Review and Preparation

The following is a review of the elements from the first and second courses of this specialization that are essential for you to be prepared for this third course. If you already took the first and second course, that’s great, and this will just be a quick recap for you. If not, then this reading should help you understand what they’re about so you can decide whether to take one or both of them before continuing with this third course.

Please do not consider this a substitute for the first two courses. There is a lot more background in Course 1, and a lot more detail in Course 2, that will add much to your understanding of these topics. If you find yourself getting lost in these recaps, please take the time to go through the other courses.

The Virtual Machine Exercise Environment

This course uses the same virtual machine (VM) used by Courses 1 and 2 for practicing the material presented, and sometimes for completing quiz questions. If you have not already used this VM for a previous course, look at the technical requirements first, then (if your machine meets those requirements) follow the instructions for downloading and installing it.

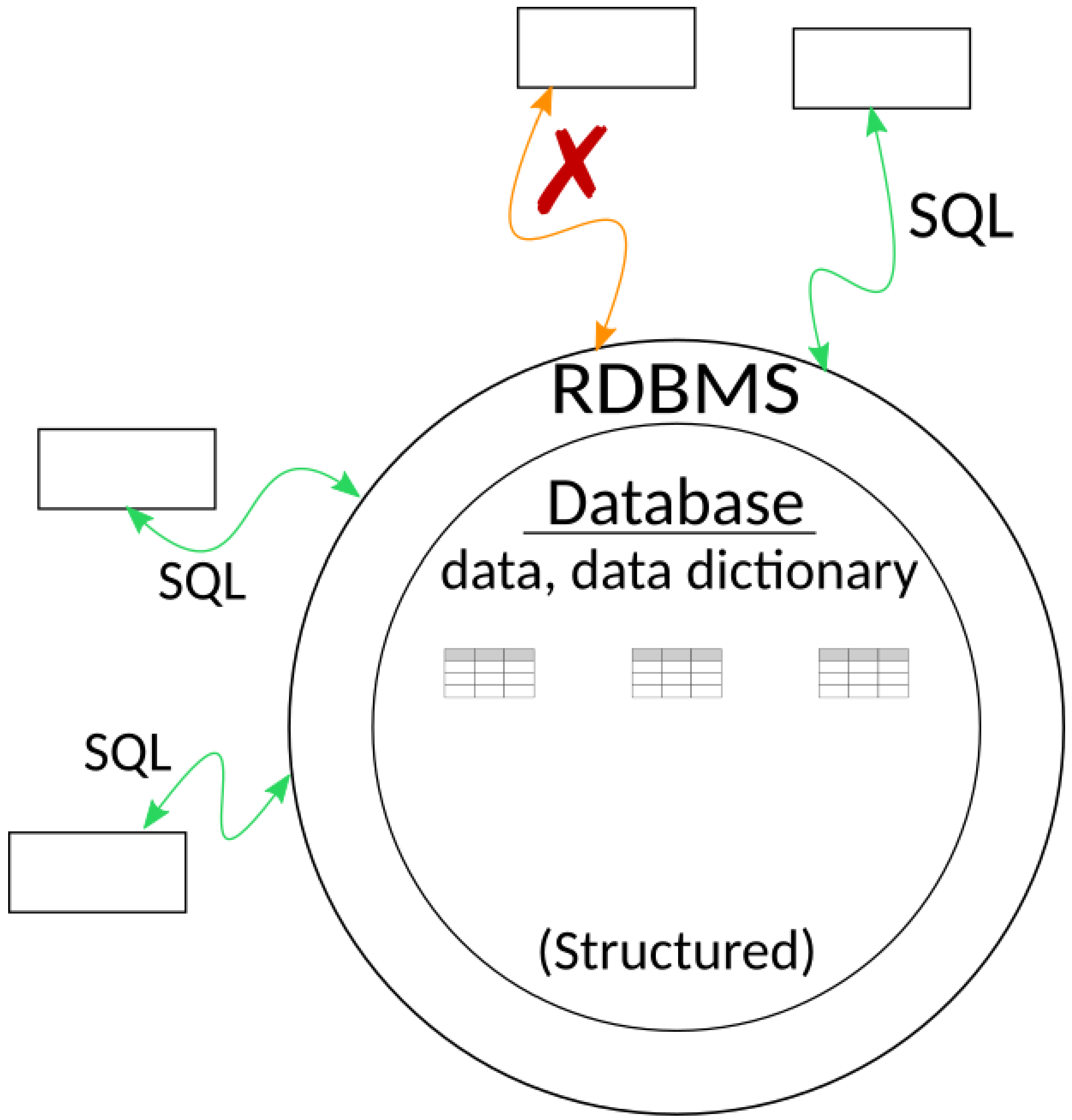
Course 1 Review

Moving into Course 3, it's important to understand how big data systems store and process data, including in the cloud, and how the tools access the data. These points were addressed in Course 1, along with more in-depth background information.

Storing and Accessing Data, Comparison

An RDBMS system keeps your table definitions (that is, the schema) in a *data dictionary*, which is *tightly coupled* with your tables: it's always kept in exact alignment, accurately describing the tables you create. This tight coupling also means that the schema governs what is allowed to be stored as data. These systems are called *schema on write* because the schema is applied before the data is stored. Databases manage all insertions and updates, and they typically throw an error if you try to do something like insert a character string value into a numeric column. If the data doesn't fit the schema, it can't be added to the table.

In a conventional RDBMS, data storage is *encapsulated* by your database software. Other programs cannot access the data storage directly: file access is usually blocked to any program other than the database software, and even with access, files are usually of a proprietary format that is not useable except through the database software. Figure 1 shows a SQL RDBMS with four attempts to access the data; the three using SQL are successful (green arrows) while the fourth, using another method (orange arrow with red X), is not.



SQL
RDBMS
Database
data, data dictionary
SQL
SQL
(Structured)

Figure 1

While the traditional RDBMS supports only structured data with access only through SQL, a big data system supports a variety of data, and also a variety of ways to access and use the data. The data store in your big data system can be called a *data lake*, or *data reservoir*, or *enterprise data hub.* The data lake can retain large varieties of data, of all sorts. Some contents might be structured and so easily usable by a SQL engine like Apache Hive or Apache Impala, and some might not be. See Figure 2.

Some programs read and write content directly to the data lake, using direct file access. These can be simple programs, written in languages like Python or Java or C, or large-scale distributed applications, like MapReduce or Spark programs. These programs can access files of potentially any format and type, and are suitable for working with structured or unstructured data.

In order to use SQL on your data, you create table definitions in a *metastore*, which—for Hive and Impala—happens to be called the *Hive metastore* because of its origin as a part of Hive. The metastore takes the place of the data dictionary in an RDBMS: it contains table definitions that enable table-like access to some of the contents in the data lake. The metastore is not kept directly in the data lake, but alongside it.

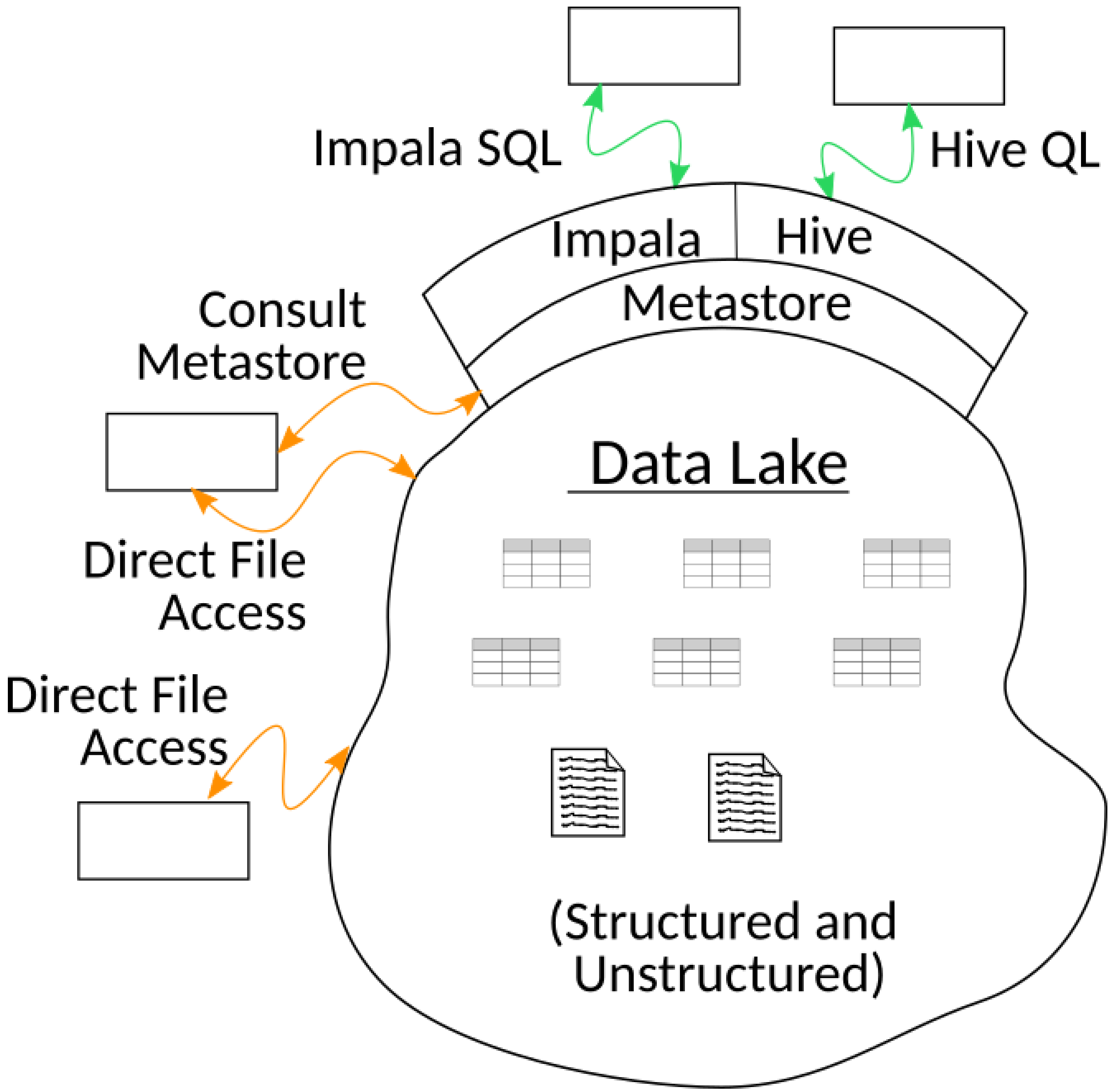

Impala SQL
Hive QL
Impala
Hive
Metastore
Consult Metastore
Direct File Access
Direct File Access
Data Lake
(Structured and Unstructured)

Figure 2

Because of the way big data systems separate your data and metadata, you can create table definitions that are *loosely coupled* to files. When your data and metadata are loosely coupled, your table definitions are not necessarily in lock-step with all your data. In fact, some files may reside in the data lake without any information about them in the metastore.

The table definitions also do not *govern* the file contents, but instead provide *schema on read* to let you view the files in table form. That is, the tool you use to access the data validates it when it's accessed (read), not when it's added (written) to the data lake. This lets your data lake accept files of any sort. The advantage is that loading data can be very fast, because it's nothing more than copying files. The disadvantage is that any errors in the data files will not be discovered until a query is performed. Missing or invalid data is generally represented as NULL values in query results, with two notable exceptions: missing STRING fields may be represented as empty strings instead of NULLs; and in Impala, out-of-range numeric values are returned as the minimum or maximum allowed value for the data type.

Impala and Hive share the one metastore to find table and column definitions, and then access files in the data lake on your behalf when you issue SQL statements. Other applications, like Spark programs, can optionally consult the metastore to find out about table definitions for data, but this is not required in order to access the data. A single file may be used by an Impala query, a Hive query, a general-purpose Spark program, or any number of other programs. This is especially true when the file has structured or semi-structured contents.

Distributed Storage and Processing

At the time of this writing, a single disk drive usually stores around a terabyte of data, or a handful of terabytes. Now consider if you want to store 30 terabytes, or 30 petabytes, or more! In modern technology there is no choice but to store your data across multiple disk drives, and the largest data stores must necessarily span thousands of disk drives. So, a big data store relies on *distributed storage*, splitting data into pieces and scattering the pieces across many disks, instead of storing a single large file. Figure 3 shows a file split into pieces, sometimes called *blocks*, with those blocks distributed across multiple disks for storage of the file. A typical size for a single block is 128 megabytes.

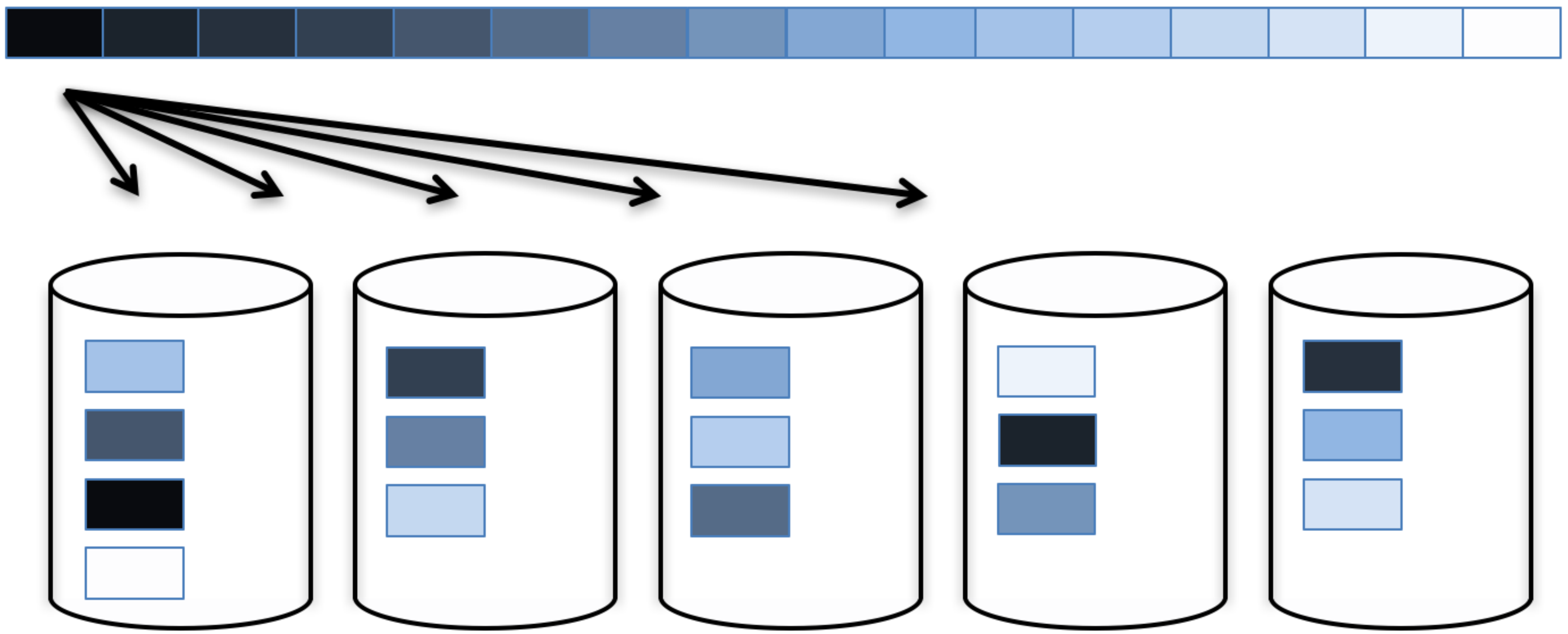


Figure 3

With this idea, if any one disk fails, then a part of your file is lost. With many disks, failure must be an expected, common occurrence. In order to keep files available, a distributed system keeps redundant copies of blocks on different disks. The system notices the failure of a disk, and it automatically makes additional copies of the lost blocks using the existing replicas.

An installation of a cluster of machines in a corporate data center is usually made up of banks of standard computers, each with its own memory, processors, and disk storage, combined in a single computing cluster. Such a *Hadoop cluster* pairs computing power—memory and CPU—together with the disk storage.

Figure 4 illustrates parallel reads and concurrent processing for a program that finds all the records and the total net amounts for the purchases and payments of a few customers.

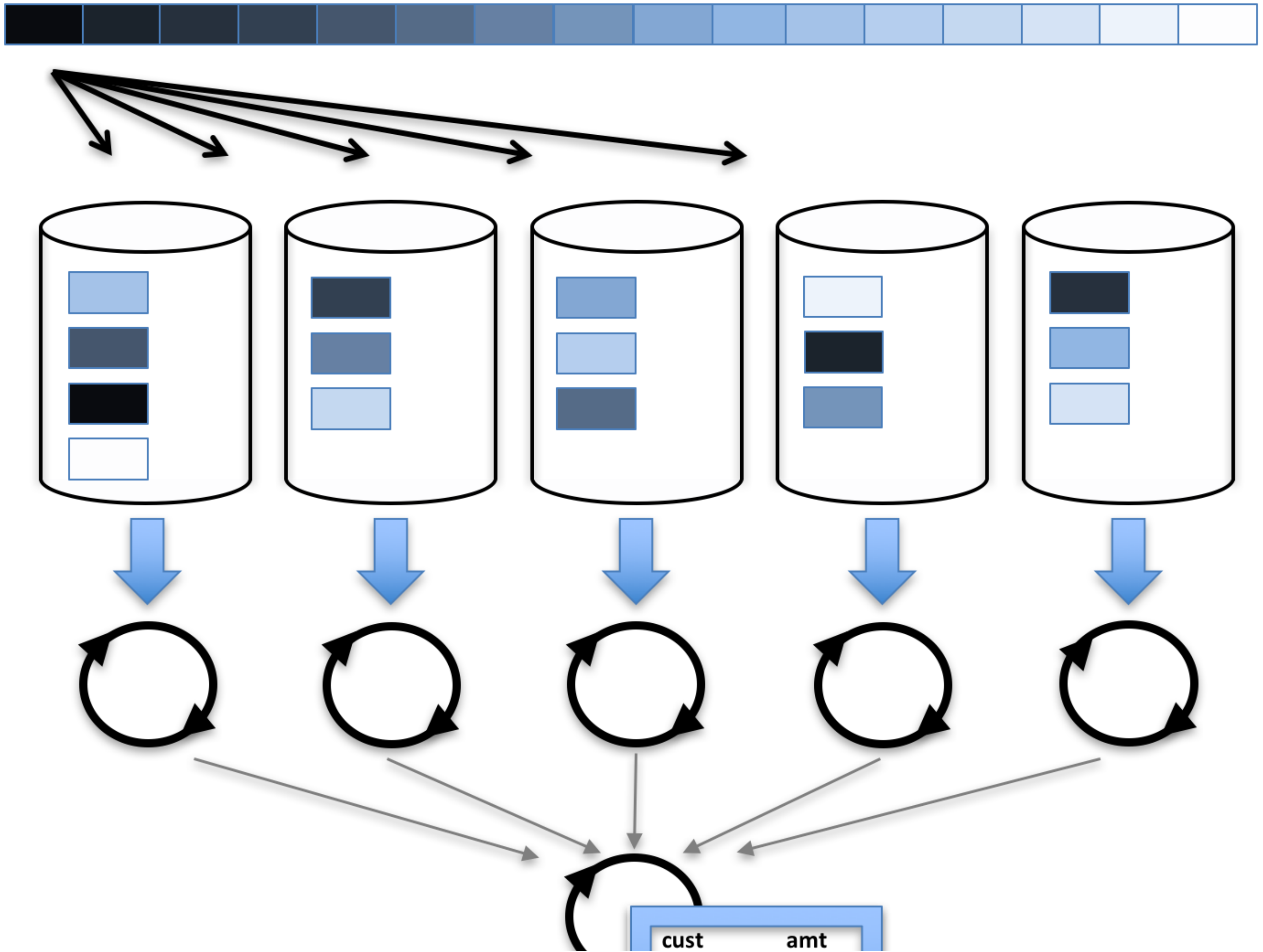

cust
amt

Figure 4

If your program calculation is sufficiently complicated, then there may be additional stages of distributed processing, in which intermediate results are reorganized by grouping or sorting, then processed further, and then there is final collection and display. See Figure 5. This intermediate reorganization of interim results can be called a *shuffle* of the data. In principle, a program may require any number of processing stages, with a shuffle between each stage.

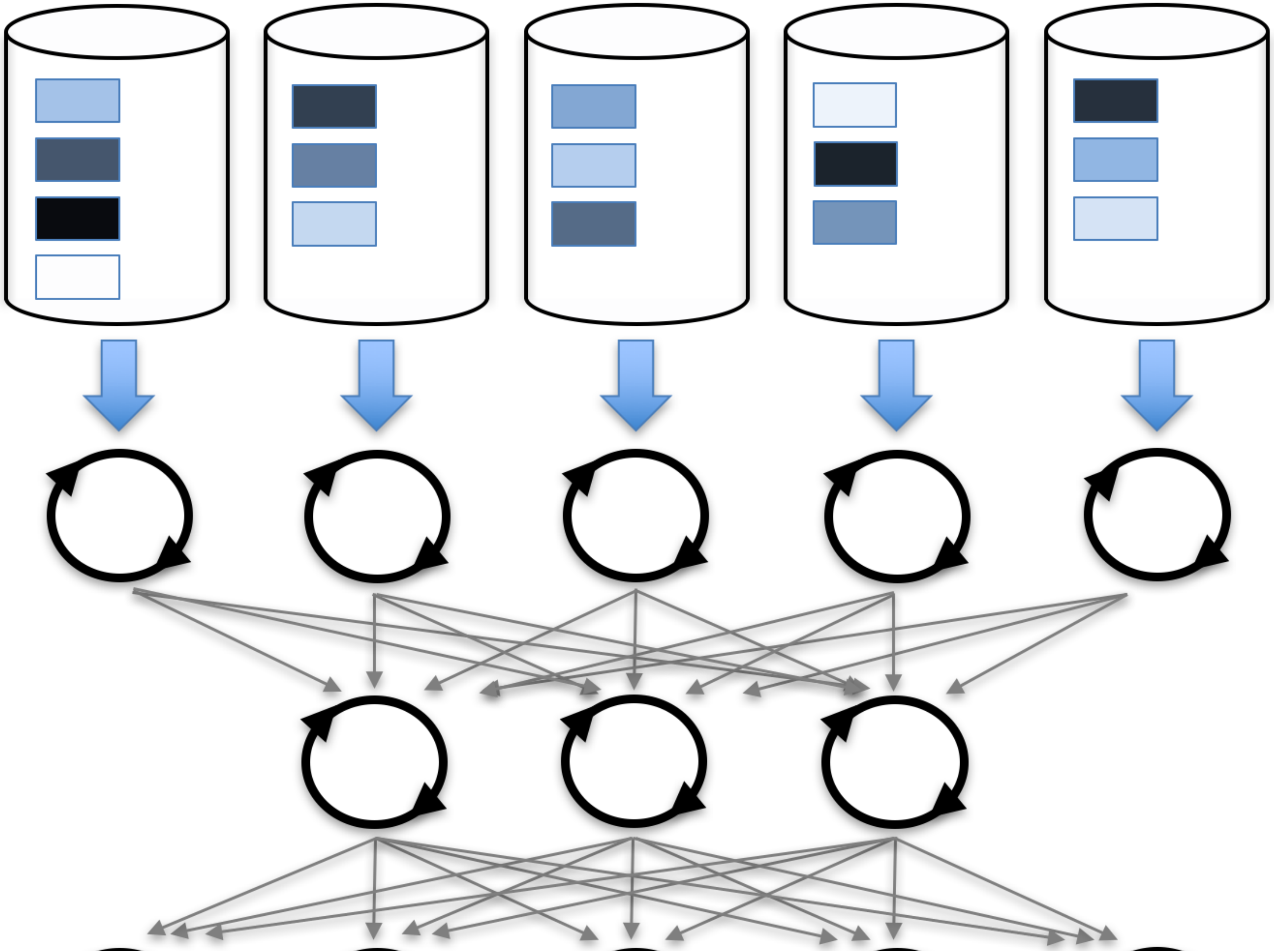


Figure 5

In the general case, your big data program may not produce a small amount of data to display, but may instead read a large data set, and produce another new large dataset. In this case, your program will perform not just distributed reads, but also distributed writes. See Figure 6.

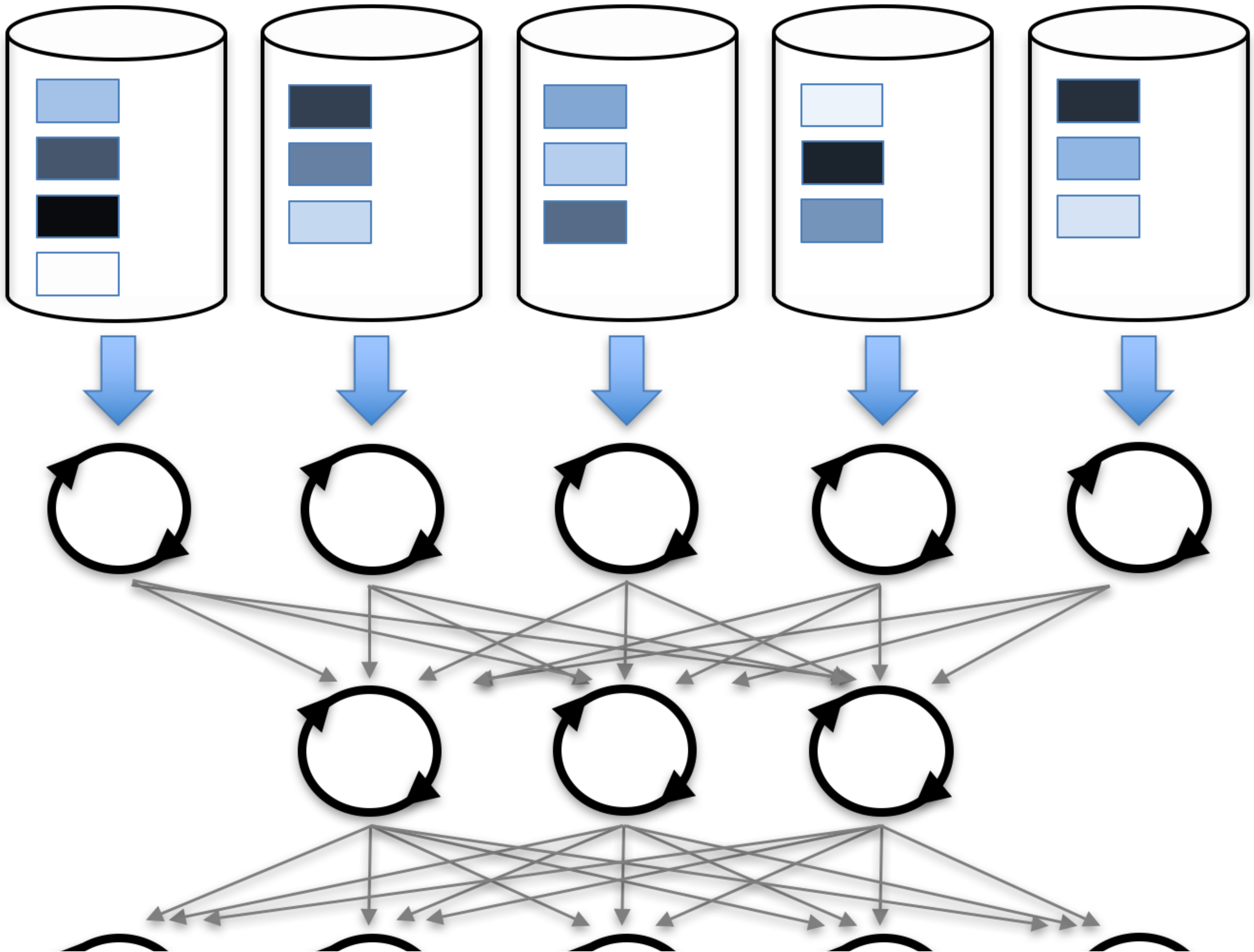


Figure 6

Storage Options (Cloud, On Premises, or Hybrid)

The internet and modern computing have produced explosive growth in data volumes, but also new ways to store data. When storing petabytes of data for analysis, you have some options. Keep in mind, it's not just that you want to *store* your data—you also want to *analyze* it to gain new insights. So you really have related decisions to make, about storage and processing for your big data.

Some organizations physically store their data on servers in their own data centers. (A *server* is simply a computer that provides services to other computers through a network.) Storing data on servers in your own data center is called on-premises or *on-prem* storage. For a typical setup, each server in the cluster contributes both data storage (with a set of disk drives) and processing capacity (with CPUs and random access memory). Data storage on the disk drives in an on-prem *Hadoop* cluster is typically managed by the file system software called *Hadoop Distributed File System* (HDFS). With an on-prem cluster like this, you can increase storage and computing capacity together by adding more servers—also called *nodes* or *hosts*—to the cluster.

You can store your data just as well using cloud services, such as Amazon Web Services (AWS), Microsoft Azure, or Google Cloud Platform (GCP). These services let companies choose to keep storage and computing power together, or keep data in cloud storage, and use (and pay for) computing power only when needed.

Or, you can take a hybrid approach, maintaining some data and computing power on premises, and some in the cloud.

Course 2 Review

Course 2 presents an in-depth look at the SELECT statement, the cornerstone of SQL-based data analysis. It introduces Hue as a tool for exploring databases and tables, and for running queries. Some utility statements for SQL are also presented.

Databases and Tables

In Course 2, the word *database* refers to a logical container for a group of tables. Tables are organized into databases. Within one database, the tables all need to have different names, but two different tables can have the same name if they are in different databases.

However, this word *database* also has broader meanings. Any organized collection of data could be called a *database*. SQL engines in general are often called *databases*. And a specific *instance* of a SQL engine is often called a *database*. In Course 1, the word *database* sometimes refers to these broader meanings. But in Course 2 and Course 3, the word *database* is primarily used to mean a logical container for a group of tables.

Hue

Hue is a web browser-based analytics workbench. Among other things, you can use it to view tables, along with their schema and sample data, or to run queries using different SQL engines including Apache Hive and Impala. In the virtual machine (VM) supplied for these courses, Hue is assumed to be the main access point.

In Figure 7, panel 1 allows you to explore the databases and the tables within them. Panel 2 is the query editor, where you can enter SELECT statements or other statements you will learn about in this course. Panel 3 is an assistant panel that presents easy access to information about tables used in your queries.

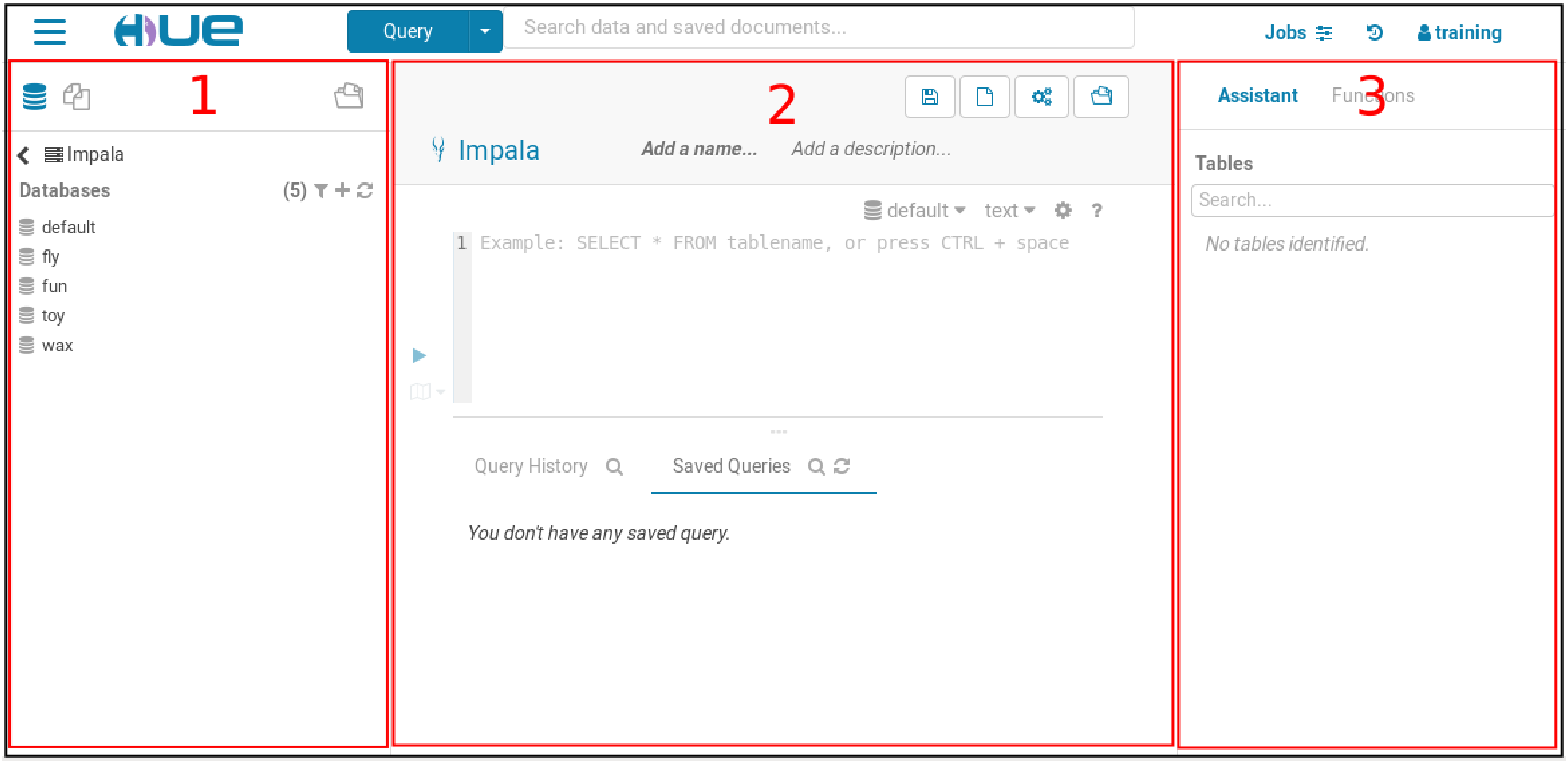


Figure 7

The "hamburger menu" (three horizontal lines) in the top left corner of Figure 7 provides access to additional areas of Hue.

Utility Statements

Some SQL interfaces do not have the point-and-click capabilities of Hue, but you can use the following statements to do some of the exploration of databases and tables that Hue provides:

- SHOW DATABASES;
- USE *database_name*;
- SHOW TABLES;
- DESCRIBE *table_name*;

SELECT Statements and Clauses

The SELECT keyword is the basis of every *query* performed in SQL. Following are the most common parts of a SELECT statement, all of which are covered in Course 2. If you are unfamiliar with any of these, please do complete Course 2 before moving on.

The SELECT list

This specifies the columns that will be included in the result set. These might be columns from the source table or expressions that might or might not incorporate columns from the source table.

FROM

The FROM clause is used to specify which table or tables will supply the data and columns used in the query. Multiple tables can be specified using UNIONs or JOINs (including implicit join syntax without explicitly using the JOIN keyword).

WHERE

The WHERE clause filters the result rows, returning only rows for which a conditional (Boolean) expression evaluates to true. Rows for which the expression is false or NULL will not be returned.

GROUP BY

Usually combined with an *aggregate function* in the SELECT list, the GROUP BY clause specifies a column or columns to use when combining multiple rows from the source table or tables into a single row in the result set.

HAVING

This is another filtering clause that allows filtering *after* grouping using the GROUP BY clause. (The WHERE clause is applied before grouping and aggregation is calculated.)

ORDER BY

The ORDER BY clause specifies how to sort the result set, if desired. Sorting can be presented in ascending order (using ASC or the default behavior) or descending order (using DESC).

LIMIT

The LIMIT clause specifies how many rows are returned.