FRIEND relationships to their source, again at $O(1)$ cost each.

Given these costs, it’s clear that, in theory at least, graph traversals can be very efficient. But such high-performance traversals only become reality when they are supported by an architecture designed for that purpose.

Native Graph Storage

If index-free adjacency is the key to high-performance traversals, queries, and writes, then one key aspect of the design of a graph database is the way in which graphs are stored. An efficient, native graph storage format supports extremely rapid traversals for arbitrary graph algorithms—an important reason for using graphs. For illustrative purposes we’ll use the Neo4j database as an example of how a graph database is architected.

First, let’s contextualize our discussion by looking at Neo4j’s high-level architecture, presented in [Figure 6-3](#). In what follows we’ll work bottom-up, from the files on disk, through the programmatic APIs, and up to the Cypher query language. Along the way we’ll discuss the performance and dependability characteristics of Neo4j, and the design decisions that make Neo4j a performant, reliable graph database.

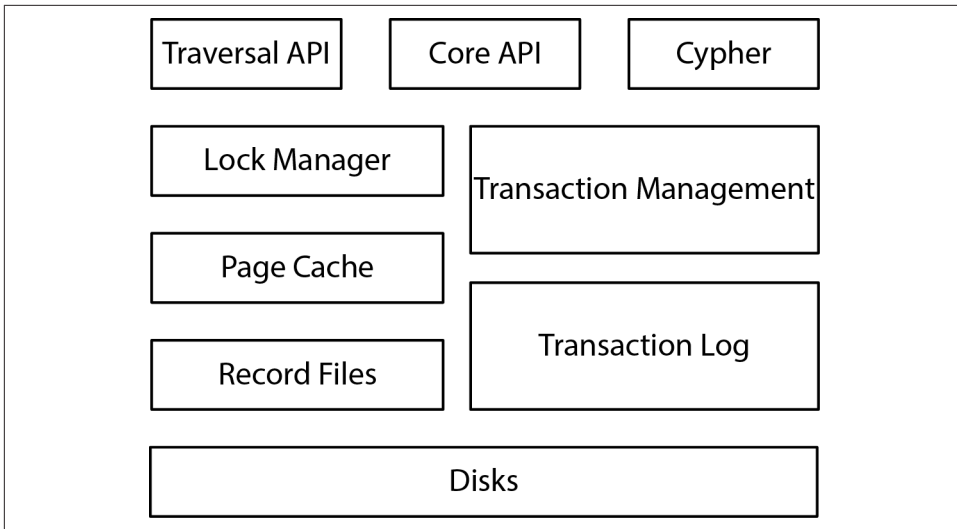


Figure 6-3. Neo4j architecture

Neo4j stores graph data in a number of different *store files*. Each store file contains the data for a specific part of the graph (e.g., there are separate stores for nodes, relationships, labels, and properties). The division of storage responsibilities—particularly the separation of graph structure from property data—facilitates performant graph traversals, even though it means the user’s view of their graph and the actual records on disk are structurally dissimilar. Let’s start our exploration of physical storage by looking at the structure of nodes and relationships on disk as shown in [Figure 6-4](#).⁸

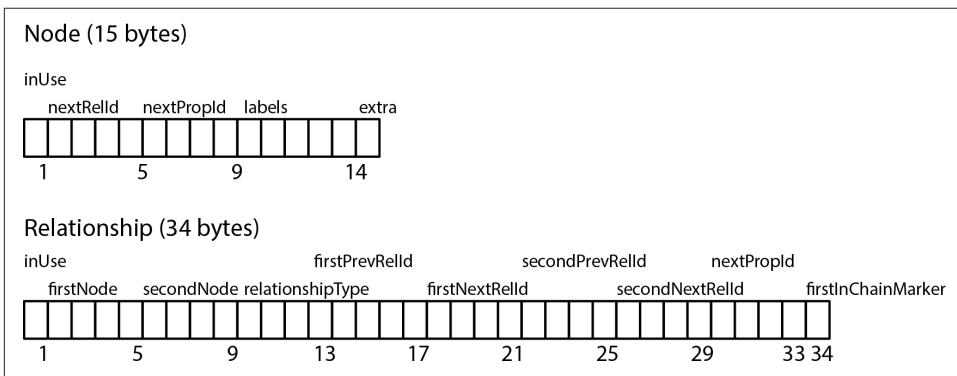


Figure 6-4. Neo4j node and relationship store file record structure

⁸ Record layout from Neo4j 2.2; other versions may have different sizes.

The node store file stores node records. Every node created in the user-level graph ends up in the node store, the physical file for which is `neostore.nodestore.db`. Like most of the Neo4j store files, the node store is a fixed-size record store, where each record is nine bytes in length. Fixed-size records enable fast lookups for nodes in the store file. If we have a node with id 100, then we know its record begins 900 bytes into the file. Based on this format, the database can directly compute a record's location, at cost $O(1)$, rather than performing a search, which would be cost $O(\log n)$.

The first byte of a node record is the in-use flag. This tells the database whether the record is currently being used to store a node, or whether it can be reclaimed on behalf of a new node (Neo4j's `.id` files keep track of unused records). The next four bytes represent the ID of the first relationship connected to the node, and the following four bytes represent the ID of the first property for the node. The five bytes for labels point to the label store for this node (labels can be inlined where there are relatively few of them). The final byte extra is reserved for flags. One such flag is used to identify densely connected nodes, and the rest of the space is reserved for future use. The node record is quite lightweight: it's really just a handful of pointers to lists of relationships, labels, and properties.

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



The node and relationship stores are concerned only with the structure of the graph, not its property data. Both stores use fixed-sized records so that any individual record's location within a store file can be rapidly computed given its ID. These are critical design decisions that underline Neo4j's commitment to high-performance traversals.

In [Figure 6-5](#), we see how the various store files interact on disk. Each of the two node records contains a pointer to that node's first property and first relationship in a relationship chain. To read a node's properties, we follow the singly linked list structure beginning with the pointer to the first property. To find a relationship for a node, we follow that node's relationship pointer to its first relationship (the `LIKES` relationship in this example). From here, we then follow the doubly linked list of relationships for that particular node (that is, either the start node doubly linked list, or the end node doubly linked list) until we find the relationship we're interested in. Having found the record for the relationship we want, we can read that relationship's proper-

ties (if there are any) using the same singly linked list structure as is used for node properties, or we can examine the node records for the two nodes the relationship connects using its start node and end node IDs. These IDs, multiplied by the node record size, give the immediate offset of each node in the node store file.

Doubly Linked Lists in the Relationship Store

Don't worry if the relationship store structure seems a little complex at first. It's not as simple as the node store or property store.

It's helpful to think of a relationship record as “belonging” to two nodes—the start node and the end node of the relationship. Clearly, we don't want to store two relationship records, because that would be wasteful. And yet it's equally clear that the relationship record should somehow belong to both the start node and the end node.

That's why there are pointers (aka record IDs) for two doubly linked lists. One is the list of relationships visible from the start node. The other is the list of relationships visible from the end node. That each list is doubly linked simply enables us to rapidly iterate through that list in either direction, and insert and delete relationships efficiently.

Choosing to follow a different relationship involves iterating through a linked list of relationships until we find a good candidate (e.g., matching the correct type, or having some matching property value). Once we have a suitable relationship we're back in business, multiplying ID by record size, and thereafter chasing pointers.

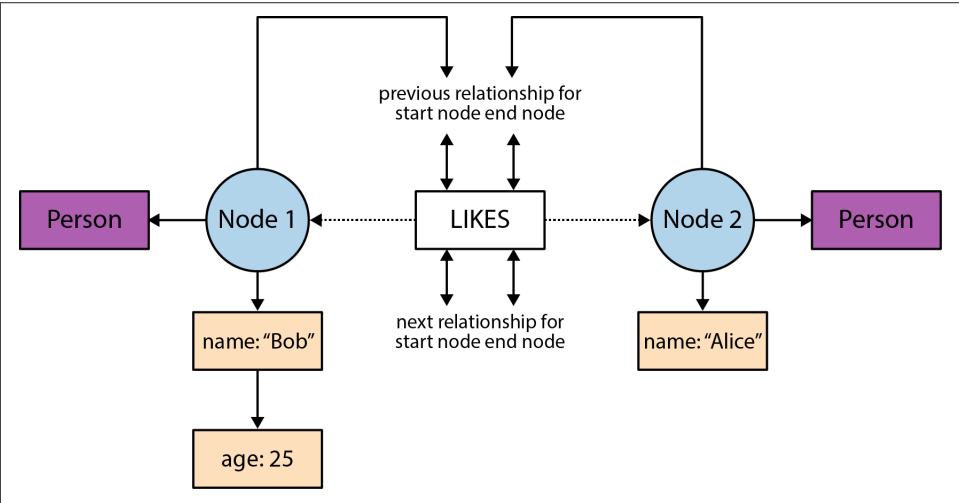


Figure 6-5. How a graph is physically stored in Neo4j

With fixed-sized records and pointer-like record IDs, traversals are implemented simply by chasing pointers around a data structure, which can be performed at very high speed. To traverse a particular relationship from one node to another, the database performs several cheap ID computations (these computations are much cheaper than searching global indexes, as we'd have to do if faking a graph in a nongraph native database):

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

Should we wish to constrain the traversal to relationships with particular types, we'd add a lookup in the relationship type store. Again, this is a simple multiplication of ID by record size to find the offset for the appropriate relationship type record in the relationship store. Similarly if we choose to constrain by label, we reference the label store.

In addition to the node and relationship stores, which contain the graph structure, we have property store files, which persist the user's data in key-value pairs. Recall that Neo4j, being a property graph database, allows properties—name-value pairs—to be attached to both nodes and relationships. The property stores, therefore, are referenced from both node and relationship records.

Records in the property store are physically stored in the `neostore.propertystore.db` file. As with the node and relationship stores, property records are of a fixed size. Each property record consists of four property blocks and the ID of the next property in the property chain (remember, properties are held as a singly linked list on disk as compared to the doubly linked list used in relationship chains). Each property occupies between one and four property blocks—a property record can, therefore, hold a maximum of four properties. A property record holds the property type (Neo4j allows any primitive JVM type, plus strings, plus arrays of the JVM primitive types), and a pointer to the property index file (`neostore.propertystore.db.index`), which is where the property name is stored. For each property's value, the record contains either a pointer into a dynamic store record or an inlined value. The dynamic stores allow for storing large property values. There are two dynamic stores: a dynamic string store (`neostore.propertystore.db.strings`) and a dynamic array store (`neostore.propertystore.db.arrays`). Dynamic records comprise linked lists

of fixed-sized records; a very large string, or large array, may, therefore, occupy more than one dynamic record.

Inlining and Optimizing Property Store Utilization

Neo4j supports store optimizations, whereby it inlines some properties into the property store file directly (`neostore.propertystore.db`). This happens when property data can be encoded to fit in one or more of a record's four property blocks. In practice this means that data like phone numbers and zip codes can be inlined in the property store file directly, rather than being pushed out to the dynamic stores. This results in reduced I/O operations and improved throughput, because only a single file access is required.

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

Having an efficient storage layout is only half the picture. Despite the store files having been optimized for rapid traversals, hardware considerations can still have a significant impact on performance. Memory capacity has increased significantly in recent years; nonetheless, very large graphs will still exceed our ability to hold them entirely in main memory. Spinning disks have millisecond seek times in the order of single digits, which, though fast by human standards, are ponderously slow in computing terms. Solid state disks (SSDs) are far better (because there's no significant seek penalty waiting for platters to rotate), but the path between CPU and disk is still more latent than the path to L2 cache or main memory, which is where ideally we'd like to operate on our graph.

To mitigate the performance characteristics of mechanical/electronic mass storage devices, many graph databases use in-memory caching to provide probabilistic low-latency access to the graph. From Neo4j 2.2, an off-heap cache is used to deliver this performance boost.

As of Neo4j 2.2, Neo4j uses an LRU-K *page cache*. The page cache is an LRU-K page-affined cache, meaning the cache divides each store into discrete regions, and then holds a fixed number of regions per store file. Pages are evicted from the cache based on a least frequently used (LFU) cache policy, nuanced by page popularity. That is, unpopular pages will be evicted from the cache in preference to popular pages, even if

the latter haven't been touched recently. This policy ensures a statistically optimal use of caching resources.

Programmatic APIs

Although the filesystem and caching infrastructures are fascinating in themselves, developers rarely interact with them directly. Instead, developers manipulate a graph database through a query language, which can be either imperative or declarative. The examples in this book use the Cypher query language, the declarative query language native to Neo4j, because it is an easy language to learn and use. Other APIs exist, however, and depending on what we are doing, we may need to prioritize different concerns. It's important to understand the choice of APIs and their capabilities when embarking on a new project. If there is any one thing to take away from this section, it is the notion that these APIs can be thought of as a stack, as depicted in **Figure 6-6**: at the top we prize expressiveness and declarative programming; at the bottom we prize precision, imperative style, and (at the lowest layer) “bare metal” performance.

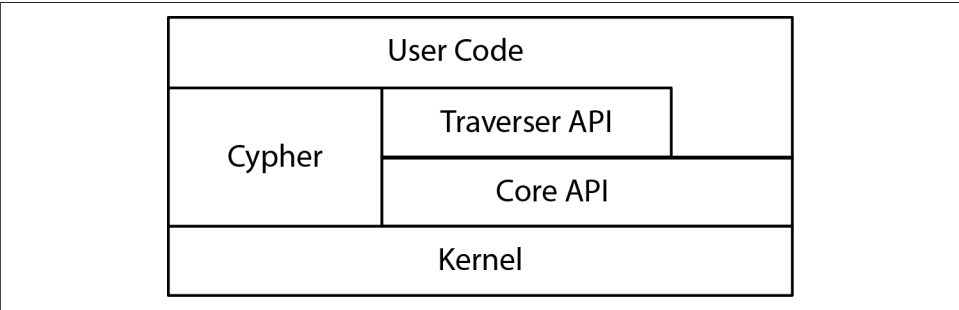


Figure 6-6. Logical view of the user-facing APIs in Neo4j

We discussed Cypher in some detail in **Chapter 3**. In the following sections we'll step through the remaining APIs from the bottom to the top. This API tour is meant to be illustrative. Not all graph databases have the same number of layers, nor necessarily layers that behave and interact in precisely the same way. Each API has its advantages and disadvantages, which you should investigate so you can make an informed decision.

Kernel API

At the lowest level of the API stack are the kernel's **transaction event handlers**. These allow user code to listen to transactions as they flow through the kernel, and thereafter to react (or not) based on the data content and lifecycle stage of the transaction.

Kernel Transaction Event Handlers

A typical use case for transaction event handlers is to prevent physical deletion of records. A handler can be set up to intercept deletion of a node and instead simply mark that node as logically deleted (or in a more sophisticated manner, move the node “back in time” by creating timestamped archive relationships).

Core API

Neo4j’s Core API is an imperative Java API that exposes the graph primitives of nodes, relationships, properties, and labels to the user. When used for reads, the API is lazily evaluated, meaning that relationships are only traversed as and when the calling code demands the next node. Data is retrieved from the graph as quickly as the API caller can consume it, with the caller having the option to terminate the traversal at any point. For writes, the Core API provides transaction management capabilities to ensure atomic, consistent, isolated, and durable persistence.

In the following code, we see a snippet of code borrowed from the [Neo4j tutorial](#) in which we try to find human companions from the *Doctor Who* universe:⁸

```
// Index lookup for the node representing the Doctor is omitted for brevity

Iterable<Relationship> relationships =
    doctor.getRelationships( Direction.INCOMING, COMPANION_OF );

for ( Relationship rel : relationships )
{
    Node companionNode = rel.getStartNode();
    if ( companionNode.hasRelationship( Direction.OUTGOING, IS_A ) )
    {
        Relationship singleRelationship = companionNode
            .getSingleRelationship( IS_A,
                                   Direction.OUTGOING );
        Node endNode = singleRelationship.getEndNode();
        if ( endNode.equals( human ) )
        {
            // Found one!
        }
    }
}
```

This code is very imperative: we simply loop round the Doctor’s companions and check to see if any of the companion nodes have an IS_A relationship to the node rep-

⁸ *Doctor Who* is the world’s longest-running science fiction show and a firm favorite of the Neo4j team.

representing the human species. If the companion node is connected to the human species node, we do something with it.

Because it is an imperative API, the Core API requires us to fine-tune it to the underlying graph structure. This can be very fast. At the same time, however, it means we end up baking knowledge of our specific domain structure into our code. Compared to the higher-level APIs (particularly Cypher) more code is needed to achieve an equivalent goal. Nonetheless, the affinity between the Core API and the underlying record store is plain to see—the structures used at the store and cache level are exposed relatively faithfully by the Core API to user code.

Traversal Framework

The Traversal Framework is a declarative Java API. It enables the user to specify a set of constraints that limit the parts of the graph the traversal is allowed to visit. We can specify which relationship types to follow, and in which direction (effectively specifying relationship filters); we can indicate whether we want the traversal to be performed breadth-first or depth-first; and we can specify a user-defined path evaluator that is triggered with each node encountered. At each step of the traversal, this evaluator determines how the traversal is to proceed next. The following code snippet shows the Traversal API in action:

```
Traversal.description()
    .relationships( DoctorWhoRelationships.PLAYED, Direction.INCOMING )
    .breadthFirst()
    .evaluator( new Evaluator()
    {
        public Evaluation evaluate( Path path )
        {
            if ( path.endNode().hasRelationship(
                DoctorWhoRelationships.REGENERATED_TO ) )
            {
                return Evaluation.INCLUDE_AND_CONTINUE;
            }
            else
            {
                return Evaluation.EXCLUDE_AND_CONTINUE;
            }
        }
    } );
```

With this snippet it's plain to see the predominantly declarative nature of the Traversal Framework. The `relationships()` method declares that only `PLAYED` relationships in the `INCOMING` direction may be traversed. Thereafter, we declare that the traversal should be executed in a `breadthFirst()` manner, meaning it will visit all nearest neighbors before going further outward.

The Traversal Framework is declarative with regard to navigating graph structure. For our implementation of the Evaluator, however, we drop down to the imperative Core API. That is, we use the Core API to determine, given the path to the current node, whether or not further hops through the graph are necessary (we can also use the Core API to modify the graph from inside an evaluator). Again, the native graph structures inside the database bubble close to the surface here, with the graph primitives of nodes, relationships, and properties taking center stage in the API.

Core API, Traversal Framework, or Cypher?

Given these several different methods for querying a graph, which should we choose?

The Core API allows developers to fine-tune their queries so that they exhibit high affinity with the underlying graph. A well-written Core API query is often faster than any other approach. The downside is that such queries can be verbose, requiring considerable developer effort. Moreover, their high affinity with the underlying graph makes them tightly coupled to its structure. When the graph structure changes, they can often break. Cypher can be more tolerant of structural changes—things such as variable-length paths help mitigate variation and change.

The Traversal Framework is both more loosely coupled than the Core API (because it allows the developer to declare informational goals), and less verbose, and as a result a query written using the Traversal Framework typically requires less developer effort than the equivalent written using the Core API. Because it is a general-purpose framework, however, the Traversal Framework tends to perform marginally less well than a well-written Core API query.

If we find ourselves in the unusual situation of coding with the Core API or Traversal Framework (and thus eschewing Cypher and its affordances), it's because we are working on an edge case where we need to finely craft an algorithm that cannot be expressed effectively using Cypher's pattern matching. Choosing between the Core API and the Traversal Framework is a matter of deciding whether the higher abstraction/lower coupling of the Traversal Framework is sufficient, or whether the close-to-the-metal/higher coupling of the Core API is in fact necessary for implementing an algorithm correctly and in accordance with our performance requirements.

This concludes our brief survey of graph programming APIs, using the native Neo4j APIs as an example. We've seen how these APIs reflect the structures used in the lower levels of the Neo4j stack, and how this alignment permits idiomatic and rapid graph traversals.

It's not enough for a database to be fast, however; it must also be dependable. This brings us to a discussion of the nonfunctional characteristics of graph databases.

Nonfunctional Characteristics

At this point we've understood what it means to construct a native graph database, and have seen how some of these graph-native capabilities are implemented, using Neo4j as our example. But to be considered *dependable*, any data storage technology must provide some level of guarantee as to the durability and accessibility of the stored data.⁸

One common measure by which relational databases are traditionally evaluated is the number of transactions per second they can process. In the relational world, it is assumed that these transactions uphold the ACID properties (even in the presence of failures) such that data is consistent and recoverable. For nonstop processing and managing of large volumes, a relational database is expected to scale so that many instances are available to process queries and updates, with the loss of an individual instance not unduly affecting the running of the cluster as a whole.

At a high level at least, much the same applies to graph databases. They need to guarantee consistency, recover gracefully from crashes, and prevent data corruption. Further, they need to scale out to provide high availability, and scale up for performance. In the following sections we'll explore what each of these requirements means for a graph database architecture. Once again, we'll expand on certain points by delving into Neo4j's architecture as a means of providing concrete examples. It should be pointed out that not all graph databases are fully ACID. It is important, therefore, to understand the specifics of the transaction model of your chosen database. Neo4j's ACID transactionality shows the considerable levels of dependability that can be obtained from a graph database—levels we are accustomed to obtaining from enterprise-class relational database management systems.

Transactions

Transactions have been a bedrock of dependable computing systems for decades. Although many NOSQL stores are not transactional, in part because there's an unvalidated assumption that transactional systems scale less well, transactions remain a fundamental abstraction for dependability in contemporary graph databases—including Neo4j. (There is some truth to the claim that transactions limit scalability, insofar as distributed two-phase commit can exhibit unavailability problems in pathological cases, but in general the effect is much less marked than is often assumed.)

Transactions in Neo4j are semantically identical to traditional database transactions. Writes occur within a transaction context, with write locks being taken for consis-

⁸ The formal definition of dependability is the “trustworthiness of a computing system, which allows reliance to be justifiably placed on the service it delivers” as per <http://www.dependability.org/>.

tency purposes on any nodes and relationships involved in the transaction. On successful *completion* of the transaction, changes are flushed to disk for durability, and the write locks released. These actions maintain the atomicity guarantees of the transaction. Should the transaction fail for some reason, the writes are discarded and the write locks released, thereby maintaining the graph in its previous consistent state.

Should two or more transactions attempt to change the same graph elements concurrently, Neo4j will detect a potential deadlock situation, and serialize the transactions. Writes within a single transactional context will not be visible to other transactions, thereby maintaining isolation.

How Transactions Are Implemented in Neo4j

The transaction implementation in Neo4j is conceptually straightforward. Each transaction is represented as an in-memory object whose state represents writes to the database. This object is supported by a lock manager, which applies write locks to nodes and relationships as they are created, updated, and deleted. On transaction roll-back, the transaction object is discarded and the write locks released, whereas on successful completion the transaction is committed to disk.

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

Once a transaction has committed, the system is in a state where changes are guaranteed to be in the database even if a fault then causes a non-pathological failure. This, as we shall now see, confers substantial advantages for recoverability, and hence for ongoing provision of service.

Recoverability

Databases are no different from any other software system in that they are susceptible to bugs in their implementation, in the hardware they run on, and in that hardware's power, cooling, and connectivity. Though diligent engineers try to minimize the possibility of failure in all of these, at some point it's inevitable that a database will crash—though the mean time between failures should be very long indeed.

In a well-designed system, a database server crash, though annoying, ought not affect availability, though it may affect throughput. And when a failed server resumes oper-

ation, it must not serve corrupt data to its users, irrespective of the nature or timing of the crash.

When recovering from an unclean shutdown, perhaps caused by a fault or even an overzealous operator, Neo4j checks in the most recently active transaction log and replays any transactions it finds against the store. It's possible that some of those transactions *may* have already been applied to the store, but because replaying is an idempotent action, the net result is the same: after recovery, the store will be consistent with all transactions successfully committed prior to the failure.

Local recovery is all that is necessary in the case of a single database instance. Generally, however, we run databases in clusters (which we'll discuss shortly) to assure high availability on behalf of client applications. Fortunately, clustering confers additional benefits to recovering instances. Not only will an instance become consistent with all transactions successfully committed prior to its failure, as discussed earlier, it can also quickly catch up with other instances in the cluster, and thereby be consistent with all transactions successfully committed *subsequent* to its failure. That is, once local recovery has completed, a replica can ask other members of the cluster—typically the master—for any newer transactions. It can then apply these newer transactions to its own dataset via transaction replay.

Recoverability deals with the capability of the database to set things right after a fault has arisen. In addition to recoverability, a good database needs to be highly available to meet the increasingly sophisticated needs of data-heavy applications.

Availability

In addition to being valuable in and of themselves, Neo4j's transaction and recovery capabilities also benefit its high-availability characteristics. The database's ability to recognize and, if necessary, repair an instance after crashing means that data quickly becomes available again without human intervention. And of course, more live instances increases the overall availability of the database to process queries.

It's uncommon to want individual disconnected database instances in a typical production scenario. More often, we cluster database instances for high availability. Neo4j uses a master-slave cluster arrangement to ensure that a complete replica of the graph is stored on each machine. Writes are replicated out from the master to the slaves at frequent intervals. At any point, the master and some slaves will have a completely up-to-date copy of the graph, while other slaves will be catching up (typically, they will be but milliseconds behind).

For writes, the classic write-master with read-slaves is a popular topology. With this setup, all database writes are directed at the master, and read operations are directed at slaves. This provides asymptotic scalability for writes (up to the capacity of a single

spindle) but allows for near linear scalability for reads (accounting for the modest overhead in managing the cluster).

Although write-master with read-slaves is a classic deployment topology, Neo4j also supports writing through slaves. In this scenario, the slave to which a write has been directed by the client first ensures that it is consistent with the master (it “catches up”); thereafter, the write is synchronously transacted across both instances. This is useful when we want immediate durability in two database instances. Furthermore, because it allows writes to be directed to any instance, it offers additional deployment flexibility. This comes at the cost of higher write latency, however, due to the forced catchup phase. It does *not* imply that writes are distributed around the system: all writes must still pass through the master at some point.

Other Replication Options in Neo4j

In Neo4j version 1.8 onward it’s possible to specify that writes to the master are replicated in a best-effort manner to an arbitrary number of replicas before a transaction is considered complete. This provides an alternative to the “at least two” level of durability achieved by writing through slaves. See [“Replication” on page 81](#) for more details.

Another aspect of availability is contention for access to resources. An operation that contends for exclusive access (e.g., for writes) to a particular part of the graph may suffer from sufficiently high latency as to appear unavailable. We’ve seen similar contention with coarse-grained table-level locking in RDBMSs, where writes are latent even when there’s logically no contention.

Fortunately, in a graph, access patterns tend to be more evenly spread, especially where idiomatic graph-local queries are executed. A graph-local operation is one that starts at one or more given places in the graph and then traverses the surrounding subgraphs. The starting points for such queries tend to be things that are especially significant in the domain, such as users or products. These starting points result in the overall query load being distributed with low contention. In turn, clients perceive greater responsiveness and higher availability.

The Benefits of Idiomatic Queries

Jackie Stewart, the Formula 1 racing driver, is reputed to have said that to drive a car well you don't need to be an engineer but you do need mechanical sympathy. That is, the best performance comes as a result of the driver and car working together harmoniously.

In much the same way, graph database queries are considered mechanically sympathetic to the database when they are framed as idiomatic, graph-local queries that begin their traversal from one or more start points. The underlying infrastructure, including caching and store access, is optimized to support this kind of workload.

Idiomatic queries have beneficial side effects. For example, because caching is aligned with idiomatic searches, queries that are themselves idiomatic tend to exploit caches better and run faster than nonidiomatic queries. In turn, queries that run fast free up the database to run more of them, which means higher throughput and the sense of better availability from the client's point of view because there's less waiting around.

Unidiomatic queries (e.g., those which pick random nodes/relationships rather than traversing) exhibit the opposite characteristics: they disrespect the underlying caching layers and therefore run more slowly because more disk I/O is needed. Because the queries run slowly, the database can process fewer of them per second, which means the availability of the database to do useful work diminishes from the client's point of view.

Whatever the database, understanding the underlying storage and caching infrastructure will help us construct idiomatic—and hence, mechanically sympathetic—queries that maximize performance.

Our final observation on availability is that scaling for cluster-wide replication has a positive impact, not just in terms of fault-tolerance, but also responsiveness. Because there are many machines available for a given workload, query latency is low and availability is maintained. But as we'll now discuss, scale itself is more nuanced than simply the number of servers we deploy.

Scale

The topic of scale has become more important as data volumes have grown. In fact, the problems of data at scale, which have proven difficult to solve with relational databases, have been a substantial motivation for the NOSQL movement. In some sense, graph databases are no different; after all, they also need to scale to meet the workload demands of modern applications. But scale isn't a simple value like transactions per second. Rather, it's an aggregate value that we measure across multiple axes.

For graph databases, we will decompose our broad discussion on scale into three key themes:

1. Capacity (graph size)
2. Latency (response time)
3. Read and write throughput

Capacity

Some graph database vendors have chosen to eschew any upper bounds in graph size in exchange for performance and storage cost. Neo4j has taken a somewhat unique approach historically, having maintained a “sweet spot” that achieves faster performance and lower storage (and consequently diminished memory footprint and IO-ops) by optimizing for graph sizes that lie at or below the 95th percentile of use cases. The reason for the trade-off lies in the use of fixed record sizes and pointers, which (as discussed in “[Native Graph Storage](#)” on page 152) it uses extensively inside of the store. At the time of writing, the current release of Neo4j can support single graphs having tens of billions of nodes, relationships, and properties. This allows for graphs with a social networking dataset roughly the size of Facebook’s.



The Neo4j team has publicly expressed the intention to support 100B+ nodes/relationships/properties in a single graph as part of its roadmap.

How large must a dataset be to take advantage of all of the benefits a graph database has to offer? The answer is, smaller than you might think. For queries of second or third degree, the performance benefits show with datasets having a few single-digit thousand nodes. The higher the degree of the query, the more extreme the delta. The ease-of-development benefits are of course unrelated to data volume, and available regardless of the database size. The authors have seen meaningful production applications range from as small as a few tens of thousands of nodes, and a few hundred thousand relationships, to billions of nodes and relationships.

Latency

Graph databases don’t suffer the same latency problems as traditional relational databases, where the more data we have in tables—and in indexes—the longer the join operations (this simple fact of life is one of the key reasons that performance tuning is nearly always the very top issue on a relational DBA’s mind). With a graph database, most queries follow a pattern whereby an index is used simply to find a starting node (or nodes). The remainder of the traversal then uses a combination of pointer chasing

and pattern matching to search the data store. What this means is that, unlike relational databases, performance does not depend on the total size of the dataset, but only on the data being queried. This leads to performance times that are nearly constant (that is, are related to the size of the result set), even as the size of the dataset grows (though as we discussed in [Chapter 3](#), it's still sensible to tune the structure of the graph to suit the queries, even if we're dealing with lower data volumes).

Throughput

We might think a graph database would need to scale in the same way as other databases. But this isn't the case. When we look at IO-intensive application behaviors, we see that a single complex business operation typically reads and writes a set of related data. In other words, the application performs multiple operations on a logical sub-graph within the overall dataset. With a graph database such multiple operations can be rolled up into larger, more cohesive operations. Further, with a graph-native store, executing each operation takes less computational effort than the equivalent relational operation. Graphs scale by doing less work for the same outcome.

For example, imagine a publishing scenario in which we'd like to read the latest piece from an author. In a RDBMS we typically select the author's works by joining the authors table to a table of publications based on matching author ID, and then ordering the publications by publication date and limiting to the newest handful. Depending on the characteristics of the ordering operation, that might be a $O(\log(n))$ operation, which isn't so very bad.

However, as shown in [Figure 6-7](#), the equivalent graph operation is $O(1)$, meaning constant performance irrespective of dataset size. With a graph we simply follow the outbound relationship called `WROTE` from the author to the work at the head of a list (or tree) of published articles. Should we wish to find older publications, we simply follow the `PREV` relationships and iterate through a linked list (or, alternatively, recurse through a tree). Writes follow suit because we always insert new publications at the head of the list (or root of a tree), which is another constant time operation. This compares favorably to the RDBMS alternative, particularly because it naturally maintains constant time performance for reads.

Of course, the most demanding deployments will overwhelm a single machine's capacity to run queries, and more specifically its I/O throughput. When that happens, it's straightforward to build a cluster with Neo4j that scales horizontally for high availability and high read throughput. For typical graph workloads, where reads far outstrip writes, this solution architecture can be ideal.

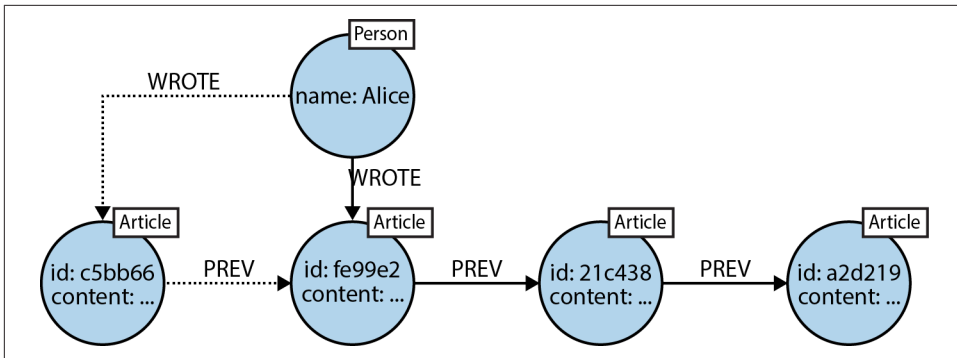


Figure 6-7. Constant time operations for a publishing system

Should we exceed the capacity of a cluster, we can spread a graph across database instances by building *sharding* logic into the application. Sharding involves the use of a synthetic identifier to join records across database instances at the application level. How well this will perform depends very much on the shape of the graph. Some graphs lend themselves very well to this. Mozilla, for instance, uses the Neo4j graph database as part of its next-generation cloud browser, Pancake. Rather than having a single large graph, it stores a large number of small independent graphs, each tied to an end user. This makes it very easy to scale.

Of course, not all graphs have such convenient boundaries. If our graph is large enough that it needs to be broken up, but no natural boundaries exist, the approach we use is much the same as what we would use with a NOSQL store like MongoDB: we create synthetic keys, and relate records via the application layer using those keys plus some application-level resolution algorithm. The main difference from the MongoDB approach is that a native graph database will provide you with a performance boost anytime you are doing traversals within a database instance, whereas those parts of the traversal that run between instances will run at roughly the same speed as a MongoDB join. Overall performance should be markedly faster, however.

The Holy Grail of Graph Scalability

The future goal of most graph databases is to be able to partition a graph across multiple machines without application-level intervention, so that read and write access to the graph can be scaled horizontally. In the general case this is known to be an NP Hard problem, and thus impractical to solve.

A naïve solution to the problem can lead to unpredictable query times as a result of graph traversals unexpectedly jumping between machines over the (slow) network. A sensible implementation, by contrast, understands *minimum point cut* within the

context of a particular domain, thereby minimizing cross-machine traversals.¹ While there is exciting research work going on in the area at the time of writing, there is no database yet that provably supports this behavior.

Summary

In this chapter we've shown how property graphs are an excellent choice for pragmatic data modeling. We've explored the architecture of a graph database, with particular reference to the architecture of Neo4j, and discussed the nonfunctional characteristics of graph database implementations and what it means for them to be dependable.

¹ See [*http://en.wikipedia.org/wiki/Cut_\(graph_theory\)*](http://en.wikipedia.org/wiki/Cut_(graph_theory))