MODULE 5

Graph Databases

Graph databases allow you to store entities and relationships between these entities. Entities are also known as nodes, which have properties. Think of a node as an instance of an object in the application. Relations are known as edges that can have properties. Edges have directional significance; nodes are organized by relationships which allow you to find interesting patterns between the nodes. The organization of the graph lets the data to be stored once and then interpreted in different ways based on relationships.

11.1 What Is a Graph Database?

In the example graph in Figure 11.1, we see a bunch of nodes related to each other. Nodes are entities that have properties, such as `name`. The node of `Martin` is actually a **node** that has **property** of `name` set to `Martin`.

We also see that edges have types, such as `likes`, `author`, and so on. These properties let us organize the nodes; for example, the nodes `Martin` and `Pramod` have an **edge** connecting them with a relationship type of `friend`. Edges can have multiple properties. We can assign a property of `since` on the `friend` relationship type between `Martin` and `Pramod`. Relationship types have directional significance; the `friend` relationship type is bidirectional but `likes` is not. When `Dawn likes NoSQL Distilled`, it does not automatically mean `NoSQL Distilled likes Dawn`.

Once we have a graph of these nodes and edges created, we can query the graph in many ways, such as "get all nodes employed by `Big Co` that `like NoSQL Distilled`." A query on the graph is also known as **traversing** the graph. An advantage of the graph databases is that we can change the traversing requirements without having to change the nodes or edges. If we want to "get all nodes that `like NoSQL Distilled`," we can do so without having to change the existing data or the model of the database, because we can traverse the graph any way we like.

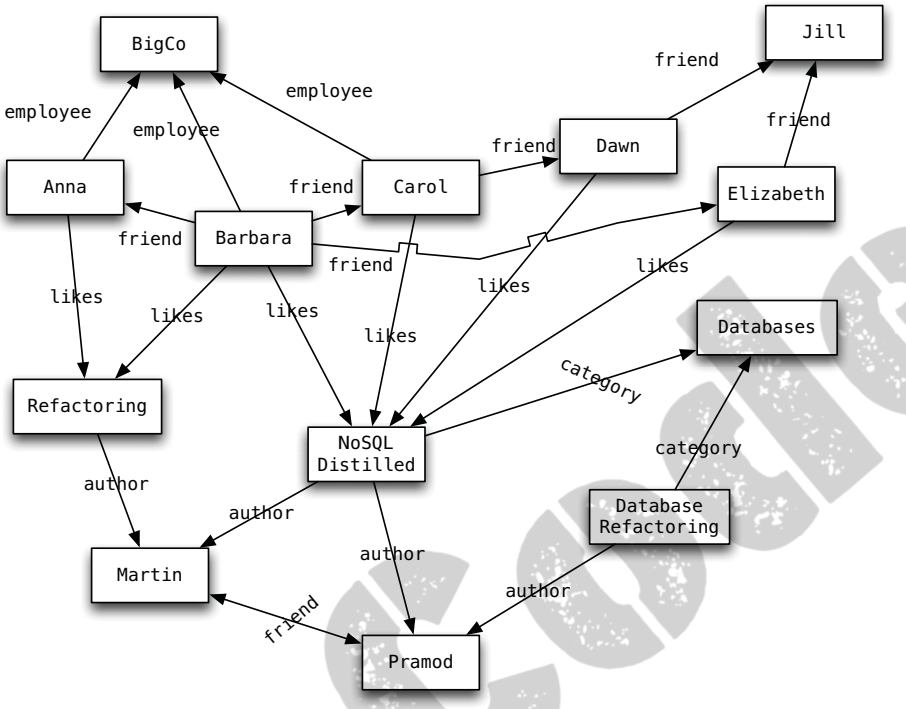

Figure 11.1 *An example graph structure*

Usually, when we store a graph-like structure in RDBMS, it's for a single type of relationship ("who is my manager" is a common example). Adding another relationship to the mix usually means a lot of schema changes and data movement, which is not the case when we are using graph databases. Similarly, in relational databases we model the graph beforehand based on the `Traversal` we want; if the `Traversal` changes, the data will have to change.

In graph databases, traversing the joins or relationships is very fast. The relationship between nodes is not calculated at query time but is actually persisted as a relationship. Traversing persisted relationships is faster than calculating them for every query.

Nodes can have different types of relationships between them, allowing you to both represent relationships between the domain entities and to have secondary relationships for things like category, path, time-trees, quad-trees for spatial indexing, or linked lists for sorted access. Since there is no limit to the number and kind of relationships a node can have, they can all be represented in the same graph database.

11.2 Features

There are many graph databases available, such as Neo4J [Neo4J], Infinite Graph [Infinite Graph], OrientDB [OrientDB], or FlockDB [FlockDB] (which is a special case: a graph database that only supports single-depth relationships or adjacency lists, where you cannot traverse more than one level deep for relationships). We will take Neo4J as a representative of the graph database solutions to discuss how they work and how they can be used to solve application problems.

In Neo4J, creating a graph is as simple as creating two nodes and then creating a relationship. Let's create two nodes, `Martin` and `Pramod`:

```
Node martin = graphDb.createNode();
martin.setProperty("name", "Martin");

Node pramod = graphDb.createNode();
pramod.setProperty("name", "Pramod");
```

We have assigned the `name` property of the two nodes the values of `Martin` and `Pramod`. Once we have more than one node, we can create a relationship:

```
martin.createRelationshipTo(pramod, FRIEND);

pramod.createRelationshipTo(martin, FRIEND);
```

We have to create relationship between the nodes in both directions, for the direction of the relationship matters: For example, a `product` node can be liked by `user` but the `product` cannot like the `user`. This directionality helps in designing a rich domain model (Figure 11.2). Nodes know about `INCOMING` and `OUTGOING` relationships that are traversable both ways.

Relationships are first-class citizens in graph databases; most of the value of graph databases is derived from the relationships. Relationships don't only have a type, a start node, and an end node, but can have properties of their own. Using these properties on the relationships, we can add intelligence to the relationship—for example, since when did they become friends, what is the distance between the nodes, or what aspects are shared between the nodes. These properties on the relationships can be used to query the graph.

Since most of the power from the graph databases comes from the relationships and their properties, a lot of thought and design work is needed to model the relationships in the domain that we are trying to work with. Adding new relationship types is easy; changing existing nodes and their relationships is similar to data migration ("Migrations in Graph Databases," p. 131), because these changes will have to be done on each node and each relationship in the existing data.

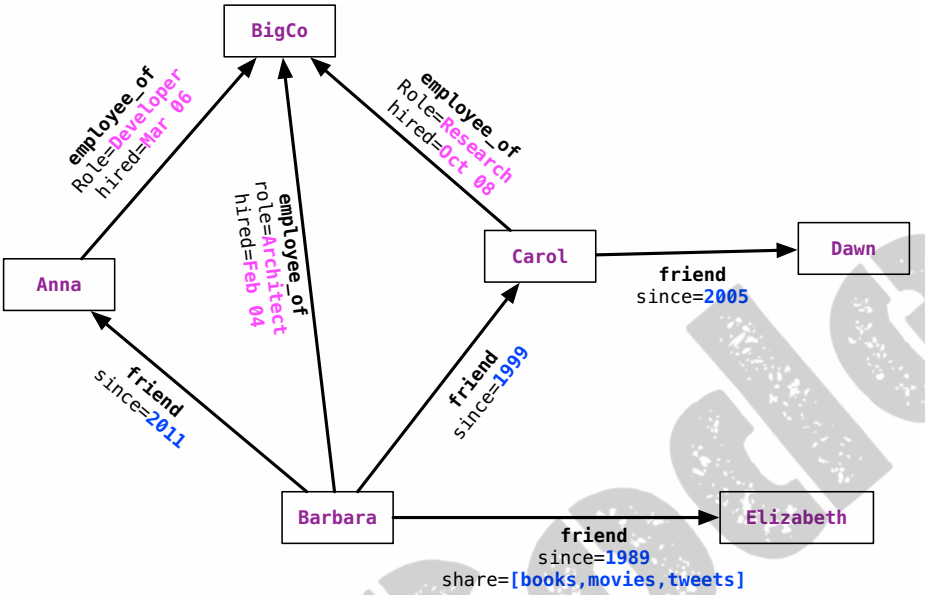

Figure 11.2 *Relationships with properties*

11.2.1 Consistency

Since graph databases are operating on connected nodes, most graph database solutions usually do not support distributing the nodes on different servers. There are some solutions, however, that support node distribution across a cluster of servers, such as Infinite Graph. Within a single server, data is always consistent, especially in Neo4J which is fully ACID-compliant. When running Neo4J in a cluster, a write to the master is eventually synchronized to the slaves, while slaves are always available for read. Writes to slaves are allowed and are immediately synchronized to the master; other slaves will not be synchronized immediately, though—they will have to wait for the data to propagate from the master.

Graph databases ensure consistency through transactions. They do not allow dangling relationships: The start node and end node always have to exist, and nodes can only be deleted if they don't have any relationships attached to them.

11.2.2 Transactions

Neo4J is ACID-compliant. Before changing any nodes or adding any relationships to existing nodes, we have to start a transaction. Without wrapping operations in transactions, we will get a `NotInTransactionException`. Read operations can be done without initiating a transaction.

```
Transaction transaction = database.beginTx();
try {
    Node node = database.createNode();
    node.setProperty("name", "NoSQL Distilled");
    node.setProperty("published", "2012");
    transaction.success();
} finally {
    transaction.finish();
}
```

In the above code, we started a transaction on the database, then created a node and set properties on it. We marked the transaction as `success` and finally completed it by `finish`. A transaction has to be marked as `success`, otherwise Neo4J assumes that it was a failure and rolls it back when `finish` is issued. Setting `success` without issuing `finish` also does not commit the data to the database. This way of managing transactions has to be remembered when developing, as it differs from the standard way of doing transactions in an RDBMS.

11.2.3 Availability

Neo4J, as of version 1.8, achieves high availability by providing for replicated slaves. These slaves can also handle writes: When they are written to, they synchronize the write to the current master, and the write is committed first at the master and then at the slave. Other slaves will eventually get the update. Other graph databases, such as Infinite Graph and FlockDB, provide for distributed storage of the nodes.

Neo4J uses the Apache ZooKeeper [ZooKeeper] to keep track of the last transaction IDs persisted on each slave node and the current master node. Once a server starts up, it communicates with ZooKeeper and finds out which server is the master. If the server is the first one to join the cluster, it becomes the master; when a master goes down, the cluster elects a master from the available nodes, thus providing high availability.

11.2.4 Query Features

Graph databases are supported by query languages such as Gremlin [Gremlin]. Gremlin is a domain-specific language for traversing graphs; it can traverse all graph databases that implement the Blueprints [Blueprints] property graph. Neo4J also has the Cypher [Cypher] query language for querying the graph. Outside these query languages, Neo4J allows you to query the graph for properties of the nodes, traverse the graph, or navigate the nodes relationships using language bindings.

Properties of a node can be indexed using the indexing service. Similarly, properties of relationships or edges can be indexed, so a node or edge can be found by the value. Indexes should be queried to find the starting node to begin a traversal. Let's look at searching for the node using node indexing.

If we have the graph shown in Figure 11.1, we can index the nodes as they are added to the database, or we can index all the nodes later by iterating over them. We first need to create an index for the nodes using the **IndexManager**.

```
Index<Node> nodeIndex = graphDb.index().forNodes("nodes");
```

We are indexing the nodes for the `name` property. Neo4J uses Lucene [Lucene] as its indexing service. We will see later that we can also use the full-text search capability of Lucene. When new nodes are created, they can be added to the index.

```
Transaction transaction = graphDb.beginTx();
try {
    Index<Node> nodeIndex = graphDb.index().forNodes("nodes");
    nodeIndex.add(martin, "name", martin.getProperty("name"));
    nodeIndex.add(pramod, "name", pramod.getProperty("name"));
    transaction.success();
} finally {
    transaction.finish();
}
```

Adding nodes to the index is done inside the context of a transaction. Once the nodes are indexed, we can search them using the indexed property. If we search for the node with the name of `Barbara`, we would query the index for the property of `name` to have a value of `Barbara`.

```
Node node = nodeIndex.get("name", "Barbara").getSingle();
```

We get the node whose `name` is `Martin`; given the node, we can get all its relationships.

```
Node martin = nodeIndex.get("name", "Martin").getSingle();
allRelationships = martin.getRelationships();
```

We can get both `INCOMING` or `OUTGOING` relationships.

```
incomingRelations = martin.getRelationships(Direction.INCOMING);
```

We can also apply directional filters on the queries when querying for a relationship. With the graph in Figure 11.1, if we want to find all people who like `NoSQL Distilled`, we can find the `NoSQL Distilled` node and then get its relationships with `Direction.INCOMING`. At this point we can also add the type of relationship to the query filter, since we are looking only for nodes that `LIKE NoSQL Distilled`.

```
Node nosqlDistilled = nodeIndex.get("name",
                            "NoSQL Distilled").getSingle();
relationships = nosqlDistilled.getRelationships(INCOMING, LIKES);
for (Relationship relationship : relationships) {
likesNoSQLDistilled.add(relationship.getStartNode());
}
```

Finding nodes and their immediate relations is easy, but this can also be achieved in RDBMS databases. Graph databases are really powerful when you want to traverse the graphs at any depth and specify a starting node for the traversal. This is especially useful when you are trying to find nodes that are related to the starting node at more than one level down. As the depth of the graph increases, it makes more sense to traverse the relationships by using a `Traverser` where you can specify that you are looking for `INCOMING`, `OUTGOING`, or `BOTH` types of relationships. You can also make the traverser go top-down or sideways on the graph by using `Order` values of `BREADTH_FIRST` or `DEPTH_FIRST`. The traversal has to start at some node—in this example, we try to find all the nodes at any depth that are related as a `FRIEND` with `Barbara`:

```
Node barbara = nodeIndex.get("name", "Barbara").getSingle();

Traverser friendsTraverser = barbara.traverse(Order.BREADTH_FIRST,
    StopEvaluator.END_OF_GRAPH,
    ReturnableEvaluator.ALL_BUT_START_NODE,
    EdgeType.FRIEND,
    Direction.OUTGOING);
```

The `friendsTraverser` provides us a way to find all the nodes that are related to `Barbara` where the relationship type is `FRIEND`. The nodes can be at any depth—friend of a friend at any level—allowing you to explore tree structures.

One of the good features of graph databases is finding paths between two nodes—determining if there are multiple paths, finding all of the paths or the shortest path. In the graph in Figure 11.1, we know that `Barbara` is connected to `Jill` by two distinct paths; to find all these paths and the distance between `Barbara` and `Jill` along those different paths, we can use

```
Node barbara = nodeIndex.get("name", "Barbara").getSingle();
Node jill = nodeIndex.get("name", "Jill").getSingle();
PathFinder<Path> finder = GraphAlgoFactory.allPaths(
        Traversal.expanderForTypes(FRIEND,Direction.OUTGOING)
                                    ,MAX_DEPTH);
Iterable<Path> paths = finder.findAllPaths(barbara, jill);
```

This feature is used in social networks to show relationships between any two nodes. To find all the paths and the distance between the nodes for each path, we first get a list of distinct paths between the two nodes. The length of each path is the **number of hops** on the graph needed to reach the destination node from the start node. Often, you need to get the shortest path between two nodes; of the two paths from `Barbara` to `Jill`, the shortest path can be found by using

```
PathFinder<Path> finder = GraphAlgoFactory.shortestPath(
        Traversal.expanderForTypes(FRIEND, Direction.OUTGOING)
                                     , MAX_DEPTH);
Iterable<Path> paths = finder.findAllPaths(barbara, jill);
```

Many other graph algorithms can be applied to the graph at hand, such as Dijkstra's algorithm [Dijkstra's] for finding the shortest or cheapest path between nodes.

```
START beginingNode = (beginning node specification)
MATCH (relationship, pattern matches)
WHERE (filtering condition: on data in nodes and relationships)
RETURN (What to return: nodes, relationships, properties)
ORDER BY (properties to order by)
SKIP (nodes to skip from top)
LIMIT (limit results)
```

Neo4J also provides the **Cypher** query language to query the graph. Cypher needs a node to `START` the query. The start node can be identified by its node ID, a list of node IDs, or index lookups. Cypher uses the `MATCH` keyword for matching patterns in relationships; the `WHERE` keyword filters the properties on a node or relationship. The `RETURN` keyword specifies what gets returned by the query—nodes, relationships, or fields on the nodes or relationships.

Cypher also provides methods to `ORDER`, `AGGREGATE`, `SKIP`, and `LIMIT` the data. In Figure 11.2, we find all nodes connected to `Barbara`, either incoming or outgoing, by using the `--`.

```
START barbara = node:nodeIndex(name = "Barbara")
MATCH (barbara)--(connected_node)
RETURN connected_node
```

When interested in directional significance, we can use

```
MATCH (barbara)<--(connected_node)
```

for incoming relationships or

```
MATCH (barbara)-->(connected_node)
```

for outgoing relationships. Match can also be done on specific relationships using the `:RELATIONSHIP_TYPE` convention and returning the required fields or nodes.

```
START barbara = node:nodeIndex(name = "Barbara")
MATCH (barbara)-[:FRIEND]->(friend_node)
RETURN friend_node.name,friend_node.location
```

We start with `Barbara`, find all outgoing relationships with the type of `FRIEND`, and return the friends' names. The relationship type query only works for the depth of one level; we can make it work for greater depths and find out the depth of each of the result nodes.

```
START barbara=node:nodeIndex(name = "Barbara")
MATCH path = barbara-[:FRIEND*1..3]->end_node
RETURN barbara.name,end_node.name, length(path)
```

Similarly, we can query for relationships where a particular relationship property exists. We can also filter on the properties of relationships and query if a property exists or not.

```
START barbara = node:nodeIndex(name = "Barbara")
MATCH (barbara)-[relation]->(related_node)
WHERE type(relation) = 'FRIEND' AND relation.share
RETURN related_node.name, relation.since
```

There are many other query features in the Cypher language that can be used to query database graphs.

11.2.5 Scaling

In NoSQL databases, one of the commonly used scaling techniques is sharding, where data is split and distributed across different servers. With graph databases, sharding is difficult, as graph databases are not aggregate-oriented but relationship-oriented. Since any given node can be related to any other node, storing related nodes on the same server is better for graph traversal. Traversing a graph when the nodes are on different machines is not good for performance. Knowing this limitation of the graph databases, we can still scale them using some common techniques described by Jim Webber [Webber Neo4J Scaling].

Generally speaking, there are three ways to scale graph databases. Since machines now can come with lots of RAM, we can add enough RAM to the server so that the working set of nodes and relationships is held entirely in memory. This technique is only helpful if the dataset that we are working with will fit in a realistic amount of RAM.

We can improve the read scaling of the database by adding more slaves with read-only access to the data, with all the writes going to the master. This pattern of writing once and reading from many servers is a proven technique in MySQL clusters and is really useful when the dataset is large enough to not fit in a single machine's RAM, but small enough to be replicated across multiple machines. Slaves can also contribute to availability and read-scaling, as they can be configured to never become a master, remaining always read-only.

When the dataset size makes replication impractical, we can shard (see the "Sharding" section on p. 38) the data from the application side using domain-specific knowledge. For example, nodes that relate to the North America can be created on one server while the nodes that relate to Asia on another. This application-level sharding needs to understand that nodes are stored on physically different databases (Figure 11.3).

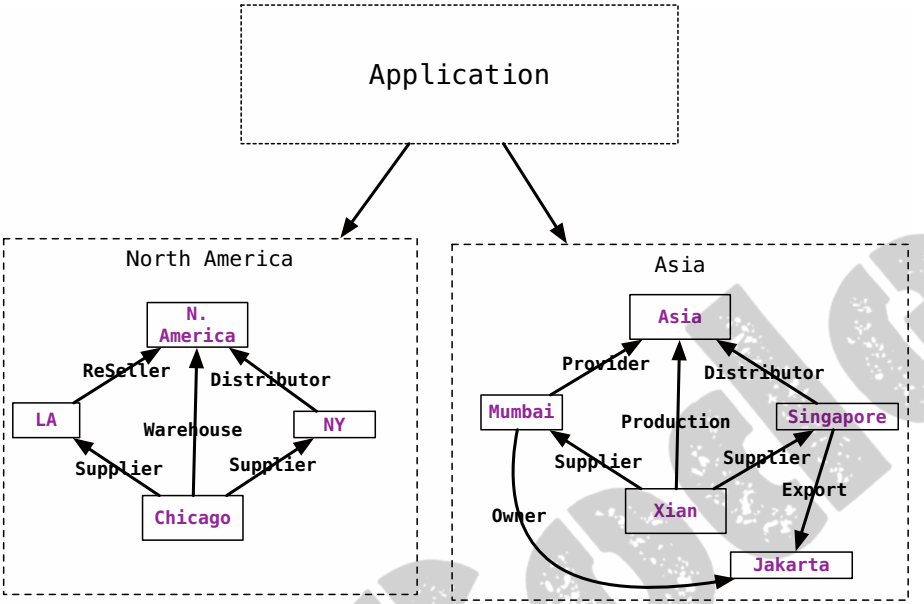

Figure 11.3 *Application-level sharding of nodes*

11.3 Suitable Use Cases

Let's look at some suitable use cases for graph databases.

11.3.1 Connected Data

Social networks are where graph databases can be deployed and used very effectively. These social graphs don't have to be only of the friend kind; for example, they can represent employees, their knowledge, and where they worked with other employees on different projects. Any link-rich domain is well suited for graph databases.

If you have relationships between domain entities from different domains (such as social, spatial, commerce) in a single database, you can make these relationships more valuable by providing the ability to traverse across domains.

11.3.2 Routing, Dispatch, and Location-Based Services

Every location or address that has a delivery is a node, and all the nodes where the delivery has to be made by the delivery person can be modeled as a graph of nodes. Relationships between nodes can have the property of distance, thus

allowing you to deliver the goods in an efficient manner. Distance and location properties can also be used in graphs of places of interest, so that your application can provide recommendations of good restaurants or entertainment options nearby. You can also create nodes for your points of sales, such as bookstores or restaurants, and notify the users when they are close to any of the nodes to provide location-based services.

11.3.3 Recommendation Engines

As nodes and relationships are created in the system, they can be used to make recommendations like "your friends also bought this product" or "when invoicing this item, these other items are usually invoiced." Or, it can be used to make recommendations to travelers mentioning that when other visitors come to Barcelona they usually visit Antonio Gaudi's creations.

An interesting side effect of using the graph databases for recommendations is that as the data size grows, the number of nodes and relationships available to make the recommendations quickly increases. The same data can also be used to mine information—for example, which products are always bought together, or which items are always invoiced together; alerts can be raised when these conditions are not met. Like other recommendation engines, graph databases can be used to search for patterns in relationships to detect fraud in transactions.

11.4 When Not to Use

In some situations, graph databases may not be appropriate. When you want to update all or a subset of entities—for example, in an analytics solution where all entities may need to be updated with a changed property—graph databases may not be optimal since changing a property on all the nodes is not a straightforward operation. Even if the data model works for the problem domain, some databases may be unable to handle lots of data, especially in global graph operations (those involving the whole graph).