CHAPTER 3

Graph Platforms and Processing

In this chapter, we'll quickly cover different methods for graph processing and the most common platform approaches. We'll look more closely at the two platforms used in this book, Apache Spark and Neo4j, and when they may be appropriate for different requirements. Platform installation guidelines are included to prepare you for the next several chapters.

Graph Platform and Processing Considerations

Graph analytical processing has unique qualities such as computation that is structure-driven, globally focused, and difficult to parse. In this section we'll look at the general considerations for graph platforms and processing.

Platform Considerations

There's debate as to whether it's better to scale up or scale out graph processing. Should you use powerful multicore, large-memory machines and focus on efficient data structures and multithreaded algorithms? Or are investments in distributed processing frameworks and related algorithms worthwhile?

A useful evaluation approach is the *Configuration that Outperforms a Single Thread* (COST), as described in the research paper "Scalability! But at What COST?" by F. McSherry, M. Isard, and D. Murray. COST provides us with a way to compare a system's scalability with the overhead the system introduces. The core concept is that a well-configured system using an optimized algorithm and data structure can outperform current general-purpose scale-out solutions. It's a method for measuring performance gains without rewarding systems that mask inefficiencies through parallelization. Separating the ideas of scalability and efficient use of resources will help us build a platform configured explicitly for our needs.

Some approaches to graph platforms include highly integrated solutions that optimize algorithms, processing, and memory retrieval to work in tighter coordination.

Processing Considerations

There are different approaches for expressing data processing; for example, stream or batch processing or the map-reduce paradigm for records-based data. However, for graph data, there also exist approaches which incorporate the data dependencies inherent in graph structures into their processing:

Node-centric
: This approach uses nodes as processing units, having them accumulate and compute state and communicate state changes via messages to their neighbors. This model uses the provided transformation functions for more straightforward implementations of each algorithm.

Relationship-centric
: This approach has similarities with the node-centric model but may perform better for subgraph and sequential analysis.

Graph-centric
: These models process nodes within a subgraph independently of other subgraphs while (minimal) communication to other subgraphs happens via messaging.

Traversal-centric
: These models use the accumulation of data by the traverser while navigating the graph as their means of computation.

Algorithm-centric
: These approaches use various methods to optimize implementations per algorithm. This is a hybrid of the previous models.

> *Pregel* is a node-centric, fault-tolerant parallel processing framework created by Google for performant analysis of large graphs. Pregel is based on the *bulk synchronous parallel* (BSP) model. BSP simplifies parallel programming by having distinct computation and communication phases.
>
> Pregel adds a node-centric abstraction atop BSP whereby algorithms compute values from incoming messages from each node's neighbors. These computations are executed once per iteration and can update node values and send messages to other nodes. The nodes can also combine messages for transmission during the communication phase, which helpfully reduces the amount of network chatter. The algorithm completes when either no new messages are sent or a set limit has been reached.

Most of these graph-specific approaches require the presence of the entire graph for efficient cross-topological operations. This is because separating and distributing the graph data leads to extensive data transfers and reshuffling between worker instances. This can be difficult for the many algorithms that need to iteratively process the global graph structure.

Representative Platforms

To address the requirements of graph processing, several platforms have emerged. Traditionally there was a separation between graph compute engines and graph databases, which required users to move their data depending on their process needs:

Graph compute engines
: These are read-only, nontransactional engines that focus on efficient execution of iterative graph analytics and queries of the whole graph. Graph compute engines support different definition and processing paradigms for graph algorithms, like node-centric (e.g., Pregel, Gather-Apply-Scatter) or MapReduce-based approaches (e.g., PACT). Examples of such engines are Giraph, GraphLab, Graph-Engine, and Apache Spark.

Graph databases
: From a transactional background, these focus on fast writes and reads using smaller queries that generally touch a small fraction of a graph. Their strengths are in operational robustness and high concurrent scalability for many users.

Selecting Our Platform

Choosing a production platform involves many considersations, such as the type of analysis to be run, performance needs, the existing environment, and team preferences. We use Apache Spark and Neo4j to showcase graph algorithms in this book because they both offer unique advantages.

Spark is an example of a scale-out and node-centric graph compute engine. Its popular computing framework and libraries support a variety of data science workflows. Spark may be the right platform when our:

- Algorithms are fundamentally parallelizable or partitionable.
- Algorithm workflows need "multilingual" operations in multiple tools and languages.
- Analysis can be run offline in batch mode.
- Graph analysis is on data not transformed into a graph format.
- Team needs and has the expertise to code and implement their own algorithms.
- Team uses graph algorithms infrequently.

- Team prefers to keep all data and analysis within the Hadoop ecosystem.

The Neo4j Graph Platform is an example of a tightly integrated graph database and algorithm-centric processing, optimized for graphs. It is popular for building graph-based applications and includes a graph algorithms library tuned for its native graph database. Neo4j may be the right platform when our:

- Algorithms are more iterative and require good memory locality.
- Algorithms and results are performance sensitive.
- Graph analysis is on complex graph data and/or requires deep path traversal.
- Analysis/results are integrated with transactional workloads.
- Results are used to enrich an existing graph.
- Team needs to integrate with graph-based visualization tools.
- Team prefers prepackaged and supported algorithms.

Finally, some organizations use both Neo4j and Spark for graph processing: Spark for the high-level filtering and preprocessing of massive datasets and data integration, and Neo4j for more specific processing and integration with graph-based applications.

Apache Spark

Apache Spark (henceforth just Spark) is an analytics engine for large-scale data processing. It uses a table abstraction called a DataFrame to represent and process data in rows of named and typed columns. The platform integrates diverse data sources and supports languages such as Scala, Python, and R. Spark supports various analytics libraries, as shown in Figure 3-1. Its memory-based system operates by using efficiently distributed compute graphs.

GraphFrames is a graph processing library for Spark that succeeded GraphX in 2016, although it is separate from the core Apache Spark. GraphFrames is based on GraphX, but uses DataFrames as its underlying data structure. GraphFrames has support for the Java, Scala, and Python programming languages. In spring 2019, the "Spark Graph: Property Graphs, Cypher Queries, and Algorithms" proposal was accepted (see "Spark Graph Evolution" on page 33). We expect this to bring a number of graph features using the DataFrame framework and Cypher query language into the core Spark project. However, in this book our examples will be based on the Python API (PySpark) because of its current popularity with Spark data scientists.

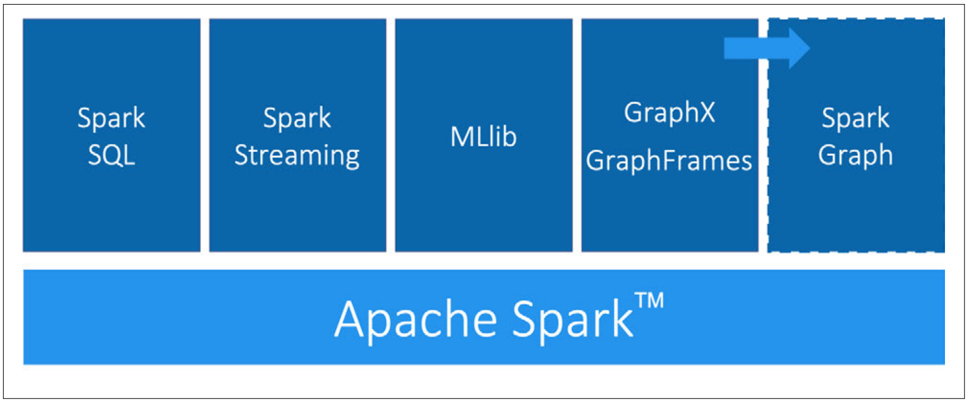


Figure 3-1. Spark is an open-source distributed and general-purpose clustercomputing framework. It includes several modules for various workloads.

Spark Graph Evolution

The Spark Graph project is a joint initiative from Apache project contributors in Databricks and Neo4j to bring support for DataFrames, Cypher, and DataFrames-based algorithms into the core Apache Spark project as part of the 3.0 release.

Cypher started as a declarative graph query language implemented in Neo4j, but through the openCypher project it's now used by multiple database vendors and an opensource project, Cypher for Apache Spark (CAPS).

In the very near future, we look forward to using CAPS to load and project graph data as an integrated part of the Spark platform. We'll publish Cypher examples after the Spark Graph project is implemented.

This development does not impact the algorithms covered in this book but may add new options to how procedures are called. The underlying data model, concepts, and computation of graph algorithms will remain the same.

Nodes and relationships are represented as DataFrames with a unique ID for each node and a source and destination node for each relationship. We can see an example of a nodes DataFrame in Table 3-1 and a relationships DataFrame in Table 3-2. A GraphFrame based on these DataFrames would have two nodes, `JFK` and `SEA`, and one relationship, from `JFK` to `SEA`.

Table 3-1. Nodes DataFrame

id	city	state
JFK	New York	NY
SEA	Seattle	WA

Table 3-2. Relationships DataFrame

src	dst	delay	tripId
JFK	SEA	45	1058923

The nodes DataFrame must have an `id` column—the value in this column is used to uniquely identify each node. The relationships DataFrame must have `src` and `dst` columns—the values in these columns describe which nodes are connected and should refer to entries that appear in the `id` column of the nodes DataFrame.

The nodes and relationships DataFrames can be loaded using any of the DataFrame data sources, including Parquet, JSON, and CSV. Queries are described using a combination of the PySpark API and Spark SQL.

GraphFrames also provides users with an extension point to implement algorithms that aren't available out of the box.

Installing Spark

You can download Spark from the Apache Spark website. Once you've downloaded it, you need to install the following libraries to execute Spark jobs from Python:

```
pip install pyspark graphframes
```

You can then launch the *pyspark* REPL by executing the following command:

```
export SPARK_VERSION="spark-2.4.0-bin-hadoop2.7"
./${SPARK_VERSION}/bin/pyspark \
  --driver-memory 2g \
  --executor-memory 6g \
  --packages graphframes:graphframes:0.7.0-spark2.4-s_2.11
```

At the time of writing the latest released version of Spark is *spark-2.4.0-bin-hadoop2.7*, but that may have changed by the time you read this. If so, be sure to change the `SPARK_VERSION` environment variable appropriately.

Although Spark jobs should be executed on a cluster of machines, for demonstration purposes we're only going to execute the jobs on a single machine. You can learn more about running Spark in production environments in *Spark: The Definitive Guide*, by Bill Chambers and Matei Zaharia (O'Reilly).

You're now ready to learn how to run graph algorithms on Spark.

Neo4j Graph Platform

The Neo4j Graph Platform supports transactional processing and analytical processing of graph data. It includes graph storage and compute with data management and analytics tooling. The set of integrated tools sits on top of a common protocol, API, and query language (Cypher) to provide effective access for different uses, as shown in Figure 3-2.

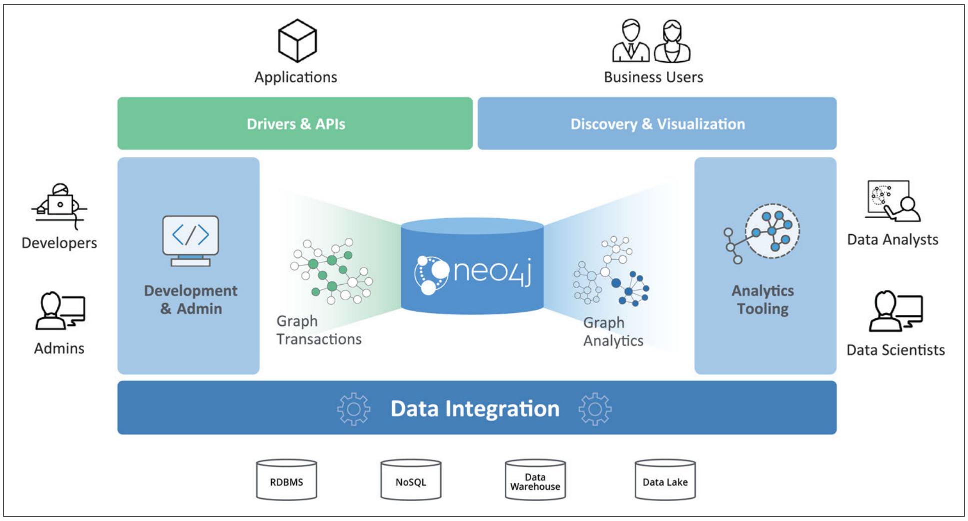


Figure 3-2. The Neo4j Graph Platform is built around a native graph database that supports transactional applications and graph analytics.

In this book, we'll be using the Neo4j Graph Algorithms library. The library is installed as a plug-in alongside the database and provides a set of user-defined procedures that can be executed via the Cypher query language.

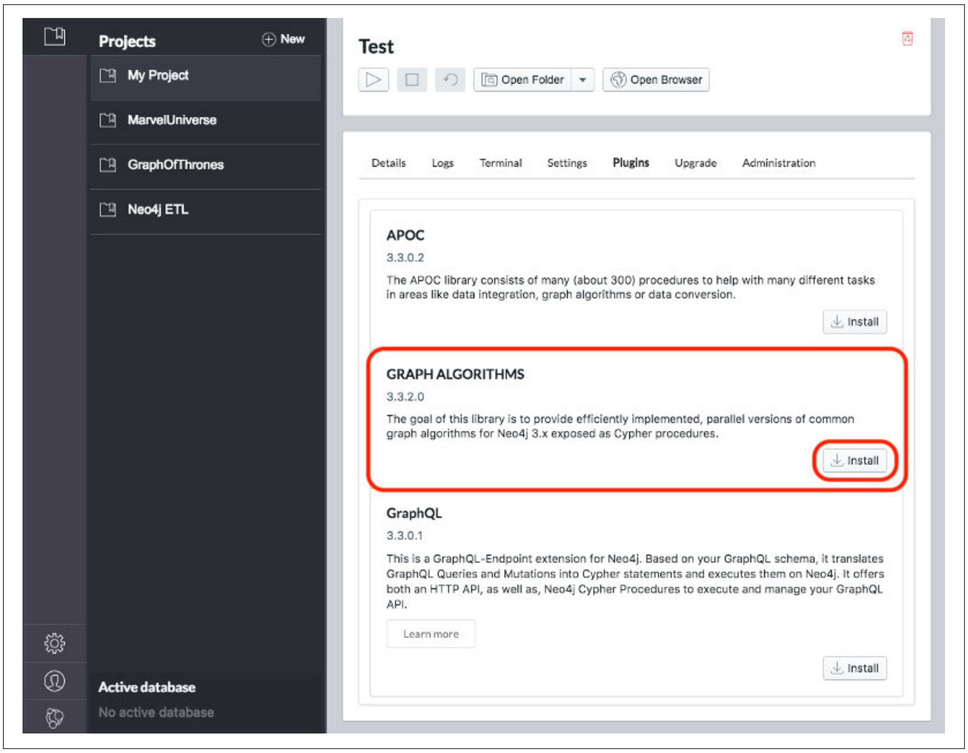
The graph algorithm library includes parallel versions of algorithms supporting graph analytics and machine learning workflows. The algorithms are executed on top of a task -based parallel computation framework and are optimized for the Neo4j platform. For different graph sizes there are internal implementations that scale up to tens of billions of nodes and relationships.

Results can be streamed to the client as a tuples stream and tabular results can be used as a driving table for further processing. Results can also be optionally written back to the database efficiently as node properties or relationship types.

In this book, we'll also be using the Neo4j Awesome Procedures on Cypher (APOC) library. APOC consists of more than 450 procedures and functions to help with common tasks such as data integration, data conversion, and model refactoring.

Installing Neo4j

Neo4j Desktop is a convenient way for developers to work with local Neo4j databases. It can be downloaded from the Neo4j website. The graph algorithms and APOC libraries can be installed as plug-ins once you've installed and launched the Neo4j Desktop. In the lefthand menu, create a project and select it. Then click Manage on the database where you want to install the plug-ins. On the Plugins tab, you'll see options for several plug-ins. Click the Install button for graph algorithms and APOC. See Figures 3-3 and 3-4.



APOC

3.3.0.2

The APOC library consists of many (about 300) procedures to help with many different tasks in areas like data integration, graph algorithms or data conversion.

GRAPH ALGORITHMS

3.3.2.0

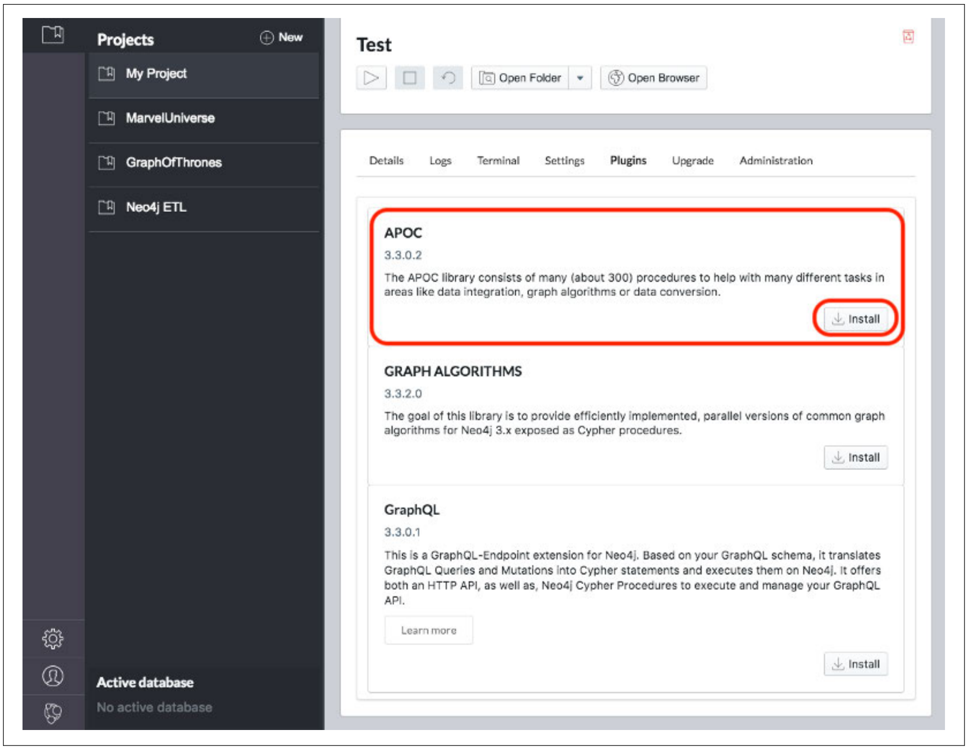
The goal of this library is to provide efficiently implemented, parallel versions of common graph algorithms for Neo4j 3.x exposed as Cypher procedures.

GraphQL

3.3.0.1

This is a GraphQL-Endpoint extension for Neo4j. Based on your GraphQL schema, it translates GraphQL Queries and Mutations into Cypher statements and executes them on Neo4j. It offers both an HTTP API, as well as, Neo4j Cypher Procedures to execute and manage your GraphQL API.

Figure 3-3. Installing the graph algorithms library



APOC
3.3.0.2
The APOC library consists of many (about 300) procedures to help with many different tasks in areas like data integration, graph algorithms or data conversion.

GRAPH ALGORITHMS
3.3.2.0
The goal of this library is to provide efficiently implemented, parallel versions of common graph algorithms for Neo4j 3.x exposed as Cypher procedures.

GraphQL
3.3.0.1
This is a GraphQL-Endpoint extension for Neo4j. Based on your GraphQL schema, it translates GraphQL Queries and Mutations into Cypher statements and executes them on Neo4j. It offers both an HTTP API, as well as, Neo4j Cypher Procedures to execute and manage your GraphQL API.

Figure 3-4. Installing the APOC library

Jennifer Reif explains the installation process in more detail in her blog post "Explore New Worlds—Adding Plugins to Neo4j". You're now ready to learn how to run graph algorithms in Neo4j.

Summary

In the previous chapters we've described why graph analytics is important to studying real-world networks and looked at fundamental graph concepts, analysis, and processing. This puts us on solid footing for understanding how to apply graph algorithms. In the next chapters, we'll discover how to run graph algorithms with examples in Spark and Neo4j.