appendix B Supported search engines and vector databases

We use the open source Apache Solr search engine as our default search engine throughout the book for consistency, but all of the algorithms in the codebase are designed to work with a wide variety of search engines and vector databases. To that end, other than cases where engine-specific syntax is required to demonstrate a point, we have implemented search functionality using a generic `engine` interface that allows you to easily swap in your preferred search engine or vector database. In this appendix, we'll cover the list of supported engines, how to swap out the default engine, and how the major abstractions (`engine` and `collection`) are used throughout the book.

B.1 Supported engines

The list of supported engines will continue to grow over time, but at the time of publication, the following engines are initially supported:

- `solr`—Apache Solr
- `opensearch`—OpenSearch
- `bonsai`—Bonsai
- `weaviate`—Weaviate

To see the full, most up-to-date list of all supported engines, please visit **https://aipoweredsearch.com/supported-engines**.

B.2 Swapping out the engine

Normally, you'll only be working with one search engine or vector database at a time when running the book's code examples. To use any particular engine, you just need to specify the engine's name (as enumerated above) when starting up Docker.

You would launch OpenSearch, for example, like this:

```
docker compose up opensearch
```

This will both start any necessary Docker containers needed to run `opensearch` (or the engine you specify) as well as set this engine as active in the book's Jupyter notebooks for use in all the code examples.

Note that some engines, such as managed search and API-based services, do not require any additional local Docker containers, since their services are hosted elsewhere. Also, some engines may require additional configuration parameters, such as API keys, remote addresses/URLs, ports, and so on. These parameters can be set in the .env file at the root of the project.

If you want to use a different engine at any time, you can restart the Docker containers and specify the new engine you want to use:

```
docker compose up bonsai
```

If you want to launch more than one engine at a time to experiment, you can provide a list of the engines you wish to start at the end of the `docker compose up` command:

```
docker compose up solr opensearch weaviate
```

The first engine you reference in your `docker compose up` command will be set as your active engine in the Jupyter notebooks, with the others available on standby.

If you'd like to switch to one of the standby engines within your live Jupyter notebooks (for example, to `opensearch`), you can do this at any time by simply running the following command in any notebook:

```
import aips
aips.set_engine("opensearch")
```

Keep in mind that if you call `set_engine` for an engine that is not currently running, this will later result in errors if that engine is still unavailable when you try to use it.

You can also check the currently set engine at any time by running this command:

```
aips.get_engine().name
```

B.3 The engine and collection abstractions

The search engine industry is full of different terminology and concepts, and we have tried to abstract away as much of that as possible in the codebase. Most search engines started with lexical keyword search and have since added support for vector search, and many vector databases started with vector search and have since added lexical search. For our purposes, we just think of all these systems as *matching and ranking engines*, and we use the term *engine* to refer to all of them.

Likewise, each engine has a concept of one or more logical partitions or containers for adding data. In Solr and Weaviate, these

containers are called *collections*; in OpenSearch, Elasticsearch, and Redis these are called *indexes*; and in Vespa they are called *applications*. In MongoDB, the original data is stored in a *collection*, but it can then be copied into an *index* for search purposes. Naming also varies further among other engines.
For proper abstraction, we always use the term *collection* in the codebase, so every engine that gets implemented has a `collection` interface through which you can query or add documents.

Common public methods on the `engine` interface include

- `engine.create_collection(collection_name)` —Creates a new collection
- `engine.get_collection(collection_name)` —Returns an existing collection

Common public methods on the `collection` interface include

- `collection.search(**request)` —Runs a search and returns results. Individual request parameters should be passed in as Python keyword arguments, like `collection.search(query="keyword), limit=10"`.
- `collection.add_documents(docs)` —Adds a list of documents to the collection
- `collection.write(dataframe)` —Writes each row from a Spark dataframe to the collection as a document
- `collection.commit()` —Ensures that recently added documents are persisted and available for searching

The `engine` and `collection` interfaces also internally implement schema definitions and management for all the datasets used in the book.

Because the `collection.write` method takes a dataframe, we utilize helpers as needed when loading data from additional data

sources, such as CSV or SQL:

- `collection.write(from_csv(csv_file))` —Writes each row in the CSV file to the collection as a document
- `collection.write(from_sql(sql_query))` —Runs the SQL query and writes each returned row to the collection as a document

No additional engine-specific implementation is needed for loading from these additional data sources, as any data source that can be mapped to a Spark dataframe is implicitly supported.

B.4 Adding support for additional engines

While we hope to eventually support most major search engines and vector databases, you may find that your favorite engine is not currently supported. If that is the case, we encourage you to add support for it and submit a pull request to the codebase. The `engine` and `collection` interfaces are designed to be easy to implement, and you can use the default `solr` implementation or any other already-implemented engines as a reference.

Not all data stores have full support for all the capabilities implemented in *AI-Powered Search*. For example, a pure vector database may not support lexical keyword matching and ranking, and some search engines may not support vector search. Likewise, some specialized capabilities may only be available in certain engines.

While the default `solr` engine supports all of the *AI-Powered Search* capabilities implemented in the book, other engines may require workarounds, integration of additional libraries, or delegation of some capabilities to another engine for certain algorithms. Most engines don't have native support for semantic knowledge graphs and text tagging, for example, so many of the

engine implementations will delegate these one-off capabilities to other libraries.

We hope that the `engine` and `collection` abstractions will make it easy for you to add support for your favorite engine and also to potentially contribute it to the book's codebase to benefit the larger community of *AI-Powered Search* readers and practitioners. Happy searching!

Previous chapter
‹ appendix A Running the code examples

Next chapter
index ›