Graph Databases

Robinson, I. et al. (2015), Graph Databases (2nd ed.).

Elvis Nava

Chapter 1 - Introduction

The Labeled Property Graph model
A labeled property graph has the following characteristics:

- It contains nodes and relationships.
- Nodes contain properties (key-value pairs).
- Nodes can be labeled with one or more labels.
- Relationships are named and directed, and always have a start and end node.
- Relationships can also contain properties.

Graph Databases
A graph database management system (henceforth, a graph database) is an online database management system with Create, Read, Update, and Delete (CRUD) methods that expose a graph data model. Graph databases are generally built for use with transactional (OLTP) systems. Accordingly, they are normally optimized for transactional performance, and engineered with transactional integrity and operational availability in mind.

There are two properties of graph databases we should consider when investigating graph database technologies:

- The underlying storage.
  Some graph databases use native graph storage that is optimized and designed for storing and managing graphs.
- The processing engine.
  Some definitions require that a graph database use index-free adjacency, meaning that connected nodes physically “point” to each other in the database.
Chapter 2 - Options for storing connected data

Relational Databases lack relationships

Relationships do exist in the vernacular of relational databases, but only at modeling time, as a means of joining tables. In our treatment of connected data, we often need to disambiguate the semantics of the relationships that connect entities, as well as qualify their weight or strength. Relational relations do nothing of the sort. Worse still, as outlier data multiplies, and the overall structure of the dataset becomes more complex and less uniform, the relational model becomes burdened with large join tables, sparsely populated rows, and lots of null-checking logic.

Example: computing friends-of-friends on a social network stored on a relational database in the form of an adjacency table is very computationally expensive.

NOSQL Databases also lack relationships

Most NOSQL databases—whether key-value-, document-, or column-oriented—store sets of disconnected documents/values/columns. This makes it difficult to use them for connected data and graphs.

One well-known strategy for adding relationships to such stores is to embed an aggregate’s identifier inside the field belonging to another aggregate—effectively introducing foreign keys. But this requires joining aggregates at the application level, which quickly becomes prohibitively expensive.

Graph Databases embrace relationships

The previous examples have dealt with implicitly connected data. As users we infer semantic dependencies between entities, but the data models—and the databases themselves—are blind to these connections. To compensate, our applications must create a network out of the flat, disconnected data at hand, and then deal with any slow queries and latent writes across denormalized stores that arise.

What we really want is a cohesive picture of the whole, including the connections between elements. In contrast to the stores we’ve just looked at, in the graph world, connected data is stored as connected data.

Relationships in a graph naturally form paths. Querying the graph involves following paths. Because of the fundamentally path-oriented nature of the data model, the majority of path-based graph database operations are highly aligned with the way in which the data is laid out, making them extremely efficient.
Chapter 3 - Data modeling with graphs

An introduction to Cypher

Cypher is a graph database query language, currently specific to Neo4j.

Cypher enables a user to ask the database to find data that matches a specific pattern. Example:

![Cypher diagram]

Is a pattern matched by:

```
(emil)<-[KNOWS]-{jim}-[KNOWS]->{ian}-[KNOWS]->(emil)
```

*ian, jim, and emil* are identifiers. Identifiers allow us to refer to the same node more than once when describing a pattern.

We can also specify property values and node labels:

```
(emil:Person {name:'Emil'})
<-[:KNOWS]-{jim:Person {name:'Jim'}}
-[:KNOWS]->{ian:Person {name:'Ian'}}
-[:KNOWS]->(emil)
```

A Cypher query anchors one or more parts of a pattern to specific locations in a graph using predicates, and then flexes the unanchored parts around to find local matches.

**The MATCH Clause**

The MATCH clause precedes the drawn ASCII pattern to match. The pattern itself may specify the anchoring, or it may be specified in the WHERE clause.
For example:

MATCH (a:Person)-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c)
WHERE a.name = 'Jim'
RETURN b, c

The RETURN Clause
This clause specifies which nodes, relationships, and properties in the matched data should be returned to the client.

The other clauses we can use in a Cypher query include:

- **WHERE**
  Provides criteria for filtering pattern matching results.
- **CREATE** and **CREATE UNIQUE**
  Create nodes and relationships.
- **MERGE**
  Ensures that the supplied pattern exists in the graph, either by reusing existing nodes and relationships that match the supplied predicates, or by creating new nodes and relationships.
- **DELETE**
  Removes nodes, relationships, and properties.
- **SET**
  Sets property values.
- **FOREACH**
  Performs an updating action for each element in a list.
- **UNION**
  Merges results from two or more queries.
- **WITH**
  Chains subsequent query parts and forwards results from one to the next. Similar to piping commands in Unix.
- **START**
  Specifies one or more explicit starting points—nodes or relationships—in the graph. (START is deprecated in favor of specifying anchor points in a MATCH clause.)

A Comparison of Relational and Graph Modeling

When we model a domain using the Relational Model, we introduce complexity in the form of additional constraints such as foreign keys. Moreover, a normalized model may be too slow for real time use, so we may need to denormalize data, which sometimes isn’t a trivial task, and also introduces substantial redundancy. In addition, the schema, being rigid, is not easily changeable in the future.

In Graph Modeling, we can have a performant model closely aligned with the domain, supporting evolution while also maintaining data integrity. After agreeing to the domain of the model, it is not transformed into tables, but simply maintained as-is and enriched with further labels and properties.
Cross-Domain Models

Graph Modeling allows us to easily merge different domains in a single graph, allowing us to make cross-domain queries.

Indexes and Constraints

To support efficient node lookup, Cypher allows us to create indexes per label and property combinations.

```
CREATE INDEX ON :Venue(name)
```

For unique property values we can also specify constraints that assure uniqueness.

```
CREATE CONSTRAINT ON (c:Country) ASSERT c.name IS UNIQUE
```
Chapter 4 - Building a Graph Database Application

Application Architecture

Embedded vs Server Neo4j:
In **embedded** mode, Neo4j runs in the same process as our application. Embedded Neo4j is ideal for hardware devices, desktop applications, and for incorporating in our own application servers. Running Neo4j in **server** mode is the most common means of deploying the database today. At the heart of each server is an embedded instance of Neo4j.

Neo4j **clusters** for high availability and horizontal read scaling using master-slave replication.

- Replication:
  Although all writes to a cluster are coordinated through the master, Neo4j does allow writing through slaves, but even then, the slave that’s being written to syncs with the master before returning to the client.

- Buffer writes using queues:
  In high write load scenarios, we can use queues to buffer writes and regulate load. With this strategy, writes to the cluster are buffered in a queue. A worker then polls the queue and executes batches of writes against the database. Not only does this regulate write traffic, but it reduces contention and enables us to pause write operations without refusing client requests during maintenance periods.

- Global Clusters:
  A multiregion cluster enables us to service reads from the portion of the cluster geographically closest to the client. In these situations, however, the latency introduced by the physical separation of the regions can sometimes disrupt the coordination protocol.

Load Balancing:

- Separate read traffic from write traffic.
  Given the recommendation to direct the majority of write traffic to the master, we should consider clearly separating read requests from write requests.

- Cache sharding.
  Queries run fastest when the portions of the graph needed to satisfy them reside in main memory. When a graph holds many billions of nodes, relationships, and properties, not all of it will fit into main memory.
  One solution is to use a technique called cache sharding, which consists of routing each request to a database instance in an HA cluster where the portion of the graph necessary to satisfy that request is likely already in main memory.
Chapter 6 - Graph Database Internals

Native Graph Processing

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.

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.

Native Graph Storage

Neo4j architecture:

Traversal API Core API Cypher

Lock Manager Transaction Management

Page Cache

Record Files Transaction Log

Disks

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).

Node (15 bytes)

<table>
<thead>
<tr>
<th>inUse</th>
<th>nextRelid</th>
<th>nextPropId</th>
<th>labels</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>14</td>
</tr>
</tbody>
</table>

Relationship (34 bytes)

<table>
<thead>
<tr>
<th>inUse</th>
<th>firstPrevRelId</th>
<th>secondPrevRelId</th>
<th>nextPropId</th>
<th>firstNode</th>
<th>secondNode</th>
<th>relationshipType</th>
<th>firstNextRelId</th>
<th>secondNextRelId</th>
<th>firstInChainMarker</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>17</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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`. 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 $O(\log n)$.

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.

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 properties (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.

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.
Programmatic APIs

APIs in Neo4j:

<table>
<thead>
<tr>
<th>User Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traverser API</td>
</tr>
<tr>
<td>Core API</td>
</tr>
<tr>
<td>Kernel</td>
</tr>
</tbody>
</table>

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. 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.

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 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.

ACID in Graph databases
Not all graph databases products guarantee ACID like relational databases.

Neo4j fully supports Transactions. Transactions in Neo4j are semantically identical to traditional database transactions.

Recoverability: When recovering from an unclean shutdown, 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.

**Availability:** In addition to being valuable in and of themselves, Neo4j’s transaction and recovery capabilities also benefit its high-availability characteristics. Neo4j uses a master-slave cluster arrangement to ensure that a complete replica of the graph is stored on each machine. Although write-master with read-slaves is a classic deployment topology, Neo4j also supports writing through slaves.

**Scale**

**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 by optimizing for graph sizes that lie at or below the 95th percentile of use cases.

**Latency:** 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.

**Throughput:** In general, graph databases have less requirements for the same amount of work, relative to relational databases. 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.
14. Graph Databases

Chapter 1: Introduction

A graph is a set of nodes (vertices) and relationships (edges) that connect them. There are directed and undirected graphs.

A labeled property graph has the following characteristics:

- it contains nodes and relationships
- nodes contain properties (key-value pairs)
- nodes can be labeled with one or more labels
- relationships are named and directed, and always have a start and end node

Graph database management system: or graph database, online database management system with Create, Read, Update, and Delete (CRUD) methods that expose a graph data model; generally built for use with transactional (OLTP) systems

Native graph storage: optimized and designed for storing and managing graphs (other ways to store are: serializing graph into a relational or object-oriented database, or some other general purpose data store)

Index-free adjacency: connected nodes physically “point” to each other in the database

Graph compute engines: a technology that enables global graph computational algorithms to be run against large datasets; designed to e.g. identify clusters in the data

Reasons to use graph databases:

- performance: join-intensive queries perform badly when dataset grows in relational databases; in graph databases when queries are localized to a portion of the graph, performance only depends on subgraph size and not overall size -> better performance
- flexibility: more flexible data model, especially useful when real shape of data not known beforehand
- agility: schema-freeness and testability of graph database’s API align well with today’s agile and test-driven software deployment practices

Chapter 2: Options for storing connected Data

Accommodating connected, semi-structured datasets inside other database structures:

- relational databases:
  as data becomes more complex and less uniform, tables grow large, but with sparsely populated rows -> overhead when joining, lots of null-checking logic
  relational databases struggle with highly connected domains

- NOSQL databases (key-value-, document-, or column-oriented):
  Also lack relationships; well-known strategy for adding relationships: embed an aggregate’s identifier inside field belonging to another aggregate (like foreign keys); BUT: requires joining aggregates at application level (expensive)
  lacks back-pointers (unable to run some queries); can be solved by also including the back information in the value; BUT: still traversing is expensive (since aggregates have no sense of locality), and disk utilization is increased due the additionally stored metadata

- graph database: relationships in a graph naturally form paths, querying and traversing involves following the path TODO
Chapter 3: Data Modelling with Graphs

Since graphs are very natural and intuitive (“whiteboard friendly”, “we tend to abstract details using circles and boxes and connecting them with arrows and lines”), the gap between our conceptualization of the world and the database’s instantiation of that model becomes smaller (as opposed to relational data management techniques).

**Querying graphs: An introduction to Cypher**

Cypher is a graph database query language. Cypher is designed to be easily read and understood by developers, database professionals and business stakeholders. It enables users to ask for data that matched a specific pattern.

A cypher query anchors one or more patterns to specific locations in the graph (using predicates, based on labels and property predicates in the query, using metainformation about existing indices, constraints, predicates) and then flexes unanchored parts around to find local matches.

Cypher Clauses (lot of common clauses with SQL with same meaning):

- **MATCH**: does the pattern matching mentioned above, e.g.:
  
  ```cypher
  MATCH (a:Person)-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c)
  WHERE a.name = 'Jim'
  RETURN b, c
  ```

- **RETURN**: specifies which nodes, relationships, and properties in the matched data should be returned

- **WHERE**: provides criteria for filtering pattern matching results

- **CREATE** and **CREATE UNIQUE**: creates nodes and relationships

- **MERGE**: ensures that the supplied pattern exists in the graph, either by reusing existing nodes and relationships that match the supplied predicates, or by creating new ones

- **DELETE**: removes nodes, relationships, and properties

- **SET**: set property values

- **FOREACH**: performs an updating action for each element in the list

- **UNION**: merges results from two or more queries

- **WITH**: chains subsequent query parts (query chaining) and forwards results from one to the next. Similar to piping commands in Unix.

  ```cypher
  MATCH (bard:Author {lastname:'Shakespeare'})-[w:WROTE_PLAY]->(play)
  WITH play
  ORDER BY w.year DESC
  RETURN collect(play.title) AS plays
  ```

- **START**: Specifies one or more explicit starting points—nodes or relationships—in the graph. (START is deprecated in favor of specifying anchor points in a MATCH clause.)
A comparison of Relational and Graph Modeling

To introduce graph modeling the author of the book models a domain using relational- and graph-based techniques. The domain is a simple data center management domain, apps (as clients), servers, load balancers, racks for servers etc.

Modeling such a domain with a relational database includes the step of first finding a suitable logical model. Then the model is mapped into tables and relations (normalized to reduce data redundancy). But then to be efficient the data is denormalized (avoid expensive joins).

The problem with such a database is that any changes of the model, result in the steps: design, normalize, denormalize. This makes database refactorings or migration (introducing structural changes into a database) slow, risky, and expensive.

However, modeling the domain with a graph database is easier, since domain modeling is isomorphic to graph modeling. Thus, just adding labels, and properties to the original model is the only thing needs to be done. This allows for changes in the domain to be easily adopted in the graph model. (Pictures below: left: original domain; right: graph model)
Cross-Domain Models

Business insight often depends on us understanding the hidden network effects at play in a complex value chain. To generate this understanding, we need to join domains together without distorting or sacrificing the details particular to each domain. Property graphs provide a solution here. Using a property graph, we can model a value chain as a graph of graphs in which specific relationships connect and distinguish constituent subdomains.

The graph on the right side describes and connects 3 different domains (distinguishable by the differently formatted relationships):

dotted for the literary domain, solid for the theatrical domain, and dashed for the geospatial domain

(In the book, the following subchapters are dedicated to show how to create and query this graph using Cypher. See the book for more details and examples.)

Common Modeling Pitfalls

Creating models by adopting relationships described in our lax and imprecise everyday language. For example, the sentence: “Bob sent an email to Charlie” is for convenience shortened to “Bob emailed Charlie”. If we were to model according to the second phrase, we would introduce a EMAILED relationship between Bob and Charlie. But this means we would lose information like the content of the email, date and time, etc. Of course, these could be added as properties to the EMAILED relationship. But this does not solve the problem that we would lose information like people in CC, etc.

The better solution would be to introduce a node Email, resulting in a star-shaped graph structure.
The next difficulty is migration. Since the underlying model may change, migration cannot be completely avoided.

Adding to the graph, using new kinds of relationships, will not affect any existing queries, and is completely safe. Changing the graph, using existing relationship types, and changing the properties (not just the property values) of existing nodes might be safe.

This chapter includes a detailed example on how to build a model. Please see the book for more details.

**Identifying Nodes and Relationships**

The modeling process can best be summed up as an attempt to create a graph structure that expresses the questions we want to ask of our domain. That is, design for queryability:

1. Describe the client or end-user goals that motivate our model.
2. Rewrite these goals as questions to ask of our domain.
3. Identify the entities and the relationship that appear in these questions.
4. Translate these entities and relationships into Cypher path expressions.
5. Express the questions we want to ask of our domain as graph patterns using path expressions similar to the ones we used to model the domain.

By examining the language, we use to describe our domain, we can very quickly identify the core elements in our graph:

- Common nouns become labels: “user” and “email,” for example, become the labels User and Email.
- Verbs that take an object become relationship names: “sent” and “wrote,” for example, become SENT and WROTE.
- A proper noun—a person or company’s name, for example—refers to an instance of a thing, which we model as a node, using one or more properties to capture that thing’s attributes.

**Avoiding Anti-Patterns**

- In general, don’t encode entities into relationships (see email example above).
- Graphs are naturally additive structures. Thus, it is quite natural to add facts in form of new nodes. It’s bad practice to conflate data elements at write time to preserve query-time efficiency. In general, if modeled in accordance of the questions that will be asked of the data, adding more data should not decrease performance.

**Chapter 4: Building a Graph Database Application**

This chapter is about building a graph database application.

The first question it discusses is: how to build a data model? It has some guidelines about when to use nodes and when relationships, how fine-grained or generic a relationship should be, and closes with some examples (e.g. how to model time).

The next section is about the application architecture. It contrasts the two modes databases can run in: as a server accessed through a client library or embedded.

It also covers testing.

The last topic covered is importing and bulk loading data (using batch import).

**Chapter 6: Graph Database Internals**
This chapter discusses the implementation of graph databases. Since there is no single universal architecture pattern, the chapter describes the most common architecture patterns and components expected to find inside a graph database.

**Native Graph Processing**

A graph database has native processing capabilities if it exhibits a property called *index-free adjacency* (each node maintains direct references to its adjacent nodes, acting therefore as a micro-index of other nearby nodes, which is much cheaper than global indexes).

A non-native graph database engine uses (global) indexes to link nodes together. These indexes add a layer of indirection. Native engines use relationships, not indexes. (picture below)

<table>
<thead>
<tr>
<th></th>
<th>Native</th>
<th>Non-native</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate relationship</td>
<td>O(1)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>Traverse network of m steps</td>
<td>O(m)</td>
<td>O(m log n)</td>
</tr>
</tbody>
</table>

Additionally, with index-free adjacency, bidirectional joins are effectively precomputed and stored in the database as relationships.

As opposed to this, using indexes to simulate connections becomes problematic when trying to traverse in the “opposite” direction from the one for which the index was constructed. Now one can either create a reverse-lookup index for each traversal scenario, or perform a brute-force search through the original index, which is an $O(n)$ operation.

Thus, indexes work well for small graphs but become inefficient for large graphs.

**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.

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.
Most of the Neo4j store files have fixed-sized records which enables fast lookups, since given an ID, by simply multiplying it with the record size one knows where a record begins. Thus, the database can directly compute a record's location, at cost $O(1)$, rather than performing an index search, which would be cost $O(\log n)$.

For example, in Neo4j, the node store is a fixed-size record store, where each record is nine bytes in length. If we have a node with id 100, then we know its record begins 900 bytes into the file.

Records in the property store are physically stored in the neostore.propertystore.db file (also fixed sized records). Each property record consists of four property blocks and the ID of the next property in the property chain (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. Thus, a property record can 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 (more efficient since only single file accessed). 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.

Further Optimizations that can be done are in-memory caching (to provide probabilistic low-latency access to the graph). From Neo4j 2.2, an off-heap cache is used. As of Neo4j 2.2, Neo4j uses an LRU-K page cache. That means, 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**

In addition to Cypher, there are some more APIs used manipulate the graph database.

**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. (typical use case: prevent physical deletion of records by intercepting the deletion and simply marking the records as logically deleted)

**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 (relationships 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. The caller has 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

The highly imperative API allows for fine-tuning, but also requires thorough knowledge of the domain. The code is also quite domain-specific. But if well-written, often faster than any other approach.

**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. By manipulating the path evaluator, one drops down to the imperative Core API.

The code is usually more loosely coupled, less verbose and, “easier”.

**Nonfunctional Characteristics**

A dependable data storage technology must provide some level of guarantee as to the durability and accessibility of the stored data. Also, number of transactions per second, the upholding of the ACID properties (in relational world), and scalability are further measures.

**Transactions**: Transactions in Neo4j are semantically identical to traditional database transactions. Each transaction is represented as an in-memory object, its 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 (for consistency). On transaction rollback, the transaction object is discarded, and the write locks released (graph in its previous consistent state), whereas on successful completion the transaction is committed to disk (durability).

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.

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.

**Recoverability**: When recovering from an unclean shutdown (crash), 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.

In the case of a single database instance only local recovery is necessary. Generally, however, databases are run in clusters which has additional benefits to recovering instances. In a cluster situation, an instance can become completely consistent (also with transactions after failure) by asking replicas for new transactions and applying them.

**Availability**: Neo4j uses a master-slave cluster arrangement for high availability. Writes are replicated out from the master to the slaves at a frequent interval. But Neo4j also supports writing through slaves. In this case, the slave first needs to ensure that it is consistent with the master (it “catches up”) and the write is transacted across both instances. This provides immediate durability in two database instances (master and slave instances).

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 starve. 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.

Scale:

- **Capacity:** Some graph databases have 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 I/O-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”) it uses extensively inside of the store. The Neo4j team has publicly expressed the intention to support 100B+ nodes/relationships/properties in a single graph as part of its roadmap.

- **Latency:** Graph databases don’t suffer the same latency problems as traditional relational databases (no expensive joins). This means performance does not depend on total size of dataset, but only on data being queried.

- **Throughput:** By avoiding joins and using cheap traversals, a graph database performs better than a relational database. But, the most demanding deployments will overwhelm a single machine’s capacity to run queries, and more specifically its I/O throughput. In that case, 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. Should we exceed the capacity of a cluster, we can spread a graph across database instances by building sharding logic into the application. This means instead of a large graph, we have a large number of independent small graphs. In case, the graph has no natural way to split it, use same approach as with a NOSQL store: we create synthetic keys, and relate records via the application layer using those keys plus some application-level resolution algorithm. Then traversals within a database instance is still very performant, but traversals between instances will run at roughly the same speed as a MongoDB join.