

WEEK 1

Learning Objectives

- Explain the benefits of organizing data for later use
- Relate categories of activities for database systems to the success of relational databases and the use of SQL with them
- Differentiate between operational and analytic databases

1.1 WHAT IS DATA?

Data analysts work with guess what? Data. Starting from the beginning: What Is Data? In this specialization, data means digital data. Information that can be transmitted, stored, and processed using modern digital technologies, like the Internet, disk drives, and modern computer.

Imagine that you walk by a movie theater and see a movie poster about a new movie you want to see. You enjoy the image and you read to find out about the actors in the movie. And the showtimes at the local theater. So you get information that is interesting and useful to you. But does the site of the movie poster itself comprise data?

No, the poster presents an image to your eye. Which you read and interpret to get the actors' name and the showtimes. So an actual object or a direct sensory experience is not really data. But you can have data about those things. **Data is a representation of something that captures some features and ignores others.** Now data itself can be divided into two kinds, analog data and digital data. A photograph of the movie poster taken with a film camera is not digital data. An image from photographic film results from chemical processes on photosensitive materials with essentially infinite continuous ranges of color on paper.

In contrast, a digital scan of the movie poster or a photo taken with a smartphone is digital data. It can be saved, copied, sent to another smartphone, displayed on many screens, so on. A film photograph requires mechanical and chemical processes to reproduce. But a smartphone photograph is just numbers stored in the phone. These numbers can be copied to cloud storage, sent to another device, displayed on any number of screens. This may sound a bit philosophical but it's worthwhile to notice the transition from things to representations of those things as data. And since data analysts work with data, it's a basic skill to be able to identify what is and is not data. Also please note that from now on, whenever I say data, I always mean digital data unless I specifically say otherwise.

1.2 WHY ORGANIZE DATA?

Remember the movie poster from the last video. You look at the image and with the superpower of your own human brain, you interpret what you see to answer some simple questions. Who are the actors in the movie? What are the showtimes at this theater? You organize the sight of the poster to get some information that interests you. If you take a photograph of the movie poster with your phone, you do have data, but the form of the data as image pixels determines what use you can easily make of that data. Think about what you need to do if I share with you my photographs of every movie poster on display in a city, and then you want to find all the movies starring one certain actor. Now, imagine a simple table or spreadsheet with each row containing the movie title, the actor names and the showtimes.

This may not be as aesthetically pleasing as the pictures of posters, but it makes things much simpler if you want to find a movie with some actor or answer some simple questions about a movie. Notice, of course, you cannot answer any questions about information that is not included in your data. For instance, your table with movie title, actor names and showtimes, cannot tell you whether there are tickets available for a certain showing of the movie. What if instead of sharing my movie poster photos, I share all my photos including shots of restaurant menus, friends I meet, buildings I like. This may carry more information but it makes it even

tougher for you to answer a simple question about what movies are showing with a certain actor.

The important thing to learn here is that the **organization of data has a major impact on how easily you can use the data to answer questions**. At this point, I'd like to establish a few terms. By the way, you may have colleagues who use these terms with a meaning similar to the definitions here but not identical. It's OK as long as you know that the meanings given here are generally accepted and you can clear up the details with your colleagues if any confusions arise.

A **data store** is a **collection of data of any type**. This is a general term and it can be used for collections of different sizes. For instance, I could call the collection of photos on my smartphone a data store. I might have a cloud storage account with photos, videos and text messages. This is a different, larger data store, and I could upload all my photos from my phone to this larger data store. The service that maintains my cloud storage also has a much larger data store of all the documents saved by all of its users. **A database is an organized data store**.

A simple example of a database is a spreadsheet containing movie titles, actors, and showtimes. You can even organize your photos and call that a database. A database can contain organized data about different topics. For instance, I can have another spreadsheet of movie theaters, with name of the theater, address, and parking information, if available. If I like, I can keep the movie spreadsheet and the theater spreadsheet together in the same folder, and I can call those different parts of the same database. **A type of software that helps you organize data is a database management system or DBMS**. A DBMS enables you to create a database, add and update data, and easily retrieve data based on its organization. Electronic DBMSs were first developed in 1960s and have enjoyed increasingly widespread use since then. Informally, you may call the combination of your database and the DBMS software you use a database system.

1.3 DATA EXTRACTION FROM DIGITAL IMAGES

The examples involving the movie poster, and the information you obtain by reading it, invite the question: If I have a digital photograph of a movie poster, couldn't reading the photo be a form of data analysis?

The answer is certainly yes, but that image processing would be a kind of data analysis that is very different from the focus of this course and specialization. This reading covers such image processing and related topics briefly.

To have a computer with a digital camera "see" a movie poster, and then interpret the image in a useful way, is in the general domain of computer vision. Identifying objects or letters represented in a digital image is a kind of classification, a type of machine learning in increasing use today.

A classifier is a type of computer program that can take records with potentially many data points, and can infer one or more simple categorical values that are suitable for the record. For example, given a picture of an object (a record with many pixel values), name the object (say, "cat," "house," or "table"). The program performs the computational task of resolving all the pixel values to the simple label.

Modern classifiers can "learn" how to classify pictures by first being presented with a large number of pictures that are already properly labeled. Even after "seeing" millions of examples, a classifier system may well mislabel a new picture with different lighting, or a new style of house or table, or an image presented from a different camera angle. Users of such systems always measure accuracy by reporting what percentage of the labels coming from the program are correct:

computer systems are not perfect at classifying images. Even the latest, profoundly compute-intensive algorithms like deep learning systems report a percentage of successful classification tasks, not perfect accuracy.

When you think about it, it's not just a problem with computers: people are not always 100% accurate in identifying what they see, either. However, considering the number of neurons in your brain, and the amount of sensory input you have processed in your life, it's no surprise that you can far outperform a computer in looking across the street at a movie poster and extracting its information from your visual field.

Autonomous vehicles represent a great technical accomplishment in computer vision (and other forms of signal processing), but in the near term you can expect these applications to require more restricted settings than the ones an average human driver can manage, such as unpaved roads, pedestrians, animals, or unexpected changes in terrain.

By the way, one narrow form of computer vision is already in wide, successful use: **optical character recognition**, or OCR. If you have an image—a record of pixels—and you know you are looking for letters or numeric digits, it's a relatively simple matter to scan the image for those characters and signal when and where they appear. The success of OCR systems today is evident when you deposit a check in an automatic teller machine or scan it using a mobile check deposit app. The software finds the images of digits for the check amount and interprets each digit as one of the ten printed numerals, 0, 1, 2, up to 9. Still, the ATM or app will typically require you to verify the amount, to confirm the accuracy of the OCR software!

This reading began with the question, if I have a digital photograph of a movie poster, wouldn't reading the photo be a form of data analysis? Working with the compute-intensive, somewhat fuzzy problems of image recognition can be called a form a data analysis, but the work of someone with the job title of data analyst is likely to be rather different.

Big Data Analysis with SQL, the subject of this specialization, focuses on the use of data records that are already very **well organized with clearly defined features**:

records such as customer orders, airline flights, or store items. A data scientist, working with machine learning algorithms, may extract some clear features from data such as images, even though the accuracy of such features must be assessed as less than perfect, and you can then go on with analysis using the SQL tools you will learn here. Indeed, modern enterprises gain useful insights from the mass of data they have today using an interplay of the data analysis skills you will develop here, and statistical techniques such as machine learning.

1.4 WHAT DOES A DBMS DO?

Stop a minute to consider the likely clutter of files on your phone or your computer. What are the consequences of the amount of organization or lack of organization in your own data? What if you had 100 times or 1,000 times more data than you have now? What if you need to share your data with other people, and they need to share with you?

A database management system can't solve all your data organizing problems but it can certainly help. If you look up the term database management system on the web, you'll quickly find some definition that says a DBMS is software that gives you a systematic way to organize and manage your own data in one or more databases. **The DBMS should give you a way to perform at least four general activities.** You need to design the kinds of data your database will hold, change what data you have in your database, get data out of it, and manage who has access to different parts of your data. Right there, you have four different activities you want to perform with data.

Working with digital data, you need some way to consistently, systematically, perform those four activities. So, the **DBMS helps you design, update, retrieve, and manage your data in a consistent way.** I'm going to discuss each of these activities in turn. When you design a database, you decide what kinds of data you want to have and you set up different places where you want to put different kinds of data. Picture a kitchen; it will have one place for plates, another place for glasses, and another place for eating utensils like forks and spoons. Designing a database is

similar to setting up those places, deciding what different kinds of things you'll keep, and where they'll go. For instance, a database for a restaurant can have records about food ingredients, and different records about employee hours at the restaurant, and still different records about customer sale receipts.

Of course for data, you have not only different places for different types of data, but you also have different features for those different data records. For instance, you can have weights or item counts for food ingredients, or money amounts for sales receipts. The DBMS should help you set this up and keep your data organized. When you update a database, you add data, change data that's already there, or remove data. For the kitchen, suppose you buy 100 mangoes for a special seasonal dessert, over a busy weekend you use half the mangoes, so you have 50 left. Then over the next four days, you use up the rest. You can make changes to the data in your database as these changes occur. Buy 100 mangoes, new record. Use half the mangoes, change the quantity. Use the rest, remove the record. When you record your purchases and consumption of food items in the database, it helps you to consistently keep track of your inventory.

The next general type of activity is surely one of the main reasons for you to have a database: **you use it to answer questions**. In other words, you retrieve data. If you keep the database for your restaurant up-to-date, you can easily use it to find out the answers to many questions. How many mangoes do I have on hand right now? How much salt do I have for cooking? How much sugar? What were the total restaurant sales last week? Finally, the DBMS helps you manage your data. The word "manage" can mean a number of things, but here I mean, the need to control access to your data. You want to set up different user accounts with different access to different parts of the data. For the restaurant database, your kitchen staff persons need to update and retrieve data about the food inventory, but not your employee pay records. Your accountant needs access to the pay records, and so on. The initial electronic databases were created in the 1960s. Those were the early days of electronic computing systems for business. Using the best technologies at the time, engineers cobbled together ways to perform these four activities. And then in 1970, a man named E.F. Codd published an article that revolutionized the way we think about database systems, and that new way of thinking became common and remains incredibly useful today. More about that in the next video.

1.5 RELATIONAL DATABASES AND SQL

E.F. Codd was working at an IBM research facility in San Jose, California in 1970 when he published his pioneering paper, "A Relational Model of Data for Large Shared Data Banks." This paper appeared in the premier academic journal of computer science, the Communications of the Association for Computing Machinery, or just the CACM. Don't we love our acronyms? Codd's colleagues, Don Chamberlin and Ray Boyce, designed several computer languages expressly to implement Codd's ideas for working with data. In 1974, they settled on the language they called Structured Query Language, also called SQL, which may also be pronounced "sequel"—another acronym.

Codd's paper is the basis of relational database management systems or RDBMSs. SQL was especially made to work with RDBMSs. Nearly all of the popular database systems since the 1980s are relational systems and nearly all of those use SQL as their primary language. Some examples of SQL-based relational systems are Oracle, SQL Server, MySQL, DB2, PostgreSQL, Microsoft Access, and SQLite. Note, the big data era has seen the rise of other types of databases called "NoSQL" databases. That is a topic for later in this course. The great thing about SQL is that it's so simple and easy to learn. All four of the database activities from the previous video are their own simple commands in SQL.

The holding areas for different kinds of data in SQL are called tables. When you design your database, you set up different tables for different kinds of data. There are simple commands or statements to create a table, change what types of records that table will hold, or discard the table from your database. The SQL commands for these are, simply enough, CREATE, ALTER, and DROP. These commands cover the activity of designing, or more accurately defining, your database, and together they form a category of SQL commands called Data Definition Language or DDL.

Don't be intimidated by the word "language" in the label "Data Definition Language." It's just a category of SQL commands that performed one of the four activities I've already identified for a database management system. The next activity you want to perform is **to update** the data in your database. You want to add records to tables, change some of the data in those records, and remove records. The SQL commands for those actions are **INSERT, UPDATE and DELETE**.

They form the SQL category of **Data Manipulation Language or DML**.

Of course, once you set up your tables and then put some data into your database, you can do one of the main things people do with data: you can ask questions about it. The **SELECT** statement is the superstar of the SQL commands. It is the one command in SQL for asking questions and getting answers out of your database. The SELECT statement is so important that it has its own category all by itself: the **Data Query Language category or DQL**.

Finally, to have different users with different roles use your database, you can use the SQL statements **GRANT and REVOKE** in the category of **Data Control Language or DCL**. You can use the GRANT statement to create a user account or give a user certain privileges in your database. REVOKE can remove a privilege that was previously granted. For instance, you might want your users Natasha and Talia to be able to see and change what's in the employee table, while user Daniel can only see what's in the table, and user Meg shouldn't see the employee table at all.

The GRANT and REVOKE commands give you simple ways to manage these different kinds of access. Now you know the four general types of database activities in the basic SQL commands for them. The order of the statements given here has some meaning. You must create a table before you can put any data into it, and then you must put data into the table before you can query it. However, if you go by how often you will use these statements, it's a different order. SELECTs are what you will do the most often by far. DML statements take second place and DDL and DCL maybe fairly rare. So, here you have the most fundamental SQL statements and their categories. I suggest that you make sure you are familiar with these terms. If they are new to you, you may want to copy this list out by hand and

look at it often for a few days. Remind yourself of the meanings of these terms until you can use them all comfortably in conversation. Please see the reading this week, with a few special notes about SQL.

In the video "Relational Databases and SQL," I presented some basic SQL commands, which fall into four well-known categories. The following three notes provide some detail and warnings about those categories.

1.6 THREE NOTES ABOUT SQL

Rare or (sometimes) muddy terms: DQL, DML, query.

The terms on my list are all commonly used in discussions about relational databases and SQL, with one exception: **DQL (Data Query Language)**. Because DQL is a category with only one statement in it, some people may never learn or say "DQL"; they just say "SELECT."

In fact, some people include the SELECT statement in the category of DML. I don't use that classification (and I think I'm in the majority), but when you're talking to someone new, be aware that you may need to clear things up with them when they say "DML": Do they mean INSERT-UPDATE-DELETE statements, or do they also include SELECT in the DML category?

There's one more muddy word: **query**. For most people, the word query is another name for the SELECT statement. This is reasonable, because the English word query is another word for question, and whenever you ask the database a question, you do so by issuing a SELECT statement. But for some people, query can refer to any SQL statement, of any kind at all! This doesn't make good sense to me, but I've heard the words "database query" used in this way many times through the years.

1.7 COMMANDS, IN AND OUT OF SQL

I've started here with just the beginning, foundational concepts of SQL. These are not the only commands in SQL, and you'll quickly learn others. For example, you'll probably like the DESCRIBE command, to remind you of the details about how you designed a table. Or there's UPSERT, a funny kind of statement that can be an INSERT or an UPDATE depending on the circumstances.

There are even some commands that are provided with most database systems, but that are not SQL statements, like commands for backing up a database, or importing data from some other data store into your database. If there's no regular SQL command for something you need to do, that's okay; you'll just need to learn the particular command you need in the particular program you're using.

1.8 STANDARDS AND SQL DIALECTS

Speaking of different commands, there's a twist on SQL itself. The worldwide engineering community has developed a standard for SQL. SQL has evolved and grown over the years, and so the "official standard" has been revised and republished several times. However, almost no commercial vendor actually implements standard SQL exactly, 100 percent. (In fact, in the late 1980s, competing companies lobbied to get features into the standard that their competitors didn't have, and so the standard became inconsistent and no one program could possibly be 100 percent compliant with the standard at that time!)

So, you really have different SQL dialects for different software programs. Although most SQL systems are at least 90 or 95 percent alike, every system will have its own peculiar dialect, with some small differences in the exact commands that are available, or some optional details in the commands. I'm giving you fundamental concepts of SQL in these early weeks. My suggestion is that whenever you actually use a new SQL-based program, you should familiarize yourself with the particular SQL dialect that is special to that program. Then when you use a different program, learn its SQL dialect. You'll quickly deepen your understanding of both SQL in general and the particular use of SQL in the various programs you use. That is exactly the approach in the later courses of this specialization.

1.9 THE SUCCESS OF RDBMSs AND SQL

Ask anyone who's been around computing for a long time, and they'll tell you. Relational databases and SQL have enjoyed wide use for a remarkably long time. I can give several reasons for the success of RDBMSs and SQL.

- 1: E.F. Codd's original idea of the relational model is a brilliant, mathematically rigorous idea. His concept organizes the use of data that is clean and flexible, allowing users to access data from many different starting points with equal ease. In fact, Codd's work on the relational model was so groundbreaking that he received the ACM Turing Award for this work in 1981. The Turing award is the highest award there is in the computer science world, the Nobel Prize of computing.
- 2: SQL, which was originally developed for relational databases, is **easy to learn and use**. It covers most of the operations you want to perform, in a simple, coherent language. There is one RDBMS called Rel that does not use SQL, but every other one I know, does.
- 3: Both the relational model and SQL provide precise concepts for organizing and using data, but without any low level details about how an actual program should do the work. As a result, many different implementations of the relational model in SQL have been possible using different computers, different file systems, operating systems, programming languages. Innovators have kept coming up with lighter or cheaper or faster programs that still keep to the general design of a SQL-based RDBMS.
- 4: Most RDBMSs even provide ODBC (Open Database Connectivity), and JDBC (Java Database Connectivity), interfaces. These **allow virtually any programming language to issue SQL commands to a database**. As a result, nearly any program of any kind that needs to maintain some data from day to day can do so, using some form of RDBMS. There are probably several relational databases embedded in your phone right now.
- 5: Because RDBMSs and SQL have already "solved the problem" of managing data, engineers can build specialized applications that use RDBMS software to handle the data. For example, applications for accounting or legal libraries or medical records to name just a few. There are countless tools that add

useful functions to databases as well, like reporting and graphical displays of data.

- 6: Because SQL is so common and so many applications and tools "speak SQL," there's even been a surge of data stores that are not relational databases, but that still provide some dialect of SQL. There's great advantage in keeping to "the SQL way of thinking," even with non-relational systems, so that well established user skills and software skills that use SQL can be easily adapted to work with these new data stores. If you already know SQL then you'll find that this specialization will teach you to adapt your SQL skills to the new big data environment. The links below direct you to Codd's paper, which I referred to in the videos, and also to a paper by Don Chamberlain on the early days of SQL. These are excellent articles to read if you are interested in knowing more about the beginnings of relational databases.

E.F. Codd, "A relational model of data for large shared data banks",
<https://dl.acm.org/citation.cfm?doid=362384.362685>

Donald D. Chamberlain, "Early History of SQL",
<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6359709>

1.10 OPERATIONAL AND ANALYTIC DATABASES

Comparing Operational and Analytic DBs: SELECT Statements

Both operational and analytic databases are readily supported using RDBMSs and SQL. Both use DDL and DCL statements to define, organize, and manage the kinds of data to keep. For smaller applications, you can use the same RDBMS software for either type of database. In fact, you may even have one database serving both purposes.

However, the distinct needs of some applications can drive these two different types of databases to have marked differences. I wanted to **distinguish between search and analysis as general query types**. By search, I mean directly finding some data that's explicitly present in the database. For instance, given a customer's name, what is their phone number? This lookup function can be quite useful and is nearly the simplest kind of database query.

But SQL SELECT statements can also do many types of analysis in which your queries combine different parts of the data to uncover new information. Here are two simple analytical questions. In which months of the year are there more house sales in some city? What college course is taken the most often by second year college students? This information may be implicit in a database, but may require more complex queries than just a lookup to find the answers. Now, an operational database will usually serve mostly search queries, and an analytic database will serve mostly analytic queries. This is not a hard distinction. You'll use a mix of SELECT statements in most databases. But there is a pull toward mostly one or the other type. The operational system provides a kind of short-term memory to your applications. It allows programs to retrieve current data in order to keep up with some current state of affairs. **While in analysis you look deeper and deeper into some dataset to gain more and more understanding.**

1.11 COMPARING OPERATIONAL AND ANALYTIC DBs: DML ACTIVITY

Another difference between operational and analytic databases is in their frequency and type of SQL activity, especially DML activity, that you'll see more often. Remember, **DML, or Data Manipulation Language, refers to INSERT, UPDATE, and DELETE** statements, and these insert, update, and delete records in your database tables. Think about a database for airplanes entering and leaving an airport. Every time a plane approaches, a record is added to the database.

As the plane moves into landing and taxiing, its position is updated in the database. When the plane flies away, its record may be deleted. The database should accept individual DML statements updating the status of each plane. So, for a busy airport,

this can mean many DML statements per minute. The database also serves many SELECT statements per minute as different display and coordination systems keep track of the planes continually.

There's a well-known acronym for this mix of activities: **CRUD, for Create, Retrieve, Update, Delete**. An INSERT statement creates a record, a SELECT statement retrieves records, and UPDATE and DELETE statements update and delete records. I know the word "create" here is used to refer to the SQL INSERT statement, not the SQL's CREATE statement that creates a table, but I didn't make up the acronym! The word describes the continuous activity you can expect in an operational database, a mix of DML and SELECT statements.

That's what the operational system does. CRUD, CRUD, CRUD, all day long. For some people, crud is another word for junk, but certainly no insult is meant by CRUD here. Larger operational database use cases are often characterized by the number of database operations per unit time that they perform on average and at peak: often in the thousands of operations per second, and sometimes over a million database operations per second. A further characteristic is the "read-write mix." What percentage of these operations are SELECTs and what percentage are DML statements? Notice though that the understanding is that it's all CRUD activity. DDL and DCL are typically not everyday occurrences. In contrast to operational systems, an analytic database will usually not undergo continuous DML.

Suppose you want to learn about traffic patterns for the last year. You may obtain data from a separate data store about traffic delays over the last 24 hours; maybe you'll get this from an operational database. Every night you copy those records into your analytic database, and additionally you accumulate several years' worth of data. In the nightly activity, you add hundreds or thousands of new records in one go. The nature of the nightly addition of new **data is not continuous DML, but a bulk load of new data**. You may also use bulk loads to bring other static data stores into your analysis. For instance, you may want to include census data about the population density in different areas to enrich your understanding of traffic patterns. Another term for bulk load is data import. SQL does not have a standard statement for importing data in bulk, but most DBMSs do provide utilities or commands for doing so.

1.12 OPERATIONAL AND ANALYTIC DATABASES: FURTHER COMPARISONS

By now, I'm sure you've guessed that **analytic databases are often larger in size** than operational databases. If one database contains traffic data for one day, and another contains similar data covering three years, it stands to reason that the second database will have around a thousand times more data. Really, the size of an analytic database is often limited only by the maximum amount of data you can store and having more records can potentially yield more insights. Finally, I want to point out that at larger scales, the differing needs of operational and analytic database applications have pushed the development of different database technologies. Some operational systems support high volumes of DML statements, but limit dataset sizes to no more than a few thousand or a few million records total.

Other analytic systems will instead support sophisticated analytic SELECT statements, and even other analytic techniques against much higher volumes of data, while preferring bulk loads as a way to add data. I'll say more about the technical differences between operational and analytic databases next week. And I'll cover an important category of operational databases: transactional systems. Then later in the course, I'll expand the technical discussion of the differences between operational and analytic systems, as we move into big data.

WEEK 2

Learning Objectives

- Apply concepts of table and column design to existing data to identify compatibility between design and data
- Defend a preferred approach to database design (normalized or denormalized) based on a database's intended use
- Explain why database transactions and special features associated with Data Manipulation Language are not critical for analytic databases

Do you think relational databases get that name because they contain related tables? No! Would you guess that a "denormalized table" is called that because it's unusual? No! **ACID (Atomicity, Consistency, Isolation, Durability)** No. This week, I'll give you the concepts and special vocabulary that are common to all users of SQL and relational databases. You'll learn about the fundamentals of database design, and what primary keys, foreign keys, and database triggers are. These concepts are essential to your being fluent in the different uses of SQL on data, big and small. If you already know SQL, you'll get through this week very quickly, but maybe, you'll find the material interesting, and maybe I'll fill in a few gaps for you. And, I hope you'll go to the discussion boards and help round out the concepts I present for your fellow learners.

2.1 INTRODUCING TABLE SCHEMAS

As I've said before, the main organizing concept of relational database management systems is a table. Relational database theory has a formal concept called a "relation." A table implements a relation, and that's how we get the name relational database management system. A table has rows and columns. You might think of this as similar to, say, a spreadsheet. However, a table always has a strict set of columns, and then any number of rows. Look at this toy table. This table is set up to have just the columns id, name, and price. A row (or record) in this table will have exactly these three data elements: an id, a name, and a price. A row cannot have any extra data elements besides these three. But the table has no set restriction on the number of rows. Here you see 3 rows, but there can be no rows,

or a thousand rows, or a million rows or more! (If anyone can think of a million different toys to record in the table.) Look a little closer at this table, at the kinds of data you see in each column. The id column has whole numbers. The name column has words, and the price column has money amounts with units in hundredths. (Maybe dollars and cents, or maybe some other kind of money: there is no indication of what monetary unit it is.) The table is made so that each column has a declared data type, which determines what kind of data can go into that column. So then any row is just a combination of legal values for each of the columns in the table. Look now at the declared data types for these three columns. The INT data type says that the id column can only store an integer. (The allowed range is around plus or minus 2 billion, but let's not worry about that right now.) The name column has data type STRING, which is a simple data type for character data. Any character, including numbers and other symbols, can be part of a string. The price column has a different kind of numeric value: a decimal number, with up to 5 digits total, and with 2 of those digits to the right of the decimal place. So, a DECIMAL(5,2) column can have a value as high as 999.99, but no higher. A DECIMAL data type like this is good for money amounts, because it will always stay accurate to the exact number of decimal places you give in the data type. A really important idea to get here is that the table has no set limit on the number of rows it can have, but has definite limits and rules on what can go in any row. The table has a strict definition for its column names and their data types, and this comprises the structure of the data your table can accept. So, you can just sketch out the design of the table by stating the column names and their data types. See how this describes the table? This doesn't show any of the data, the actual toy records, that can be in the table. But it shows the kinds of rows or records that the table will accept. This is the structure of the table, also called its "schema" or "metadata."

2.2 NULL VALUES

Table columns each have a name and a data type, but they can also have other properties. Here's one more column property for now: NOT NULL. Sometimes a row can have no value at all for one or more of its fields, which are the values for the different columns. The special term for no value is NULL, or a NULL value. Note that NULL is not the same as zero for numbers. And for STRING columns, it's not even the same as an empty string, a string with no characters. NULL is the absence of any value at all. So then another property of a column is that it can be NOT NULL.

This means that a row must have a value supplied for that column, or it is not a legal row. Look at this revised schema. Now there are notes that the id column and the name column are both NOT NULL columns. So, a row in a table must always have a value for id and a value for name. But this note does not appear for the price column, a row may occur with no value given at all for the price. Look at the table schema and the data given here. The values in every row conform to the data types of their respective columns, and the NOT NULL columns have values in all the rows, so all the rows are legal according to the table schema. Notice here that the word "NULL" is presented for the price in one of the rows. Many of the database systems will have some way to indicate a NULL value in a row whenever the row is printed out, and the printed word NULL is one common way.

This first course emphasizes concepts, and so the syntax for INSERTing rows comes up in a later course. However, I can tell you that there are two common ways to indicate a NULL value for part of a row: one is to leave the value out of the row INSERTed, like so. This is not SQL syntax: I'm just giving the data for a row presented clearly. Since no value is given for price, the row must use a default value. In some systems, there could be a default for columns whose values are not supplied on an INSERT. If no value was specified in the table schema, the missing value must be NULL. The other way to supply a NULL value is to use the word NULL explicitly in the row values. It's important to get to know the schema of any table, because that tells you what kind of data can be in the table. And so, what kind of information you can expect to store and retrieve. There's more to say about data types and I'll continue with that in the next video.

2.3 DATA TYPES

We've looked briefly at a small table and its simple schema. Every bit of data in a relational database is a row in a table, and so the data permitted in your database will be determined by the table schemas you define using CREATE statements. Since your table schemas must always provide a data type for every column, the data types determine what kinds of data you can have. If you study the dialect of SQL used by any system, you'll find a list of supported data types. For example Apache Impala includes these data types: There's no need to learn each of these data types right now, the later courses will include these in a detailed study. For now, it's important to note two things. One, a relational database design always gives a data type for each column in a table. And two, different SQL dialects, for different software systems, will have some differences in the exact list of data types

supported. So if you actively use a particular system, you want to familiarize yourself with its available data types. In fact, if you copy data from one system to another, you may need to adapt some columns from the data types in the one system, to use the data types available in the new system. There are two special column data types that I want to mention briefly here: BLOBs and CLOBs. "BLOB" stands for Binary Large Object, and "CLOB" stands for Character Large Object. Look at the maximum allowed sizes. That's right: a table might have a BLOB column that stores, say, an entire movie in each row, or a digital x-ray. Or, a CLOB column might store the entire text of a book in each row. These large data types are very unusual, because many SQL dialects do not support them at all, and those that do usually have very limited support. There's typically no built-in way to use these columns in any analysis.

With standard SQL dialects, you cannot sort, or search, or calculate anything with the values in these large data types, and so these columns cannot be used in the kinds of SQL commands that do those things. At most, such column values can be stored in a table, and then retrieved from the table. In fact, database systems that do support BLOBs and CLOBs usually store them in a part of the file system that is altogether separate from the storage of all the other columns of a table. Again, these large data types are supported in some RDBMSs, like Oracle, but are not supported universally. For instance, Impala does not have BLOB or CLOB in its list of allowed data types. Also, the size limits of a BLOB or CLOB may be different in different systems. There's one more general category of data type: complex data types. Those are unusual data types in the conventional relational world, but they add flexibility in your table designs for big data. I'll discuss complex data types later in the course.

2.4 PRIMARY KEYS

There are a few other column properties that you can set, depending on the particular RDBMS you use. Here are a couple of those other properties: a column definition may define a default value for the column, or a numeric column may allow only positive values. It's good to learn the complete list of available column properties for whatever system you use, but I'm going to skip those to get to the next important concept. Here are two key properties that take you from columns further into overall table design: primary keys and foreign keys. "Two key properties" - get it? These properties are not required by all RDBMS software, but they are frequently supported, and they enable a well-known approach to database

design that I'll discuss this week. A "primary key" is a column - or sometimes a set of columns - that is used definitely, uniquely to identify any row in a table. I'll talk about single-column primary keys, called simple primary keys, for now. If a table has a primary key column, then for any row, the value of the column in that row identifies exactly that row, and no other row in the table. An RDBMS supports a primary key by making the column NOT NULL, and also unique. This means that if any record is submitted for INSERT to the table, the primary key must be included as a non-NULL value. And also, the database system will check all rows in the table at the time of the INSERT, and will reject the new row if its primary key value is found to be already present in the table. In this way, the primary key prevents the table from ever having two rows that are exact duplicates of one another, because no two rows can have the same primary key.

Look at this table: The RDBMS does not require you to have a primary key but the id column may well be the primary key for this table. Here the numbers in the column cover a sequential range of integers. That may not be the case, and it doesn't matter. The important thing is that there can be no two rows with the same value in the id column. So, a primary key value will always locate exactly one row in the table - or no row if the primary key value isn't found. When you design a table, you may find more than one column that could be used as the primary key, but just be aware that RDBMSs usually permit no more than one primary key, so the schema for your table will have no primary key, or one primary key, not more than one. Columns with "id" in the name are often used for primary keys. Employee id, customer id, store id: these can be used to identify exactly one employee, or one customer, or one store.

Another aspect of a primary key is that it must never be changed. For example, you can keep an employee table up-to-date by changing the name, or the salary, or the department of an employee, but the primary key - say, the employee id - never changes, and in this way the other changes reflect a change in the data about the employee, but not a change from one employee to another. This rule of no changes allowed, or "immutability," for primary keys is often not enforced by RDBMS software, but it nevertheless is an important principle for you to follow when you think about table design. Systems use different ways to represent a primary key in a table schema, but here's one: The note "PK" indicates that the id column is a primary key. PK automatically implies that a column is also NOT NULL and must have no duplicates in the table, so indicating these is optional. This design tells you that the table will always have a value for id in every row, and that no two rows will

ever have the same id. Of course different rows may have the same price. As for the name column, it may or may not make sense for you to have two rows with the same toy name, but this table design does not say anything about that: it only restricts the use of duplicates in the id column. I'll talk about foreign keys, and a bit more about primary keys, in the next video.

2.5 FOREIGN KEYS

When a table has a "foreign key," it means that a column refers to some other "primary key." Look at these two tables. Now the toy table has a new column, `maker_id`. Looking at the data, I hope you can guess that the `maker_id` column refers to the `id` column in the maker table. You can read the two tables together to see the name and city of the maker of each toy. When you have multiple tables, you can always single out a specific column by writing its table name, then a period, then the column name. Since both tables have a column named `id`, you can use their long names, `toy.id`, and `maker.id` to be clear. You can always use these longer names if you like, not just when it would be unclear. So here the maker table has primary key `maker.id`, and the toy table has foreign key `toy.maker_id`, which refers to `maker.id`.

The design of the two tables can be presented like this. The note for the `maker_id` column of the toy table states that this column is a foreign key, and that it refers to the `id` column in the maker table. Or in other words `toy.maker_id` refers to `maker.id`. Foreign key columns are always NOT NULL, and what this foreign key property means is that any value in the `maker_id` column of the toy table *must* appear in the `id` column of the maker table. Whenever you attempt to INSERT a row in the toy table, the value you supply for the `maker_id` is checked against the maker table, and your row will be accepted *only* if the value is found. So, by this design, your system implements a rule: every toy in the database must have a maker in the database.

In the last video I said that a primary key uniquely identifies a row in a table, and that it can be a single column or a "set of columns." Here's an example of a multiple-column key, called a "compound primary key." The unique identifier of a row in the `shoe_at_store` table is not one column, but the combination of `shoe_id` and `store_id`. These two columns together comprise a "compound key." They are both NOT NULL columns, and the combination of the two must always be unique for each row in the table. Notice that each column of this compound key is itself a

foreign key: the columns refer to the primary keys in the other two tables. You can see that the price column of the shoe_at_store table tells you not the generic price of a shoe, but the price of a shoe at a particular store. Looking at all three tables, you can see that the Women's Classic Ugg boot sells for one price at Boot Place and another price at Bear Foot. The shoe and store tables have designs similar to the ones you've seen before: The design of the shoe_at_store table has a compound primary key, containing two foreign keys.

Remember, I said in the last video the table can have at most one primary key. So the note PK for the shoe_id and the store_id columns tells you that these form a compound primary key. They are both NOT NULL and the combination of shoe_id and store_id values must always be unique in the table. The overall design across these three tables might be visualized like this. This gives you a big picture look, above your table designs, to your overall database design. You can see at a glance that the shoe_at_store table uses a compound key to represent the combination of shoes and stores in your data. You can expect to find data about shoes at a store, like the price or available inventory, or you can go to the shoe table to find out more about a particular shoe style, or go to the store table to find out about store details.

A SELECT statement that uses these tables in combination is called a JOIN query, and you'll see in the next course how easy SQL makes it for you to write JOINS and work with these table combinations. This use of compound keys to place a table "between" other tables is easy to do. It's not always the best way to set up your database, but it can sometimes be suitable, giving you a clear organization and predictable ways to find information. Here's an illustration of another example. I'm not going to discuss this database in detail here, but I think you can see what it represents, and this kind of design is called, nicely enough, a star schema.

2.6 TWO STRATEGIES FOR DATABASE DESIGN

In order to store data in a relational database system, you or someone else must first create the tables that will store your data. So, your choices about your tables provide the organization you will have for your database. I've given you all the basic building blocks you can use to create your tables: columns, with data types and other column properties, and primary keys and foreign keys. I've skimmed over a

few other column properties, like default values in case you don't supply a column value when you INSERT a row. However, I have explained the basic parts of table design. With these basic building blocks, you have a lot of flexibility in how you design the tables for your database. Look at this small database. We saw before that these tables have a design with primary keys, and the toy.maker_id column is set as a foreign key referencing the maker.id column. Look at this alternative table.

The RDBMS does not require you to have a primary key on a table, so this is a possible way to store the data. But you have to wonder, what is this table really about? If it's about toys, then what is the meaning of the row with the maker but no toy? Just looking at this table, can you guess that the key is about the headquarters location of a maker? If you delete one toy row from the table, don't you risk deleting the maker at the same time, whether you want to or not? What about this design? This time you have a single table that emphasizes makers. It's interesting, though, that the toys column combines the toy name, a string, with the price, a money value, and there are two of these in one row.

Of course, as more data gets added, you'd probably have one maker that makes dozens or even hundreds of different toys, so that this column alone could become quite large on some rows. SQL provides the simple CREATE statement to create a table, ALTER to change some properties of the table, and DROP to discard a table. These DDL statements are so easy to use that you can create a handful of tables in a minute or two. This is great for practice and learning, and may work well enough for a small database, but beware! If you create a database that will get larger and will see production use, you will likely have troubles that you can avoid if you start instead with a conscious approach to your database design. With all the freedom in how you set up your tables, it can help if you understand two general strategies of database design. One strategy of design is database normalization and the other is database denormalization. In the next videos, I'll talk about normalization, denormalization, the differences, and the trade-offs between these two strategies.

2.7 DATABASE NORMALIZATION

Database normalization is a strategy whereby you design each table so that it obeys certain organizational conditions or rules. The rules build to more and more strict forms of table organization, and are sequenced as First Normal Form, Second Normal Form, Third Normal Form, and so on. There are some even more strictly

organized designs called Fourth, Fifth, and Sixth Normal Forms and a few other forms in between, but the business community generally accepts Third Normal Form as the level of organization to aim for when you adopt a strategy of database normalization in your design. Let me give you an informal list of the conditions that meet Third Normal Form. This list is not numbered in relation to First, Second, or Third Normal Forms; it is a summary of Third Normal Form in its entirety. I'll discuss each of these conditions in turn. First, every table should have a primary key. I'll add that you should think about the primary key carefully. Look at this table. This table is improved with a primary key, which is part of Third Normal Form. Now you can have two customers with the same name, and still identify them as distinct from one another.

The primary key can be used as a foreign key in other tables to refer to a customer. The customer can change names, and it's clear: this does not change customers, it changes something about a customer - the name - but with the identity of that customer remaining unchanged. A common practice is to avoid using "intelligent keys" - that is, use a primary key that identifies a row, but does not have any meaning about the item represented in that row. This principle is not required to put a table into Third Normal Form, but it is usually good practice. The second "rule" or condition is that every column should be atomic, or indivisible into smaller parts. This doesn't really mean that you can never have a column with parts, but you should never have parts that appear separately somewhere else in the database. Look at this table.

Here the city and state code appear together, so you might think that this is not a column with atomic values. Here's the real issue: Do you expect to never have or need a separate table with data about a state or province? If so, then you do **not** need to split this out from the original city column. Although it has two parts, for your purposes, neither part will ever need to be separated out. On the other hand, what if you have another table like this. Here, having another table with state or province codes, there is not a simple way to organize the two tables together, since the two-letter state codes are not clearly set apart in the maker table. This problem can be solved with a change to the maker table.

Now, the maker.state_or_province column is atomic and connects directly as a foreign key to the state_or_providence.code column. So "atomic columns" means atomic for your use in your database organization. If you want your database to include more information about **part** of a column, then normalization strategy is to make that a separate column, so that it can be used as a foreign key and refer to

another table that gives more detail. Note you may have reasons of your own for keeping parts of your data in separate columns. For example, you may choose to have a customer's first name and last name in separate columns, so that you can sort by last name or use the person's first name in a letter. The important thing is that you think about how you will use your data, and organize your database with your use in mind.

The third condition of normalization is similar: a single row should not have multiple values for one type of data. Consider this very simple table. The items column has more than one element in the rows shown. When you have this one-to-many collection of data elements on one row, it is called a "repeating group," and this table design can create some difficulties. For instance, it makes it difficult to answer questions about whether a certain item is in the shopping list, or how many shopping items there are in total. This only gets worse if you have 30 items on some rows, or 200! To avoid repeating groups, break your column into multiple items into a separate table, like this.

Now there is one item per row in the shopping_item table, and one store per row in the shopping table, so there are no repeating groups. This kind of two-table design is common: orders and line items, recipes and ingredients, companies and departments, and so on. The next condition says that non-key columns should represent only information about the primary key, and not other non-key columns. Consider this example. This is a table about toys, and the id column is its primary key. It's clear that the name, price, and maker columns give facts about a toy. However, the city column is not really about a toy, but about the maker of a toy.

You've already seen a design with a more normalized approach: have a separate maker table, with city as a non-key column describing the location of the maker. Finally, Third Normal Form maintains there should be no derived columns. Look at this data. The total column is calculated as the amount, reduced by the discount percentage. Third Normal Form identifies the total column as redundant, since it can be derived from the other two columns, and so a normalized design removes this column from the table. The idea is that with the derived column present, if any of the three values - for amount, discount_pct, or total - is changed, this will introduce an inconsistency in the data - and the design gives you no way to prevent or resolve the inconsistency. There are more rigorous definitions of Third Normal Form, but just know that I've given you a reasonable definition in plain language: a table in Third Normal Form has a primary key; all columns are atomic and with no

repeating groups; non-primary-key columns give facts only about the primary key, and there are no derived columns.

2.8 DENORMALIZATION

If you guessed that "denormalization" is the opposite approach to database design as "normalization," you're not too far off, but there is a little bit more to learn. The strategy of denormalization is to consciously, deliberately "break" one or more of the rules of database normalization in your design. Of course, these rules are not actual laws that you are compelled to obey - they are just principles of one approach. You will find advantages and disadvantages to either design approach - normalization or denormalization - and I'll discuss some of the differences and trade-offs between these two approaches in the videos after this one. For now, I want to illustrate some examples of denormalized table designs.

Consider this table of shopping items. Here the primary key is the combination of date and item. You need both to identify a row uniquely. But what if you have these shopping items in a table, but you don't have any date information? Your table may look like this. You can see that there are two rows with the same item and the same quantity. So, there are duplicate rows and there is no primary key. Although this is not a normalized table, it may store information that you find useful: every time an item of shopping occurs, the table records that event. So this table records two occasions of the purchase of apples in the same quantity, and the "duplicate rows" are not really duplicates for your use. Another form of denormalization is to "pre-join" normalized table data.

Sometimes storing data in a denormalized pre-joined table improves performance, because it saves some queries the work of querying more than one table. Remember this normalized set of tables. These tables are in Third Normal Form. They provide details about individual shoe types, and stores, and the prices of particular shoes that particular stores. When I talked about foreign keys before, I noted that an SQL SELECT statement that combines information from related tables is called a JOIN query. This is because the SELECT includes information about how the tables combine, or join to one another. Here's a single denormalized table that pre-joins the tables. Now a query on this new table does not need to include any JOIN to find all the information that was previously stored in three tables. Such a query may be faster because it doesn't have to do the work of combining different tables. Another form of denormalization is to store a "derived column" in a table. Look at this table. Here, the "total" column contains the sum of amount

and shipping, though it is derived from the other two columns. Sometimes derived columns are added to make their data quickly available.

Another kind of redundancy of storage is to have a table composed entirely of summary data, like this one, assuming that there is a separate table about customers, and a table with individual orders. Like the derived column, this summary table can let queries retrieve these count and total values quickly, without having to calculate them from the other more detailed table. Now I've given you four examples of denormalized table designs. These have been examples of denormalization, not an attempt at a complete list. Note that some writers argue that a summary table does not literally violate Third Normal Form. Nevertheless, I think I'm justified in putting the use of summary tables with the others in discussing an approach to normalization or denormalization.

It is a commonplace notion among experienced database designers that, if you throw together a few tables without regards to the principles of normalization, then you do **not** have a denormalized database design: you have a mess. This is sometimes called a "non-normalized" design. A disciplined approach to denormalization is to start by defining a normalized set of tables, considering the principles of good keys and column properties, and then deliberately relaxing your normalized design for specific reasons. This practice can help you better understand your data. I've used these last two videos to explain database normalization and denormalization. In the next two videos, I'll talk about the differences and trade-offs between these two design approaches.

2.9 DIFFERENCES

Both normalized and denormalized database designs have their uses. In this video, I'll discuss some of the main differences between the two design approaches. First, I'll consider data anomalies or problems with keeping the data well organized. When E.F. Codd first formulated his original principles of normalization, he made it clear that he was trying to help reduce the occurrence of anomalies. Consider the two table designs here. The denormalized design is subject to possible INSERT, UPDATE and DELETE anomalies. Here's an INSERT anomaly. If the person.name column is the primary key and therefore NOT NULL then you cannot add a city without also adding a person at the same time. Here's an UPDATE anomaly. If you change the spelling of San Francisco to "SF", you will generate inconsistencies in the table. Unless you search the table and make sure to update all occurrences.

Here's a DELETE anomaly. If you delete the row for name "Sara" from the table, you will also delete the city Winnipeg, whether you intend to or not. The normalized tables are not subject to any of these problems. Of course, you can still have problems with your database - that's life - but the normalized design does avoid some problems like the ones I've just illustrated.

Next, notice that a normalized database automatically enforces certain rules in the allowable data in your database. Look at these simple tables. The id columns are the primary keys in the two tables and the toy.maker_id column is a foreign key that references the maker.id column. When you define a primary key it is often called a primary key constraint. And a foreign key definition is a foreign key constraint. These are indeed constraints or restriction on the data in your tables. A primary key requires that no two rows in the table can be duplicates because every value of the primary key must be unique in the table.

The intention is that a row represents an individual entity and that entity cannot occur more than once in the table. So, for instance, if you find a row about a maker in the maker table, the primary key assures you that you have found the one row. And you do not need to continue looking for other rows. (That is assuming that your primary key design is wisely done, then you take care not to assign more than one primary key to the same maker.) A denormalized table without a primary key has no such constraint. There could be multiple rows about the same maker with different, - maybe even conflicting - data in different rows.

This may be easy enough to manage with a tiny table, but can cause major problems when the table gets larger. A foreign key constraint places other restrictions on your database. Because toy.maker_id is a foreign key referencing maker.id, the following restrictions all hold on these tables. Whenever you add a row for a toy, it must have a maker that is in the maker table. If you ever change the maker for a toy, it must be to a maker in the maker table. You cannot delete a row from the maker table that would leave toys without a maker. An RDBMS can enforce these primary key and foreign key constraints. So that any DML statement that violates a constraint will not succeed in updating your data and will return an error instead. You can then build your operational programs to always maintain data in your database system. And so the response from the system, on any attempts at DML, can be used as guidance to implement certain rules in your business processes. In this case, for instance, the database enforces rules such as, "you cannot have a toy without a maker." Foreign key constraints can be used to enforce many rules. Like, "you cannot have a customer without an account rep";

"You cannot have an order without a customer"; "You cannot have an employee without a department"; and so on. A denormalized design without foreign key constraints, will not implement any of these rules in the database. Another difference is in the size of your data store. You have probably noticed by now that one general principle implied in the rules of normalization is, "don't record anything in more than one place." Look at these two designs. The storage footprint is over 25 percent larger in the denormalized table because of the repetition of the city name and state or province in each row. For such a tiny table that may not seem like much. But you can imagine that the difference in size becomes more extreme as you add more rows to the person table with more and more redundant storage of city data.

2.10 TRADE OFFS

So far you've seen three main differences between normalized and denormalized design approaches. Normalization prevents some data anomalies, enforces some business rules, and produces a smaller database size than a denormalized database. All these differences seem to suggest that normalization is a better way to go, and it often is, for these very reasons. A primary benefit for denormalization, on the other hand, is that it can improve the speed of your system - especially the speed of some SELECT statements. Look at this table with a derived column, showing the total of amount and shipping. The normalized form of this table would not have this column, and so would not risk inconsistency between the total and the other two columns.

Denormalization stores the total explicitly, which makes it quick for a SELECT statement to sort, search, and report the order total value from the table. This example is trivial, and SQL can do all those things without needing the derived column, but the computation of adding two columns does take some non-zero amount of time. If the function to compute a value for the row becomes more complex, it will take more time, and with more rows in the table, even more time to run a SELECT statement. When you denormalize and store a derived column in a table, you take on the extra work to store it correctly in your rows, in order to make the value quickly and readily available for SELECT queries. Look at this summary table from before. In two rows, this table condenses data from 83 rows of the customer_orders table, 45 rows for one customer and 38 rows for the other customer. This table does not contain all the details of the other table, but it makes a SELECT statement for these summary statistics much faster. Regarding SELECT

speed, look at these two table designs. Consider this simple question: What is the state for the person named Kiko?

A SELECT in the normalized database requires a JOIN of two tables, while the denormalized database allows a simple lookup on one table. So, the redundant storage of the city data, and the effort to keep these extra elements up to date, is a trade-off to save the time it takes to perform a JOIN to find those elements when you issue a SELECT statement. A thoroughly normalized database may require you to join many tables in order to gather all the details you want to find. So for more involved databases and larger tables, you'll find significant differences in the run time of queries against normalized, or pre-joined tables. Remember that analytic databases are primarily built for data analysis. Analytic queries perform summaries and other deep dives into your data, to uncover insights that are implicit in the data as a whole. Because data analysis leans toward queries of greater complexity across more parts of your data, analytic databases tend to perform better with more denormalized designs. Operational systems, with their ongoing mix of DML and lookup queries, often work better with normalized designs. These are not hard rules, but they are tendencies you'll see in good design for the two types of databases.

2.11 LET THERE BE THIRD NORMAL FORM

When you discuss database normalization with an experienced colleague, they may smile and use the phrase "the key, the whole key, and nothing but the key, so help me Codd." (The "Codd" in the last, optional part of the phrase refers to E.F. Codd, who first proposed the fundamental concepts of relational data, including normalization.) This is a humorous wordplay on the commonly known language in which an individual in a United States court of law swears to tell "the truth, the whole truth, and nothing but the truth, so help me God."

This phrase comes close to completely covering the details of Third Normal Form (or 3NF). The detail not included is, for a relation (or table) to be in first normal form (1NF) or higher, every record must have the same number of fields—and this includes the constraint that a column cannot have multiple values.

The phrase originated with William Kent, who wrote that "a non-key field must provide a fact about the key, [use] the whole key, and nothing but the key." His article, "[A Simple Guide to Five Normal Forms in Relational Database Theory](#)," gives

a good, simple explanation of normalization, with references to the authoritative works on the subject.

2.12 DATABASE TRANSACTIONS

Up until now I've avoided discussing a major feature of some - but not all - relational database management systems: database transactions. When I presented nine statements in SQL - SELECT, INSERT, UPDATE, DELETE, CREATE, ALTER, DROP, GRANT, REVOKE - I pointed out that this was not a complete list of all the statements in SQL. Remember, the DML category has three statements: INSERT, UPDATE, DELETE.

For a system that support transactions, I will add three more fundamental statements to the DML category: START TRANSACTION, COMMIT, and ROLLBACK. A database transaction allows you to bundle multiple DML statements into one indivisible action in the database. Consider this example: Suppose customer 860 wants to transfer 100 units of money from their savings account to their checking account. I'm not going to present complete SQL syntax here, but I will say that this transfer requires two separate UPDATE statements. One: subtract 100 from savings. And Two: add 100 to checking. The problem is this: What if the program that initiates this transfer fails midway through the process?

There is a big problem if the first action - the subtraction from savings - completes, but the second action - the addition to checking - does not happen before the program fails. Then funds simply disappear for this customer, only because of a computer glitch! This is unacceptable in any production database system. The solution to this problem is to combine multiple DML statements together into a single atomic operation: a transaction. This list gives the gist of four SQL statements, which together form one transaction. Statement 1, begins a new transaction with the database. Statements 2 and 3, the two UPDATE statements are carried out on a "pending" basis. The COMMIT statement at the end causes the pending changes of the two UPDATES to become permanent *at the same time.* So, this sequence of statements acts as one atomic action that makes two changes in the database. Up until the time that the COMMIT statement completes, none of the data changes in the transaction is permanent. The program issuing these statements could fail for some reason - like a power outage. If this occurs, then the

database will automatically ROLL BACK, or undo, the pending updates, and the tables will be left in the same state they were in before that transaction began. If the program issuing the DML statements needs to, it can issue a ROLLBACK command explicitly, and this will order the database to discard any DML changes that the program has issued, and that are still pending. In other words, a START TRANSACTION command establishes a starting point in the database state.

Then a number of INSERT, UPDATE, and DELETE statements can be issued, and these are all kept on a pending basis in the database. Finally, a COMMIT will make all these changes permanent at one time, or a ROLLBACK will discard all the pending changes since the START TRANSACTION. A failure of the program issuing commands, or a failure of the program's connection to the database, or even a power outage in the database, will result in an automatic database ROLLBACK to the state before the transaction.

So, the bundling of multiple DML statements into one transaction allows programs to take the database from one consistent state, to another consistent state, without the risk of leaving the system in some inconsistent intermediate state should anything go wrong midway. If you use these transactional commands, it's up to you to make sure that the combination of DML statements you use in one transaction should be reasonable in taking your database from an initial correct state to another correct state.

So, don't take money from savings, then COMMIT and then put money into checking, then COMMIT again! That would defeat the purpose of having transactional capability in your database! Not all RDBMSs support transactions, but those that do will usually allow a generous number of separate DML statements to be bundled together in one transaction. So, for example, you could have a single transaction that inserts an order row, plus a set of line item rows for that order; then deducts funds from a customer deposit; then adds a bill for remaining funds due; and then alters the customer's credit rating in your system - all in one transaction.

2.13 ACID

Many database users characterize the expected capabilities of a transactional RDBMS by saying that it is "ACID compliant," or informally, that the database "has ACID capabilities" or even "does ACID." The term ACID refers to guarantees that,

combined, give users confidence that they can use the RDBMS for particular kinds of applications without worries. ACID stands for: Atomicity, Consistency, Isolation, and Durability. That's the noun form of those four words. Another way of saying it is to say an ACID-compliant transaction is atomic, consistent, isolated, and durable. Personally, I find the second set of words easier to say, but you can always impress your friends and family by using the longer noun forms in a sentence. The important thing is that if you need a transactional database application - and I'll talk more about those - then you probably want to get these guarantees from your DBMS.

I'll explain these guarantees: A transaction is "atomic" if it is indivisible, meaning that all the DML operations in one transaction are guaranteed to be made effective in the database at one time. So, a transactional system provides an atomic COMMIT, exactly as I described in the last video. "Consistency" is closely related to atomicity, with an emphasis on respecting the constraints in your database design. A transaction is "consistent" if it keeps the database in agreement with the database design constraints.

Or in other words, the transaction must never be allowed to leave the database in a state violating its constraints. For example, look at this pair of tables. Now consider this attempted transaction: The DELETE at line 3 would create a broken foreign key in the row added at line 2. In order to maintain the database constraints consistently, an ACID-compliant system will disobey these commands and roll back the entire transaction. It is true that the system "could" allow statement 2, and then refuse statement 3, but this would break the notion that these statements together comprise a transaction. A consistent transaction, in its entirety, takes the database from one consistent state to a new consistent state. Transactions are "isolated" if different transactions running concurrently in different sessions do not interact with one another in their interim changes to the database. Isolation can have different aspects, but the fundamental form is further guarantee of atomicity.

Multiple transactions currently running in a database are all pending. Then, when one transaction commits, all its changes occur in one action. When a different transaction COMMITs, all its changes take effect. So, even though the statements in different transactions may be interleaved in time, the system handles them as if they were instead performed in a series. The ordering of different transactions is determined by the time of their COMMITs. An ACID-compliant system has an internal engineering necessary to keep these interleaved activities well organized.

Finally, there's "durability," and this is an important feature for a database. A transaction is considered "durable" if, when the COMMIT statement completes successfully in your program, the database guarantees that your data changes are "persistent," or safely stored in the database.

This usually means that your changes have been stored on disk or flash memory, so that even a power failure of the DBMS will not lose your data. With a guarantee of a durable COMMIT, your programs can safely clear all memory of the data that they've put into the database, as a database system enables retrieval of that data whenever it's needed again later. ACID-compliant transactions are supported by many popular RDBMSs, like Oracle, SQL Server, PostgreSQL and MySQL. As the course progresses, you will learn about the growing variety of DBMSs that address needs other than those served by transactional systems.

Nevertheless, you can usually find details about one or more of these guarantees, even for systems that are not fully ACID-compliant. In particular, you want to pay attention to what actions can be made atomic, and when your DML changes are durable. The way your DBMS handles these features will affect the way you want to write your programs that use the database, since you know that atomic operations will never be done "part way," and that durable retention of data in the database allows your program to safely "forget" the data without it being lost.

2.14 SELECT STATEMENTS IN TRANSACTIONS

In the categories of different SQL statements, the **SELECT** statement can be classified in its own separate category of *data query language*, or DQL. Alternatively, some people group the **SELECT** command together with **INSERT**, **UPDATE**, and **DELETE** in the *data manipulation language*, or DML category. It is true that SQL provides a seamless combination of all the different kinds of statements, but there is an important way that **SELECT** statements interact with DML statements: in transactions.

Transactions let you combine multiple **INSERT**, **UPDATE**, and **DELETE** statements in a single atomic action. You can also have a **SELECT** statement participating in and informing a transaction. For example, a transaction can **UPDATE** a row, then run a **SELECT** to check the resulting provisional state of the database, then **COMMIT** or **ROLLBACK** depending on the result of the **SELECT**. The pending database change provided in the **UPDATE**—while isolated from other user sessions—is reported to

your own transaction. This ability to see changes in the database *while they are still pending* is one way that **SELECT** statements may be considered part of DML.

The emphasis here is that ACID-compliant systems let you combine **SELECT** with **INSERT**, **UPDATE**, and **DELETE** statements in a transaction. A number of the features in relational systems depend upon this combination of statements for their implementation.

2.15 ENFORCING BUSINESS RULES: CONSTRAINTS AND TRIGGERS

By now you've seen a number of ways that relational database systems define and maintain the organization of data. All these aspects of your design have direct influence on the forms of data that your database can store. Actually, you can regard a well-thought-out design as a way to enforce business rules in your overall computing system. For an operational database design, a frequent approach is to use the database system as the single, central source of truth among different programs. Then any DML statement can be treated as not just a way to store data, but also as a way to validate processes.

So, for example, an **INSERT** that attempts to add a row with a **NOT NULL** column missing, or a foreign key value that is not found in the related table, will fail. The application attempting the **INSERT** can then surface the error to the actual activity being represented. With this kind of coordination between the database system and the rest of your applications, you have a powerful method for enforcing business rules. So, your database not only helps keep the data in order; it helps keep your business processes in order as well. "Database triggers" provide a way to add even more constraint to your database organization. Not all RDBMSs support triggers, but for those that do, triggers give you a richly expressive way to enforce many more business rules.

Triggers are activities that you create and store in your database, and that automatically occur as part of DML statements. An individual trigger is an activity that occurs whenever you issue an **INSERT**, an **UPDATE**, or **DELETE** statement on a particular table. The trigger can optionally return an error status, and that will cause the triggering DML statement to fail. For example, you can write a trigger that runs whenever there is an **UPDATE** on a record in a checking account table reflecting a withdrawal of funds. The trigger can perform queries to see if the

current balance, plus the overdraft limit of the account, is sufficient to cover the withdrawal. If the amount of the withdrawal is over the limit, the trigger can return an error, and the triggering UPDATE itself will fail with an error, preventing the withdrawal from succeeding.

The ability of a trigger to check all kinds of conditions in the database and then refuse DML statements that fail to meet those conditions gives operational databases an expressive, powerful way to enforce all kinds of business rules at the database level. Enforcing business rules at the database level has the advantage that multiple programs that use the database do not need to keep repeating the same logic - - indeed, all programs must obey the rules embedded in the database they use. It stands to reason that you can add triggers to a database with a CREATE TRIGGER statement, in a category of DDL data definition. When you add triggers to a table, you are indeed doing data definition: you are defining further the kinds of data that your database will accept.

Aside from adding complex constraints to a table, another activity you can perform with a trigger is "cascading DML," in which a DML statement on one table causes other DML statements to occur along with that statement. For example, suppose you have a normalized order table, with a related line_item table. The individual items in the order are recorded in the line_item table, with the order_id as a foreign key in the line_item row. With these normalized tables, if you want to delete an order you must first delete all of its line items; then you can delete the order - - otherwise, the delete of the order would violate the foreign key constraint that "you cannot have a line item that is not part of an order." You can write a trigger, attached to the DELETE action of the order table, that will also delete all the related rows in the line_item table.

So, then if you delete an order, the trigger will automatically delete the line items along with the order. This can make good sense in your table design, on the judgment that a line item has no reason to exist except as part of an order. Another example of cascading DML is the ongoing maintenance of a summary table: whenever you INSERT a customer order, a trigger can automatically update a row in an order_totals table, incrementing the order count for that customer. The great things about triggers with cascading DML is that they can contain the DML statement you issue, and the additional DML performed by the trigger, in a single transaction, so that the database consistency is always maintained. There is another feature usually provided by systems that support triggers: these systems also let you write "stored procedures," and I want to mention those briefly here. A

stored procedure is a routine, possibly with parameters, that performs some sequence of actions in your database, and that can be called by users and programs in addition to the usual SQL commands. For example, you could have a stored procedure for transferring funds between bank accounts, that can be invoked with an amount, an account to transfer from, and an account to transfer to: Then the procedure can perform the sequence of DML statements for the transfer. Stored procedures can simplify common database activities you want to perform, by keeping a logical sequence of steps in the database, rather than requiring different programs to rewrite these routines whenever they are needed. You can see that stored procedures provide a method to take database design even further beyond data alone, into areas of application programming.

2.16 BUSINESS RULES AND ACID FOR ANALYTICS?

I've previously distinguished between operational and analytic database systems; operational databases primarily track the ongoing state of a system, and analytic databases are meant to enable complex queries that perform a deep dive into data, hopefully uncovering information that was previously unseen in the data as a whole. There are overlaps between the two functions, but large organizations are likely to have separate specialized database systems to serve operational and analytic needs. One type of operational database is an OLTP system. "OLTP" stands for Online Transaction Processing.

It shouldn't surprise you that OLTP systems rely strongly on an RDBMS's ability to support transactions and business rules. ACID-compliant transactions, along with database constraints, enable the gold standard for database support of financial applications. OLTP systems, and especially online financial systems, perform an active mix of CRUD activities, and rely on ACID-compliant transactions and the enforcement of business rules to maintain consistency and order in the database, and in business processes.

So, high-quality OLTP systems must support good performance for many concurrent transactions and queries, in a healthy mix of the two. However, not all DBMSs support ACID-compliant transactions or business rules, and not all applications need them. Consider this example: a social chat system needs the ability to store and retrieve individual messages at high volume and high speed, but

the individual storage actions can be extremely simple: all they need to do is to add individual messages to the system. In relational database terms, such systems only need a guarantee of single-row atomic INSERTs, without any need for multiple statement transactions at all.

The simplicity of the data added means that a single, scalable program can handle all the INSERTs, and it is not necessary to use computing time to enforce data rules at the database level. I'll mention such applications again in a couple of weeks when I talk about NoSQL database systems. The other broad category of database system to consider is an analytic database. The primary purpose of an analytic system is to support deep, complex queries. Many analytic systems contain data harvested from an operational system, and this harvesting happens on a periodic basis in a process called "ETL" for "extract, transform, and load" activity. For example, you may have an operational database that retains up to 24 hours of traffic information for a system that supports transportation.

On a nightly basis, an ETL program can retrieve this data from the operational database and load it into a separate analytic database. This extract and load step also often requires "transformation" because, among other things, an operational database tends to be normalized, and an analytic database tends to be denormalized. This ETL action can occur overnight, every night, until the analytic database has accumulated months' or even years' worth of data, which can serve as a rich source of deep analytic insights about changes in traffic patterns. The ETL phase may take place overnight, and then the analytic queries occur during the day, when the database state is not changing.

The alternation between ETL and queries happen on a daily cycle, or some other time period, such as hourly, but there is a separation of data load and query activity, and this is fundamentally different from the ongoing mix of CRUD in an operational system. So the addition of data to the analytic system is not a series of small DML operations, but a bulk load of data. This means the burden of maintaining consistency on the data can be shifted from the database system to the ETL program. That lets us optimize the database for complex query performance, so it does not need to support transactions at all.

Because these specialized ETL programs do all the DML, database triggers and even key constraints are usually left unsupported in production analytic databases. Having the database double-check the work of the ETL program would only slow down the ETL phase. The SELECT statements that perform analytic queries do not

engage triggers or other constraints at all, so they're not needed. The discussion I've presented here presents an important form of analytic database called a "data warehouse." A data warehouse gathers accumulated data from one data source, such as an operational database as I've described, or often from multiple sources. For example, a data warehouse can combine daily traffic data, and daily weather reports, and data in area population density, and changes in housing costs, to build a very rich set of analyses of ongoing transportation needs and transportation business opportunities.

2.17 DATABASE INDEXES

In a conventional RDBMS, indexes are special datasets that help to greatly improve performance of selected queries. Consider this illustration of a small table:

Row	ID	First Name	Last Name	Department
1	96	Ada	Brandt	Engineering
2	30	Zehra	Yavuz	Sales
3	98	Yves	Laurent	Sales
4	41	Hamza	Demir	Accounting
5	50	Sophia	Petrov	Sales
6	29	Cyril	Fedorov	Engineering

7	22	Akari	Tanaka	Accounting
8	83	Goro	Otsuka	Engineering

With such a small number of rows, you can find data from this table with ease. Suppose you want details for the person with ID 50, or you want to know how often the last name "Jones" appears in the table: you scan all the rows in the table to quickly answer your question. Now suppose that the table contains eight thousand rows: you could still answer your questions by scanning the rows, but it would take far too much time to view so much data.

A database system requires time to scan rows as well, and query time is especially affected by the time it takes to locate and read the needed data from disk storage. For a computer system, the time to read eight thousand rows may be no more than a few seconds, but the time becomes much worse with eight million or eight billion rows. Database systems often allow analysts to improve query time on large tables by implementing database indexes.

2.18 INDEXES AND QUERY PERFORMANCE

Look at this figure, which illustrates an index on the ID column of the small table above:

ID	Row
22	7
29	6
30	2
41	4

50 5

83 8

96 1

98 3

Notice that the ID column is ordered, and that each record in the index shows the row where that ID occurs. You can use the fact of ordering to quickly locate ID 50 in the index. This gives you row number 5, which you can then find quickly in the original table for details about the person with ID 50.

An index like this would allow you to quickly find the data for any person given their ID, even if the table contained eight thousand rows. For a larger table, it would take more time to locate the ID, and then additional time to find the correct row in the original table, but this two-step lookup procedure—find the row number for ID 50 in the index, then locate the table row with that row number—would take far less time than a scan of the whole table without any index.

Now look at this illustration of an index on the Last Name column:

Last Name	Ro w
Brandt	1
Demir	4
Fedorov	6
Laurent	3
Otsuka	8
Petrov	5

Tanaka 7

Yavuz 2

With this index on Last Name, you can quickly determine that the last name Jones does not appear in the table at all. Without the index, you would need to scan all the rows of the table to determine how many times the last name Jones appeared. With the index, you only need to look for where Jones would appear, alphabetically, rather than looking at every row.

Sometimes an index can be even more compact:

Department	Row(s)
Accounting	4, 7
Engineering	1, 6, 8
Sales	2, 3, 5

With this index, you can quickly find the number of people in each department, or the rows with details about those people in the original table. It's worth noting that a suitable index built from your original table can allow you to answer some questions without consulting the original table at all. In this case, a count of persons in each department is available in the index itself.

Similar to the examples above, an index in a database system is a dataset, maintained along with the table being indexed, that can greatly improve the time it takes to find some data in the table. In order for the index to help accelerate table access, a few features are required:

1. **Random access** The system must have some capability of *random access* of rows in the table. That is, the system should be able to move to some spot partway in the table and read rows from there, without having to perform a disk read from the beginning of the table. In the initial table, you see a row

number for each row. Knowing the row number, you can quickly locate and read the row of interest to you. Similarly, a database system will store some kind of *location information*, like a row number, or file block number, that the system can then use to quickly locate and read the row without scanning the entire table.

2. **Well ordered index data** In the table that indexes on ID, the ID values are ordered, and this allows you—or a computer system—to quickly locate any ID value in the list. Given any random row in the index, you can compare its ID value to the one you want and judge right away whether the entry you are seeking falls before or after your current position. This means that even with very large indexes, you can quickly locate the entry you are seeking with very few reads into the index.
3. **Alignment of index and table** Of course, if the index is to be of use to you, the records in the index must correspond exactly with the rows in the table being indexed. Otherwise, the index might cause you to attempt to read rows that are not present in the table, or worse, might lead you to miss rows that are present in the table! Questions addressed using an index that is not perfectly aligned with the original table are likely to yield incorrect answers.
4. **Indexing on the right fields** In the examples given here, the index on ID aids access to the data *when you reference records by ID*. The index on last name helps with access when referencing records by that column. However, if you ask a question about first name, there is no indexing support to improve the performance of a query on that column, unless you build another index for that purpose. Every index you define requires additional space and compute time, so it is important for you to judge which columns would warrant indexing—those you will use frequently, or in queries that you need to run quickly.

2.19 INDEXES AND KEY CONSTRAINTS

In addition to using indexes to aid query performance, many systems that have table indexing will use unique indexes as a way to enforce primary key constraints. A unique index will permit no more than one entry for its key value (the value being indexed). Because the indexed items are well organized, the system can very quickly determine whether any given value is present in the index. By creating and

maintaining such an index on the primary key column of a table, the system can quickly determine if some new row submitted for insert repeats an existing primary key, and it can reject that row right away. It is fortuitous that the primary key is also a common value to use for finding a row in a table, so the unique index on primary key does two things: it helps to enforce uniqueness on the primary key values, and it delivers speed to many queries.

Foreign keys are also often enforced with the help of indexes. A database system will often create an index on a column when that column is declared as a foreign key. Consider, for example, a **department** table, and a related **employee** table. Each record in the **employee** table has a **department_id** as a foreign key to the **department** table. If the system builds an index on **employee.department_id**, then it becomes a simple matter to enforce some foreign key constraints: if a user attempts to delete a **department** row, a quick check of the index on **employee.department_id** can determine whether this delete would leave some employees without a department, and so whether the delete should be prevented.

Foreign key constraints may work in a different way from the one above. In some pairs of related tables, such as an **order** table and a **line_item** table, the foreign key of **line_item.order_id** can be used to efficiently support a *cascading delete*. That is, whenever an order is deleted, the line items for that order have no reason for being; so the system can use the **id** of that order and the **line_item.order_id** index to quickly find the companion rows in the **line_item** table and delete them.

MAINTAINING INDEXES

Notice that every index on a table requires a dataset in addition to the original table, and that the index necessarily contains data that is a *repeat* of the table data. To make use of the index, there must always be a way to maintain consistency between the index and the indexed table—point 3, "alignment of index and table," in the numbered list above. Some systems maintain a table's indexes automatically, along with changes to the table, and some do not.

In the case of key constraints, it becomes critical that the maintenance of indexes occurs automatically and *transactionally*, at the same time as DML on the indexed tables. For example, a unique index must be checked and updated at the time of each row insert, in order to correctly aid in the enforcement of a unique constraint on primary key columns.

In addition to supporting key constraints, it can be helpful if the system provides a guarantee that indexes are always automatically synchronized with their tables. Operational databases often provide automatic synchronization of tables and indexes, and they rely on database transactional capabilities to do so.

In some systems, indexes are not updated automatically at the same time as table DML. In these systems, an index will become "stale" and inconsistent with its table whenever any DML on the table is performed. In these systems, you will perform a rebuild of all needed indexes as part of ETL: load tables with new or updated data, then rebuild indexes prior to any query activity.

To summarize, there are two ways that indexes may be maintained:

- **Transactionally, in lock-step with table DML:** This will impose some performance cost on every DML statement. In exchange, the system can support primary key and foreign key constraints and always help to accelerate queries. This is good for operational databases.
- **In deferred rebuild, separate from table DML:** This does not help support key constraints at all. To use indexes for runtime performance, you must rebuild the indexes as part of data preparation before running your queries. This approach occurs often with analytic database systems.

2.20 A FINAL NOTE ON INDEXES WITH BIG DATA

It is worth noting that the Apache Hive project has had only limited support for indexes on tables in the big data space. In Hive, indexes are supported only on specific file formats, and only with deferred rebuilds. A primary obstacle is the inability of many file systems to provide random access to data, point 1 in the numbered list above. In fact, as of Apache Hive version 3.0, the project abandoned support for indexes altogether. (Many companies are still using older versions.) Analytic systems in big data accelerate query performance with other techniques, such as extremely efficient columnar file formats, like Apache Parquet, and optimized query engines like Apache Impala. You'll learn more about these later.

When enterprises need extreme speed on queries, they may choose to invest in complete, dedicated indexing systems, using software such as Solr or Elasticsearch, both of which use Apache Lucene indexing internally.

WEEK 3

Learning Objectives

- Explain how different volumes of data affect how data is stored and processed
- Categorize a variety of data and identify processes that generate data in each category
- Describe the challenges that make RDBMSs a poor choice for big data.
- Relate the extent to which data is structured to the kinds of questions that can be answered using that data

3.1 HOW BIG IS BIG DATA

The first question some people ask about big data is: How big? The answer is a bit nuanced. Let me give a few sizes. A bit, of course, is a minimal unit of digital data, usually represented as a value 0 or 1. A byte is 8 bits, and can store an integer from 0 to 255, or a single letter, or symbol from a narrow character set. 4 bytes can store an integer in the range around plus or minus 2 billion, and the most popular character encoding is UTF-8, which takes 1 to 4 bytes per character. Here are some larger units of storage, and I could go beyond exabytes to zettabytes or yottabytes. You probably know the size of the memory and disk storage of a powerful personal computer.

The memory may be a handful of gigabytes or a little more, and the disk storage is likely to be a few terabytes or less. This machine allows you to store many emails, books, spreadsheets, photographs, applications, and more. You can readily run a relational database sized at a few gigabytes. Now, think of the size database that you would consider out of the question to maintain on this personal computer. Imagine if the computer is sized up for business use, and you can probably multiply the sizes by 10 or maybe 20. (Of course I'm giving only general figures here.) Now compare this to an analytic database I know of (running Apache Hive) that was sized at 300 petabytes in early 2014. There were larger data stores then, and there are even larger ones now. The biggest organizations today think in exabytes at the scale

of their big data. So, what is the minimum volume for big data? I hate to commit to an exact number, but I once heard a speaker say around 30 terabytes.

That sounds about right to me - - something on that order of magnitude: of tens of terabytes. At that size, things slow down and you reach the limits of what you can expect to do with conventional software on a single-computer setup. The two important concepts I'm going to talk about in the next videos are one, the volume of big data is not just more: it's different. And two, technology that handles modest volumes of data breaks down with larger volumes of data. I'll elaborate in the next video, and later in the course.

Distributed Storage Consider the storage capacity of a single disk drive - - whether it is a hard disk or a more expensive solid state drive. As I'm speaking, a single drive usually stores around a terabyte, or a handful of terabytes. Now consider if you want to store 30 terabytes, or 30 petabytes, or more! In modern technology there's 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."

For distributed storage, instead of storing a large file sequentially, you can split it into pieces, and scatter those pieces across many disks. This illustration shows a file split into pieces, sometimes called blocks, with those blocks distributed across multiple disks for storage of the file. The big data platform Apache Hadoop includes a file system called the Hadoop Distributed File System, or HDFS. In HDFS, a single block is usually of size 128 megabytes. So, a one-gigabyte file would consist of 8 blocks, and a one-terabyte file would consist of 8000 blocks.

If you study Hadoop administration, you'll learn more about the placement of blocks on different disks, but the more-or-less random scatter of blocks across disks is a common general case. Notice, though, that if one disk fails, then a part of your file is lost. This presents a problem for keeping the file system available. You can usually purchase a disk drive with a tested mean time to failure of 100,000 hours, or just over 11 years. And this gives you a high degree of confidence if you just use one drive on one computer, making periodic backup copies, for a number of years.

But, what if you have a 1000 drives ganged together, and each one is needed to keep your files available? The mean time to failure is now about 100 hours - less than a week. With 10,000 drives, you can estimate a drive failure every 10 hours. So disk failure must be an expected, common occurrence for big data systems. In

order to keep files available, HDFS keeps redundant copies of blocks on different disks. The accepted standard for redundancy in HDFS is 3 copies. If one disk fails, there are still 2 copies of each lost block available. And, the file system is self-repairing: the system notices the failure of a disk, and automatically makes additional copies of the lost blocks, distributed across the remaining available disk drives.

If you study file system and disk architectures, you may recognize the approach of HDFS as similar to a RAID 10 design. It is a fairly simple approach, that requires you to invest in raw disk storage volume that is triple the usable space you need in your file system. In other words, since HDFS stores each block 3 times, it requires, for example, 3 gigabytes of disk to store a 1 gigabyte file. A Hadoop installation on premises 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 disk storage. For these systems, some disk space must be set aside for software and temporary working space on each computer.

So, the ratio of raw disk storage to usable space in the big data store is really 4-to-1, instead of the 3-to-1 ratio I gave a minute ago. Now, how much does storage cost? As I speak, hard disks cost around 3 cents US per gigabyte, and the faster solid state disk cost around 20 cents US per gigabyte. Including the 4-to-1 ratio of raw disk space to file system space, you have these estimates. Considering the largest big data stores reached to hundreds of petabytes, you can see that an organization needs a very compelling reason to choose faster solid state drives at this scale. Clearly, there's plenty of motivation to look for space-efficient file formats and to do file compression.

Alternative file systems to HDFS are also interesting, in that they may offer trade-offs between cost, storage footprint, write times, and read times. I'll talk about cloud storage later, but I just want to point out here that storage costs at big data scale are significant. This is true regardless of the storage solution you choose, whether it's cloud storage or storage on premises in your data center, and whether or not you deploy specialized hardware storage solutions. Here's the main take-away of this video: at big data scale, there must be a way to store data in your database - and even a single database table - across multiple disks, and so a conventional file system that stores each file on a single disk is not adequate. Distribution of data across a large number of disks is the norm with big data.

3.2 DISTRIBUTED PROCESSING

Now consider the time it takes to read data from a disk. Starting with the basics, assume a common hard disk drive, with a sequential read rate of around 128 megabits per second. How long does it take to scan your data? (By "scan" I mean reading through the data end-to-end, one time.) I'll stop at 1 petabyte! You can see that at the scale of big data, the read times become huge. What if you invest in higher solid state drives at around seven times the cost? I'll give the numbers for drive that is considerably faster, delivering a sequential read rate of around 3 gigabytes per second. This is much faster - - though still far from instantaneous.

And remember that an exabyte is a 1000 times the size of a petabyte, and it would take over 10 years to read once, even at the faster rate! Fortunately, distributed storage also brings you "distributed parallel reads." If your programming is clever, you can avoid scanning your file from start to finish and instead, read different parts of the file in parallel, reading from multiple disks simultaneously. Consider, say, a ten-terabyte file.

From my previous calculations, reading this file one time end-to-end would take 22 hours from a (rather large) hard disk drive, or nearly an hour from a solid state drive. Now look at the time to read the file if you combine the total read capacity of multiple disks. This rough calculation gives times assuming that your reads have no wait time and no contention with other programs for disk usage. So, these numbers only give you a sense of the times you can expect at best, and in idealized setting. And actual times will be longer. You can see that read times have a significant impact on the time you can expect to do any kind of processing on big data, and that you must use multiple parallel reads in order to achieve good read times.

What goes with multiple parallel reads? Multiple concurrent process threads, running on multiple computers. Suppose your program needs to find all the records for a few customers, and give total net amounts for their purchases and payments. A distributed set of processes can do parallel reads, selecting just the needed data for the desired customers; then these partial results can be collected in another process for final calculation and display. If your program calculation is sufficiently complicated, then there may be an additional stage of distributed processing, in which intermediate results are reorganized by grouping or sorting then processed further, and then, there is a collection and display. This intermediate reorganization

of interim results can be called a "shuffle" of the data. At big data scale, the shuffle of data between distributed processing stages involves heavy network traffic, and may require temporary disk usage on some machines to complete properly. If the calculation on your original data is even more complex, it may require another shuffle, and another processing stage before results can be gathered and displayed.

In principle, a program may require any number of processing stages, with a shuffle between each stage. In the general case, your big data program may not produce a small amount of data for your display but instead, may read a large data set, and produce another new large data set. In this case, your program will perform not just distributed reads, but also distributed writes. With large data reads plus large data writes, your program makes even more demands on disk read/write capacity. It is no wonder that these programs are "batch programs" - - that is, real-world programs on real-world big data often take minutes, or sometimes hours or more to complete.

I've had one customer for whom one data processing program ran for two weeks! And as before, with complex calculations, your program can have one or more shuffle phases prior to the final output. With large shuffle between stages, your big data program can take even more time. Perhaps you realize by now that I've been describing a generalized big data processing framework called MapReduce. This framework was first presented in a paper from Google in December, 2004. Then in 2006, a team of engineers working on big data indexing announced a project to implement Google's ideas in open source software.

That project was named Hadoop after a yellow plush elephant toy that belong to Doug Cutting's young son. Today MapReduce is a solid production-quality framework for processing big data. HDFS and MapReduce form the initial basic components of the Hadoop platform, for storing and processing big data, and they remain in use today. Apache Spark uses the same underlying processing paradigm as MapReduce, with independent parallel reads into multiple processes, and possible shuffle into another stage of processes, and so on.

Spark improves on MapReduce in a number of ways, including making use of the greater amount of memory available in servers today, so the intermediate use of temporary disk is greatly reduced, resulting in significant performance gains. Today you can choose to store data on premises in your data center using HDFS, or Apache HBase, or some other storage approach; or in the cloud, with Amazon S3, Microsoft Azure, or some other cloud storage. Some companies use a hybrid approach, with

data generated in the company's servers stored on premises, using say, HDFS, and data generated on the Internet remaining in the cloud. You can use content from both data stores in one program. Transfer of big data volumes between cloud and "on-prem" storage is not free.

Look back at the estimated time to read big data, and note that network transfers also take time, and that cloud providers will charge a fee for a high-volume transfers. For these reasons, some users say that big data has "high mass" and "high inertia," and "doesn't want to move from where it's generated." On the other hand, it can simplify your data and programming practices to have your big data consolidated, whether that is on-prem or in cloud storage. In the end, it's your decision, your business decision, whether you want to choose on-prem, cloud storage, or a hybrid of the two. Regardless of your choice, your big data store will necessarily be distributed across many storage devices, and processing will also be distributed in nature.

3.3 STRUCTURED DATA

The volume enabled by big data storage means that you can capture the contents of your operational databases at different times, many times over. But that's far from everything you can store. Here's an important example. In the recent past, organizations would keep machine logs from their applications for a limited period of time - - maybe for 24 hours, or for a few days. These logs were then used for troubleshooting problems in the short term. Today, you can capture and store logs spanning years, so that this data becomes more than a resource for troubleshooting recent problems: it is now a new source for insight about the activities and trends in your business.

It's easy to turn on informational logging in production programs, and logs can be generated, captured, and stored without interfering with the normal operations of the application. And then, there's publicly available data. Online search engines are able to record and keep details about every search ever entered, and social media sites keep every message and comment, on every subject. News archives are digitized and current news articles are published in digital form. The number of datasets, articles, photos, audio, and videos available on the internet is only increasing. You can obtain these kinds of data for little or sometimes no cost. You

can then use this data to gain richer insights into your own data. And then, there's the Internet of Things. The era of IoT is just beginning. Vehicles, appliances, and other devices will record simple measurements, generating data worldwide at a pace at least one or two orders of magnitude higher than the total production of digital data today.

The important thing to notice is that, because big data stores let you store such high volumes of data inexpensively, you can now store many new kinds of data. So, big data has not just high volume, but also has high variety. You can classify digital data as structured, semi-structured, or unstructured. These categories do not have strict boundaries, but they do have general meanings that you can understand and apply. I'll start with a discussion of structured data. A simple definition of structured data is: data that conforms to a set schema. Look at this simple schema and table from last week.

The important thing about this table is - that without even looking at the data - you have a guarantee that every record that will ever appear in the table will always conform to the table schema, without any exceptions, ever. If the columns are NOT NULL, you have even stronger guarantees on the data. This is structured data. And since every record in a relational database is a row in a defined table, you have a guarantee of structure - - even more so if the database is normalized. The widespread use of relational databases, and the table definitions required for all relational data, yield a common notion that structured data, and data in a relational database, are equivalent. This is not universally true, but it is largely so. I will discuss ways a relational database can accommodate some unstructured data shortly.

3.4 UNSTRUCTURED DATA

Unstructured data, is plainly enough, data without clear, definite structure - - especially the structure you find in normalized relational tables, using atomic values and simple data types. Natural language text is one type of unstructured data. Look at these two records: These lines carry information about individuals' names and ages, but not in a consistent, organized form, with names and ages put into atomic fields. Suppose you have ten thousand lines of text like this, where people give their name and age in years, using informal language.

You cannot readily use this content to find the average age, or the most common name. Even this simple text presents obstacles to analysis. And the broader case of

written text in general is even more challenging to analyze. Media files are also examples of unstructured data. Of course these files have a definite format, like MP3 for audio or PNG for images, but they are not files of the simple data types you have for relational database tables. You can readily store and copy media files, and you can reproduce sound and images from them using suitable technology, but what if you want to search the data? Imagine if you have MP3s of 5,000 songs, and you want to find songs with a particular singer. MP3 files do have an optional metadata tag, with up to 30 optional characters for "artist name", but if the singer names are not found there, you cannot easily search the files.

You must first tag every file with a singer names, and then you can search the tags, and not the MP3 content itself. Media files, satellite images, medical x-rays - all these are examples of unstructured data. Relational databases do allow you to store some unstructured data. I'm thinking of the character data type. Many RDBMSs provide a character string type that will store just over 65,000 characters in one field of a row. This is more than enough for most news articles, and you can use a column for natural language in this way, then that qualifies as unstructured data. Even more extreme or the BLOB and CLOB datatypes sometimes supported, that I mentioned briefly before. A BLOB column can store up to 4 gigabytes of binary data; and a CLOB, up to 4 gigabytes of character data.

Not all RDBMSs support these data types, and those that do provide no functionality at all for searching or analyzing their contents. As far as the database is concerned, BLOB and CLOB columns are undifferentiated masses of data that can be stored and later retrieved, and that is all. When you leave out BLOBs and CLOBs, and you use text only for atomic values like names and labels, then you can equate your relational tables with structured data. But the broad range of digital media and text we have today form a great mass of unstructured data.

3.5 SEMI-STRUCTURED DATA

Some data can be characterized as "semi-structured." This is usually defined as data in which fields in a record are tagged, but there is no definite schema that all records are guaranteed to meet. Look at this set of JSON records: You can see the tags: name, pcode, age, and city in these records. However, there could be other records in the dataset, and you have no guarantee that only these tags will appear, or that field values will have consistent data types. Another form of semi-structured data is XML. This clearly indicates tags and values for different records, but again,

the XML document alone does not have a schema. Many data services on the Internet provide data records on request using a programming API.

Such records most commonly come to you as JSON or XML. Of the two, JSON is more popular, mostly because it is more compact. JSON and XML are the most common forms of semi-structured data. A CSV file with column headers is another form of data with labels but no schema. Similarly, if you create a spreadsheet, you can have headings for each column, but you have no constraint that requires you to record similar data elements consistently in a column. Informally, some people use the word semi-structured to mean that data has some structure, but without their regularity or schema of structured data. Look at this log file example from the documentation on the Apache HTTP Server: An Apache Server can be configured to emit a log entry like this for every request it receives, and the total of all such log records hitting a website is indeed a rich source of information.

The fields in this line are defined: Armed with this information, you can use character patterns like regular expressions to break out pieces of the character string into specific, atomic values, and these can be shaped into a structured record to the extent that all records conform to this format. The challenge with log files is that servers may have a variety of options in what is logged, and there is no enforcement of the content or format of logs. Also, the logging options of different kinds of servers may vary, and custom logging created by engineers can vary widely in the kinds of data recorded. So, in this second, looser definition of "semi-structured data," log files serve as a good example.

3.6 WHAT ABOUT VELOCITY?

The common characteristics of big data applications, called "the three Vs of big data," are **volume, variety, and velocity**. (Some people like to add a few more Vs in their discussions, like value and veracity.)

The velocity of data—the speed at which data is generated in modern systems—is another dimension that has grown notably in recent years with the rise of the internet, especially social media. Popular search and social media sites commonly generate billions of messages per day.

Another major increase in the velocity of data generation is underway now, with the rise of the internet of things, or IoT. Already, navigation apps on smart phones generate individual location records at high frequency. In the near future, metrics

like location, temperature, vibration, and fuel level will be reported by all kinds of transportation vehicles, and other data points will be produced by devices in manufacturing, distribution, retail, medicine, and home life.

Ingest

Big data systems provide choices for data stores that can hold great volumes and types of data, but there remains the challenge of transferring records from where they originate into your big data store—that is, the ingest of this data. Aside from challenges presented by volume and variety, the velocity of big data production presses at the limits of data transmission rates.

As with data processing of high volumes of data, systems that address high velocity ingest do so by means of multiple parallel processes. For example, using open source software like [Apache Kafka](#) and [Apache Flume](#), enterprises can build ingest pipelines composed of hundreds of data streams, transferring petabytes of data daily. These systems use multiple servers and processes to achieve high rates of data throughput between multiple points, using many concurrent transfer paths. [StreamSets](#) is open source software that lets designers create, monitor, and maintain data-transfer pipelines using a graphical user interface.

Once data records have been ingested into your big data store, you can contribute to the velocity of gaining insights by learning to write correct queries quickly, and using tools such as Apache Impala that can deliver query results with low response times. Data analysts, while not mainly concerned with the mechanics of moving data from one data store to another, certainly do care about the timeliness of analytic results. Because of the variety of storage formats supported by Apache Hive and Impala, your enterprise can move data in your big data store in nearly any format, and you can begin to perform analytic queries immediately thereafter.

3.7 STREAM ANALYTICS

Hive and Impala are designed to run against a data set that remains unchanged for the duration of your query. You need a distinctively different form of processing if you wish to process data records as they are produced.

Such a form is *stream analytics*, in which analysis is done on data streams *while the records are in motion, and prior to the time records come to rest in storage*. This involves processing the data in one of two ways. One technique is to respond to each record as a data event that can be examined, and which can then trigger some other action like an update on a graphic display or a change in a machine setting.

The other technique is to use micro-batches, in which the records accumulated over a small time interval, say each second, are gathered and processed quickly to produce a near real-time information point or action in response.

Apache Spark has a subproject focused on stream processing, called [Spark Streaming](#), and [Apache Storm](#) is another open source project devoted to stream processing. Proprietary software such as [Splunk](#) focuses on both data transfer and stream processing. Interestingly, Splunk pipelines can process data records in motion, and bring the data to rest by storing it in a Hadoop cluster. This use of different software tools in combination is common in enterprises today.

Spark Streaming, Storm, and Splunk each have their own specific programming interfaces for you to do stream analytics. Confluent, a company that supports Kafka, seeks to leverage SQL skills like the ones you will learn in this specialization: they support a product called [KSQL](#), which allows SQL-like statements to process records passing through a Kafka system.

Conclusion

The software tools you will study in this specialization—Impala and Hive—are designed for SQL-style analysis of data at rest, so the velocity of accumulating big data is of less importance here than the volume or variety. The skills you develop here are vital to any work you may do in big data analysis.

While the ingest of new records into storage is not a task of data analysis, Impala and Hive do help you address the velocity of big data in one sense: they allow you to begin querying your data immediately after it comes to rest, without large delays for preparation before you begin your analysis.

For analysis of data records while they are still in motion, you should consider stream analytics. This is a separate but related activity, and you may or may not opt to work in that area in the future.

3.8 STRENGTHS OF TRADITIONAL RDBMSs

RDBMS Strengths

8. Lots of tools and solutions
7. Fast (at reasonable scale)
6. Simple security controls
5. Strong with small or medium data
4. Many good choices
3. Structure
2. Transactions, OLTP
1. Enforcing business rules

While I'm discussing the innovations of big data, it's worthwhile to once more recognize the strengths of traditional RDBMS technologies. Last week I discussed how you can use normalized table designs and database triggers to codify complex business rules in your database. The constraints you build into your data models, together with ACID-compliant transactions and stored procedures, allow you to build high-quality financial applications and other online transaction processing or OLTP systems, with your database at the center of your total application design. The strong business constraints on allowed data in your database can exert centralized control over what your business will accept as "good data," no matter

what user interfaces or other programs you build out in your organization. The efficient enforcement of business rules, and the kinds of operational databases that this affords, represent one of the great successes of SQL and relational technology. Another strength of relational systems is that - just by storing your data in tables - you will automatically have structured data, not counting large text or binary fields. I'll discuss a bit later the kinds of analysis you can do with different kinds of data, but for now, I'll just say that a great many analytic techniques rely on data first being in a tidy, structured form.

For a data analyst, having data in structured form to begin with is a great advantage, in that you can skip over some of the effort of preparing the data, and proceed more directly to analysis. Relational technology has become so widely adopted and mature that you can find good choices for relational database software that you can easily install and run on almost any computer. These include PostgreSQL and MySQL Community Edition, which are free database servers, and SQLite, a library that less programmers embed an ACID-compliant database in their own programs without needing a separate database server. This means that - when you stick to mostly structured data and your data size is small or medium - you can readily choose an RDBMS to handle your data storage, and the software will help to keep your data organized and useful.

When your database system implements the DCL commands of GRANT and REVOKE properly, you can use these to manage security on your data: Through various applications, users connect to your database with different authenticated user IDs, and you can give different users access to different parts of the data. The RDBMS helps you manage your access in an orderly fashion. A well-designed and appropriately sized RDBMS is reasonably fast. With smaller databases of a few thousand rows and a few users, many queries can be reliably served in sub-second time and analytic queries will often require only a few seconds. Larger businesses with more users and more data will of course need to invest in bigger, more expensive solutions in order to keep up runtime performance.

There are literally thousands of programs and tools that use SQL to store and retrieve data, from visualization tools and reporting, to particular applications like manufacturing support, or sales management, or healthcare. You can take advantage of this rich ecosystem of software that has grown around relational databases, to get you to working systems reasonably quickly. It's my belief that conventional RDBMS solutions, especially at small and modest sizes, will continue to be useful for many years to come.

3.9 LIMITATIONS OF TRADITIONAL RDBMSs

Limitations of Traditional RDBMSs

RDBMS Limitations

4. Difficulty of distributed transactions
3. Weak support for unstructured data
2. High cost of storage
1. Schema on write

Now I want to talk about some limitations of traditional RDBMSs, especially when presented with the challenges of big data. I've twice made the point that relational systems have the strength that they can enforce strong constraints on your data, and so they can enforce business rules. For these operational database designs, the good news is that records that violate your constraints are rejected by the database. But, the bad news is that records that violate your constraints are rejected by the database.

In other words, your database cannot store a record that does not conform to your pre-defined table schemas. This fundamental characteristic of relational systems is commonly termed "schema on write": records that do not meet your pre-defined structure are rejected with an error code and are never stored. So, schema on write can be regarded as a strength or a weakness, depending on what you're trying to do. In the big data world, you may be presented with millions of records a day, and schema on write presents an obstacle if you want to retain them all - even the "bad" ones, or the ones for which you do not have a schema defined already. Now, you might get the idea that you can define a database table with a single BLOB or CLOB column, and then you can store any records - all records of any sort - in that table.

This is the extreme of a thoroughly non-normalized data model. And yes, you could do that, but hold that thought: I'll come back to it. A great beauty of relational systems is that they allow users to simply issue a statement like CREATE TABLE, and the software takes care of all the rest. That is, you think in terms of tables with rows and columns. The database software separates you from the lower level concerns of the file storage, and managing files with all the ongoing changes to data. But make no mistake: the data processing "under the hood," beneath the level of the SQL commands you issue, is complicated.

This abstract layer of working, with isolation from implementation details, is generally considered a strength of RDBMSs, but it does not come for free. The software that keeps your data in nice neat tables for you will incur storage and processing costs to do so. And, for many production enterprise applications, significant monetary costs as well. Adding storage, processing, software licensing, and personnel needed for support, a relational database is a more expensive type of a data store, where the total cost per terabyte of data is perhaps 10 times or even 100 times the cost of a simpler data store like a file system.

The higher cost per terabyte is justified if the data you store has high value per terabyte - meaning that you have high information content in relatively small amounts of data. If you store all your personal contacts of every type - media accounts, phone numbers, addresses, everything - you will probably have a few thousand bytes of storage total; maybe ten thousand bytes. On the other hand, a single HD video can easily be 10 gigabytes.

That's a million times more storage for one video! Now, I'll return to the idea of creating a table with a single BLOB or CLOB column. The database design provides no structure, and SQL provides almost no means for searching, or sorting, or calculating any information on your column. An approach like this is an anti-pattern: an attempted solution that creates more problems than it solves. The result of this anti-pattern is a system that is essentially not a relational database at all, and its storage cost and performance would be worse than simply storing files in a disk directory, and searching them when you need them. I hope by now you can see that the careful structuring of data, and the ease of storage and manipulating structured data, while all strengths of RDBMSs, play against you when you have large amounts of semi-structured and unstructured data.

It is often estimated that less-structured data accounts for around 80% of all data, and this data, with such higher volume and lower informational value per terabyte,

cries out for other technology than traditional relational databases. There's another difficulty I'll mention here: The problems with distributed transactions. Suppose you need to run a database transaction that affects a 100 or so rows in various database tables. But at larger scale, your changes may affect rows distributed across thousands of disks, and some redundant form of storage is required to overcome unscheduled disk outages.

So, how to provide a consistent commit of all changes in a single atomic action? You could lock all the rows or tables involved in your transaction, but this would quickly defeat the usefulness of your system serving the needs of many concurrent users. These large data stores require many disks, and many computers, with networks connecting them. Distributed systems are faced with special technical difficulties of synchronization over the distances involved. There are efforts to support atomic transactions at scale, but these efforts all involve new engineering innovations to overcome the limitations of transaction handling in traditional

RDBMSs. 3.10 SQL AND STRUCTURED DATA

SQL, being a language originally designed for working with relational databases, excels at querying structured data. Like most computer languages for data analysis, SQL has great facility with numerical data types. Look at this data. Simple queries can get answers to all the following questions: What value or values appear between 4,000 and 5,000? What are the top three values? How many different values are there? How often does each value appear? What is the sum of these values? What is the minimum value? The maximum? The mean? These kinds of questions can be addressed to any set of numeric values. You can go on to find a number of summary statistics, like standard deviation, variance, and different percentiles.

If your records have pairs of numeric values, like individual heights and weights, you can compute further statistics, like correlation of the two columns. For more sophisticated numerical analysis, you are likely to move beyond SQL to a statistical language like R, Python pandas, or MATLAB. Look at this data. This column records categorical values. Each item value is really a label for a shopping item. For this data, the questions you can ask are more restricted: Does a particular item occur? How many values are there? How many distinct values? How many occurrences of each distinct value? What value or values occur most? Notice you can sort the

words to aid lookups, but there is little meaning to questions like, what is the "minimum" word or the "maximum" word? Categorical values like this are common: product names, cities, street names... You may also consider personal names to be categorical values. Datasets often use an integer to represent a categorical value. For example, grocery stores have an integer code for each food item.

Or, you may use an integer for a customer ID, or a store ID. It can be convenient to store these categories or labels as integers, and SQL will happily perform numeric calculations on such data, but your own judgment should tell you that it is not meaningful to find the sum of all store IDs, or the average employee ID. The real power of SQL is in its ability to use the structure of an entire table or multiple tables, to combine these in different ways for analytic queries. Look at these tables. Imagine more rows of the sort you see here. With this data, you can successfully answer many questions: How many stores sell Ugg brand products? What store has the lowest price for Women's Classic Ugg boots?

What boots are available anywhere in a price range from 150 to 160? What is the inventory of Reebok sneakers across all stores? How much money in inventory is each store carrying? In this course, I refrain from going into the SQL syntax, but I can assure you that there are straightforward SQL SELECT statements that can answer these questions easily. With a little creativity, combining some initial analyses, you can dig even deeper: Is there some store that sells at a higher price than other stores in general? Or a lower price? What is the most widely distributed brand? With skillful use of SQL, you can gain clear insights into these questions from data like this. The next course in this specialization, and the courses that follow, are dedicated to helping you develop your skill with SQL to answer all these kinds of questions with structured data - especially using big data.

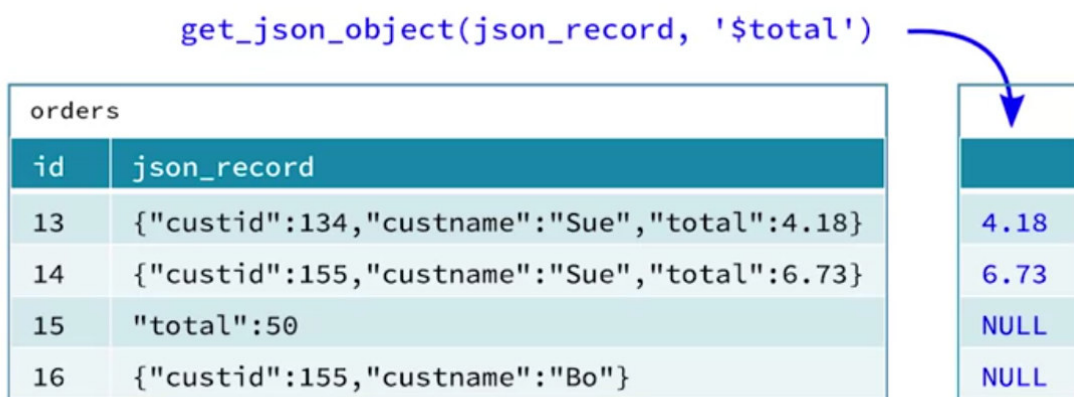
3.11 SQL AND SEMI-STRUCTURED DATA

Remember, the term "semi-structured data" has two commonly used meanings: The first of these is the stricter definition of the two: semi-structured records have their own embedded labels on fields, like JSON objects or XML documents, but no external schema that records are guaranteed to obey. Some people also accept the more relaxed second definition: semi-structured data like log files has some pattern, but no definite schema. While some may disagree that the second is a correct definition of "semi-structured data," it does describe a kind of data that you

will see in the big data world. JSON and XML are both special types of character strings, so you can have a database table with one column that is a STRING or other character datatype and containing JSON or XML. For example, look at this table: The json_record column is a STRING data type in the table schema. You may well obtain JSON records from some other data source, and then place them in a table like this.

- JSON, XML

`get_json_object(json_record, '$total')`



orders	
id	json_record
13	{"custid":134,"custname":"Sue","total":4.18}
14	{"custid":155,"custname":"Sue","total":6.73}
15	"total":50
16	{"custid":155,"custname":"Bo"}

4.18
6.73
NULL
NULL

Because the data type of the column is STRING, the table has no governance over whether the column contains JSON. Many SQL dialects provide functions for extracting parts of JSON strings. For instance, Apache Hive has a function called `get_json_object`. This function provides these results when applied to this table. In this example, the function extracts the "total" element from each JSON object in the `json_record` column.

The string in the third row is not valid JSON, and the fourth row does not contain a "total" element in the JSON object. Not finding a result, the function returns a NULL value for these records. So, the table does not enforce any schema on the JSON column, but the function can extract data from an assumed structure if it is there. There is a term for this late application of structure on data that may or may not meet the form you assume. If you've read about big data, you may know the term I have in mind already. I'll say it in another minute. There are different functions to extract possible data elements from columns containing various types of semi-structured data: JSON functions, XPath functions, and regular expressions and other string functions. Apache Hive has all these sorts of functions. Whatever SQL

dialect you use, you may want to study the list of functions available to work with semi-structured data, and such functions will generally fall into one of these three categories. In the example I've just given, the JSON is confined to a single STRING column in a table. But what if your entire dataset consists solely of JSON records?

Extracting Content from Semi-Structured Data

- JSON functions (like `get_json_object`)
- XPath functions, for XML strings
- Regular expressions and other string functions, for patterns like log records

In Course 3 of this specialization, you'll learn how to fit table definitions onto semi-structured data that originated from some other source other than a relational database. Just to repeat, here are some examples of semi-structured datasets that you can expect to have in a big data environment. Hive provides an important feature that lets you apply table definitions to datasets like this: "schema on read." This is the term I was thinking of a minute ago. This style of working with data is an important way to gain structure on your semi-structured data. Once you've added structure successfully, you can deploy all your SQL skills to perform analysis on these types of data. You will learn a great deal more about this in Course 3 of this specialization.

3.12 SQL AND UNSTRUCTURED DATA

As you know, unstructured data is of the sort that any information contained in records is not immediately, readily available. Broadly, there are two types of unstructured data: binary content, like media files; and text, like emails. Unstructured binary data is pervasive today: audio recordings, videos, and photographs are commonly in digital form, and there are also specialized binary records like seismic data or medical X-rays. Sometimes you can store binary data in your tables alongside your structured data. Consider this table: Some relational

systems, like Oracle, allow you to store a binary field using the BLOB datatype, and so you can have data like employee photos in a column like this. However, there are no SQL commands at all for analyzing the content of the photos. You can store a photo in a row, and you can retrieve the photo along with the other parts of a row. But SQL provides no means at all for sorting, or searching, or computing any results from the photos themselves. Now you can see the original meaning of the term BLOB when given as a column datatype. When Jim Starkey at Digital Equipment Corporation came up with a datatype "BLOB", he deliberately meant that featureless jelly-like mass in the monster movie "The Blob" from 1958.

It was only later that the word "blob" was tagged to mean "binary large object". From the point of view of SQL, the content has no structure and no usable data. If you want to add functionality to your SQL dialect to support a specialized binary field, you can do so using user-defined functions, or UDFs, supported by many SQL-based systems. A UDF is a function that you write in a general programming language, like C, or Java, or Python, and that you add to the database software itself. With some programming effort, you can create a set of functions to manipulate a specialized binary field in some way.

For instance, you may have a set of UDFs that manipulate parts of a field that represents genetic information. Be careful, though: this may be a poor design approach. Although it may be possible to add your own UDFs to work with custom binary content, it is likely you will play against the strengths of your database system. The power of SQL is to analyze large amounts of structured data, and the best approach is probably to keep the manipulation of binary fields in programs outside your database software. Note that many well-known digital formats include metadata tags. Digital photographs, for example, contain tags for the equipment manufacturer and exposure time of a photograph. Emails have a sender or recipient, and a timestamp.

Simple programs can extract this metadata in structured or semi-structured form, and so you can add these as separate fields to a table. Then you can use SQL to query this metadata. For example, you can find the oldest photo, or the oldest email in your database. Note, however, that you are then working with the metadata, not the unstructured content. When that metadata is not present, you may work with human analysts to add structured metadata to unstructured content. This can be prohibitively expensive for big data scale; however, some organizations like museums and libraries have used crowdsourcing to bring many individuals to the task. Today, many computing systems go beyond the analytic

techniques of SQL to find patterns in unstructured data. These programs include a variety of machine learning techniques like classifiers that perform optical character recognition to read the words in an image, or that can tag images automatically, or recognize spoken words. The important thing to note here is that such programs extract structured content from unstructured data. You can operate on unstructured data in your big data stores using machine learning algorithms; probably written in Apache Spark or a deep learning system like TensorFlow or DeepLearning4j.

Once the structured information has been obtained using these programs, this resulting structured data may be profitably analyzed using systems like Hive and Impala, for the kind of data analysis performed with ease in SQL. Besides binary data, the other form of unstructured data is natural language text. Examples of text-based unstructured data include emails, text messages, news articles - any texts where words are used, not as categorical labels like product names, but where the expressive power of the language is used to convey information between people.

Traditional SQL databases typically have some limited support for unstructured text. For instance, you can search a set of emails for the ones that contain some word. Typical relational systems do not have the benefit of extensive indexing on long-form text, and so such a search is likely to be rather slow at big data scale. Apache Hive does have a few interesting natural language functions, like the ability to find the multi-word phrases that appear most frequently in a large collection of text documents. But for the most part, you'll use separate programs for the more powerful work of natural language processing.

So, there are some similarities in the approaches you will most likely take to unstructured text and unstructured binary data: You will use some programs other than SQL databases to work with such data directly. Fortunately, big data systems permit you to store and maintain data of all sorts, bar none, and then bring a variety of tools to bear on the data. Without moving data between different systems, you can apply machine learning and natural language techniques to get structured features from your unstructured data. Then you can use those structured features in a SQL-based system like Impala or Hive to gain increased analytic insights.

WEEK 4

4.0 SQL TOOLS FOR BIG DATA ANALYSIS

Learning Objectives

- Explain the role of different types of database systems and data stores for big data applications
- Explain how dialects of SQL are beneficial for working with non-relational database systems
- Distinguish features or enhancements of SQL that are present with analytic databases on big data from those that are not
- Compare the benefits of different locations for storing big data, including how many big data systems loosely couple data and metadata

Big Database Types

- Analytic systems (data warehouses) [Impala](#), [Hive](#)
- Operational systems
 - Non-transactional, unstructured or semi-structured [HBase](#), [MongoDB](#)*
 - Non-transactional, structured [Kudu](#)
 - ACID-compliant RDBMSs [Splice Machine](#), [Trafodion](#), [Phoenix](#)
- Search [Solr](#), [Elasticsearch](#)

*Additional software can enhance these tools to support SQL, but only with structured data



4.1 OPEN SOURCE

Software products used in big data platforms are often open-source: the source code is freely available for anyone to study or use. There are several open-source software licenses, including the Apache License and the GNU General Public License (GPL).

Apache Licensing

Many of the most popular software products used in big data are released under the Apache License 2.0. Apache licensing originated with the Apache Software Foundation (ASF), a non-profit organization of software engineers world wide. Anyone can license their product using the Apache License 2.0, whether or not they are affiliated with the ASF in any way. Apache licensing makes a software product free for anyone to use, copy, and modify, without requiring any royalty payments to the license holder. (Please note, I am not a lawyer, and am not qualified to give legally binding advice here.)

Hue, the software you will use in this specialization to browse and interact with data, is an example of open-source software that is Apache licensed. Hue was developed by Cloudera, but it is available in software packages from Cloudera, Hortonworks (now merged with Cloudera), MapR, and Amazon.

Apache Projects

Beyond Apache licensing, some software products are top-level Apache projects. In addition to freely providing source code, Apache projects operate as transparent, public engineering efforts. Bug reports, enhancement requests, and status changes to these are publicly viewable. Each Apache project maintains its own public website.

An Apache contributor is someone who provides programming or documentation to a project, and anyone can become a contributor. An Apache committer is someone with the authority to incorporate a change into the main code repository of an Apache project. Apache cultivates a diversity of engineering perspectives: the committers for any project cannot all work for the same company, and decisions about the project are made by consensus among the committers. In this way, the ASF has made good on its intention to support the development of high quality software for the public good.

Some software products start in one company or team, and eventually become Apache projects. The process involves a number of steps, including these: the creator of the software donates the copyright to the ASF; the project selects a group of committers from diverse backgrounds; the group establishes an orderly consensus-based procedure for making decisions going forward; and the team builds a public website under the apache.com domain. All these practices are put into place with the supervision and assistance of the ASF. Apache Impala is an

example: Cloudera created Impala and contributed it to the ASF, and Impala became a top-level Apache project [in November 2017](#).

You can visit the home website of the ASF at apache.org. If you scroll down the page, you'll see the list of top-level Apache projects, including Hadoop, Hive, Impala, Spark, and many others.

Contribute!

Please remember that you or anyone you know can become a contributor or even a committer on an Apache project! To do so requires talent and effort, and the ASF will not pay you, but your work will be recognized and valued around the world. As your work is accepted on one or more Apache projects, you'll be helping everyone, as well as drawing the attention of the best employers in the software industry.

4.2 BIG DATA ANALYTIC DATABASES (DATA WAREHOUSES)

Big Database Types

- Analytic systems (data warehouses)
- Operational systems
 - Non-transactional, unstructured or semi-structured
 - Non-transactional, structured
 - ACID-compliant RDBMSs
- Search

-
- Such as:



In this lesson, I will survey a variety of database systems in the big data world. You are not likely to touch all these systems, at least not right away. Nevertheless, I want you to have at least an acquaintance with them, so that you will know where they stand next to the tools you will use, as a big data analyst. Then over the course of this week, I'll bring focus on to the two principle tools you will use in the hands on work.

Apache Impala and Apache Hive. Traditional relational databases, continue to be extremely useful and popular. However, with the explosion in volume and variety of big data, it became evident that these traditional systems do not serve all database needs with the best cost and performance. Alternative systems have grown in popularity in the modern era, the era of the world wide web. Among the most successful big data systems, are big analytic systems or data warehouses.

These includes Apache Impala, Apache Hive, Apache Drill and Presto. You will become skilled with Hive and Impala in the subsequent courses of this specialization. Hive and Impala each implement their own dialects of SQL. Hive QL for Hive, and Impala SQL for Impala. These dialects are similar to one another, and their uses in analytic systems emphasize a subset of SQL most suitable for data warehouses. Other large analytic database systems include Oracle data warehousing and Teradata. But these systems tend to have a much higher cost per terabyte of data than the others.

4.2 NoSQL: OPERATIONAL, UNSTRUCTURED AND SEMI-STRUCTURED

Operational, non-transactional, unstructured or semi-structured

- Good for carefully focused operational applications
- Such as:



NoSQL database systems include systems such as Apache HBase, Apache Cassandra, MongoDB, and Couchbase. Which of the following would these systems be best for?

- Large-scale transactional work with many simultaneous DML statements, such as a financial system that handles money transfers within and across records
- Focused, large-scale operations such as filtering stocks or transmitting messages through Facebook or Twitter

Correct

Correct. These systems are best for unstructured or semi-structured data with only a few, well-defined access patterns.

- A mix of analytic and operational work
- Analytic work such as querying large databases about a variety of fields within each record

Another category of operational systems are those that do not mandate a schema on your records. These include key value stores like Apache **HBase** and Apache Cassandra and document stores like MongoDB and **Couchbase**.

These systems vary in the form of data you store. Binary arrays in **HBase**, various data types in Cassandra and JSON like structures in MongoDB and Couchbase. These all belong to the category of NoSQL databases and their support for SQL is usually weak or non-existent. An important feature of these systems is that they provide simple DML and query commands and they physically organize records by

a specific lookup key. Records can be stored in massive numbers and then a record can be found rapidly by its lookup key, but not easily by other values in the record.

This is in marked contrast to the general approach you find with relational databases, where for example, you can easily find sales receipts by price, or by date, or by product, or store ID. To understand better the implications of organization by key, imagine that you must keep paper files for patients in a doctor's office over a period of years. Suppose, you alphabetize your files by patient name, then you can quickly find the file for any patient, then you can add new files in their proper place without delay.

But, if you want to find all the patients with some particular symptom, you would have a tough time of it, since their records are not organized that way. You might create a cross index by symptom, but this would require ongoing effort to maintain and even with the index, the files for any one symptom are scattered across the file set adding to your work even just to retrieve what you want. But notice, that if you can make a commitment that you will always look up files by the client name and no other way, then your simple alphabetized organization can keep things very simple and efficient for you. Similarly, **NoSQL databases perform well when you have only a few carefully defined patterns** for how you want to access records in your data. They can keep data organized to support key lookups at phenomenal scale.

Take for example, the case of an operational database for a social chat application. This app has very clear and narrowly defined access patterns, write a message for a user, broadcast messages to followers. You can have well over a million database operations per second and an even mix of reads and writes. In exchange for this kind of performance, your operational database gives up the flexibility of multiple uses. In fact, you give up the relational approach and SQL altogether. As for the non table structure of data, some developers, especially those unfamiliar with SQL, find the JSON like records of document stores, like MongoDB, to be intuitive, which contributes to their popularity.

4.3 NON-TRANSACTIONAL, STRUCTURED SYSTEMS

Non-transactional, structured

- Good for structured data, mix of analytic and simple operational uses

- Such as:



Apache Kudu

- Does
 - Enforce primary key constraints
 - Allow single DML on individual rows
- Does *not*
 - Enforce foreign key constraints
 - Allow multi-row ACID-compliant transactions

Performance of Big Data Stores

	Large analytic queries	Single-row DML statements
Impala	Excellent performance	Slower INSERT, no UPDATE or DELETE
HBase	Poor performance	Excellent performance
Kudu	Good performance	Good performance

Another type of system, **midway between NoSQL and full RDBMS systems**, would be a Non-transactional system, for structured tables. Kudu is such a system, it allows you to create tables, much like relational database tables using column definitions and common data types, like the ones you see in relational tables. Kudu can enforce primary key constraints, but not foreign Key constraints. You can perform atomic inserts, updates and deletes, on individual rows with high performance.

However, Kudu does not at this time support multi-row, acid compliant transactions. By design, **Kudu provides a compromise** in performance between the large-scale analytic databases and the large-scale NoSQL operational databases. Look at this chart, as you can see from this diagram, if you want the best performance for your large data warehouse, you will use tools built especially for this use, like Impala and Hive. You will focus on good data models and file storage

to optimize for big analytic queries. On the other hand, if you want lightning fast CRUD operations at large scale, you will need a great operational system like **HBase**.

But each of these two approaches is great at one thing, operational or analytic work and weak at the other thing. Kudu, while not the best at either thing, is good at both things. When the features and performance of Kudu meet your needs, then it simplifies your work to combine the different uses in one tool. Note that Impala is integrated with Kudu so, all the analytic Impala SQL select statements you will learn over the rest of this specialization will work unchanged with data stored in Kudu. I want to emphasize that Kudu provides tables, a SQL interface through Impala and atomic single row DML statements. But it stops short of full, multi-row, acid compliant transactions. In the Frequently Asked Questions, on the Apache Kudu website, you will find the statement that, Kudu is designed to eventually be fully ACID-compliant. But this is not the case, as I'm speaking.

For which of the following would Apache Kudu be the preferred system, over an analytic data store or a NoSQL system?

- Work that is restricted to analytic queries on large databases, accessing through a variety of fields within each record
- Large-scale transactional work with many simultaneous DML statements, such as a financial system that handles money transfers within and across records
- Work that is restricted to focused, large-scale operations such as filtering stocks or transmitting messages through Facebook or Twitter
- A mix of significant analytic work and significant operational work

Correct

Correct. Other systems may be a better choice when each type of work is the bulk of your access, but Kudu provides good performance when both are needed in a substantial manner.

4.4 BIG DATA ACID-COMPLIANT RDBMSs

There are considerable technical challenges to providing full acid compliant transactions at massive scale. Today, there are a few systems that build out their own attempts at cracking this problem. They include the Proprietary Systems,

Splice Machine, and the open source projects, Apache **Trafodion** and Apache **Phoenix**. Interestingly, all three of these solutions are based on Apache HBase.

They take the excellent performance of single row DAML in HBase, and add relational table structure and all the mechanisms needed for database consistency. Each project involves its own extensive additional software over HBase. These systems may be a good choice if you need to support massive databases, that have online transaction processing or OLTP. Like a large financial application, or a large travel scheduling system. Between the technical issues, the relative immaturity and the high expense, these systems are not so widely used as the other systems I'm discussing in this lesson. Since these systems fully implement SQL on big data, they can serve large analytic queries, such as the ones you will develop in this specialization. I think it's unlikely that you will see these systems on the job, especially not for data warehouse applications.



4.5 SEARCH ENGINES

Another very different kind of datastore is a search engine. These applications are not exactly focused on general purpose, operations or analysis. Instead, they specialize in letting you quickly search a mass of undifferentiated documents. The most well-known examples are Apache Solr and Elasticsearch. Solr Cloud refers to a clustered configuration of multiple Solr servers to achieve massive scale. Cloudera

search is Solr Cloud fitted onto HDFS storage with tools that let you index extremely large datasets found in Hadoop clusters.

Both **Solr and Elasticsearch** contain within the Apache Lucene, a high-performance indexing system that by the way was created by Doug Cutting, the co-founder of the Hadoop project. They are both especially strong with flexible lookups on text data. I have mentioned the limitations of relational systems with unstructured text data before. If you have some data set like the text of a few hundred thousand news articles, a relational database would be crippled, trying to find some particular quotation or passage. In contrast, a search engine like Cloudera search can index every word, with every word position in every article. So a look up of all the articles say with the word automobile and the word family, within five words of each other, can be found in a matter of milliseconds.

These systems can even handle misspellings and synonyms on search terms. When you open a web browser and perform a search on some phrase and get results in under a second, you are using a search engine like Solr or Elasticsearch. Interestingly, the records in a Lucene index can have a schema. Like the day, author, publisher, and URL of a news article along with the article. More recent versions of Solr and Elasticsearch had taken advantage of this structure and have added SQL as a way to access these records even though SQL is not the principle language of these systems.

Please don't worry if you find this survey of technologies overwhelming. Big data is one of the great expansions in modern computing and engineers worldwide are working to build even more useful systems in this area. In fact, most of these systems keep encroaching on one another in their attempts to add features. For example, Solr added its own SQL interface in April 2016. In this specialization, you will focus on the most well-established technologies for big data warehouses, Hive and Impala. However, your skills will be adaptable to a number of other tools in the future, not just the specific ones we use in these courses. You may notice that for the classifications here, there was a subgroup in the category of operational systems for semi and unstructured data, and another subgroup for structured data, but there were no such subgroups for analytic systems. While you can do some analysis activity on unstructured data, data warehouses depend upon structure in order to support the activity of deep analytic SQL queries. I would identify machine learning and natural language processing as disciplines for analysis of unstructured data these activities include the work of finding patterns in data that lacks structure to begin with.

What makes search engines different from operational or analytic database systems?

- They include extensive indexing and imperfect matches

Correct

Correct. The special strength of search engines is in their ability to index all the words in long text documents, and to support fuzzy matches to quickly find documents. This rapid search of free text is a specialty not found in the conventional features of relational systems or SQL.

- They include schema on their records
- Their data can be accessed by means other than SQL
- They are built on Apache HBase, adding structure for easier filtering

4.6 CHALLENGES

SQL is so well understood, and so widely supported that it's useful to adapt SQL, with the wide world of big data. Refitting SQL to support massive data warehouses, is a success story today because SQL gives you a concise and ambiguous way to request information from tables of any size. By the way, a data warehouse is a large analytic database emphasis on large. But there are challenges at larger scale, relational systems and especially acid compliant transactional databases based at least two major challenges with big data, transactions and data variety. Implementing acid compliant transactions with multiple concurrent users is non-trivial, even on a system with one computer.

If you wanted to implement a transactional system yourself, you could start by studying the book transaction processing by Jim Gray and Andreas Reuter, over a thousand pages and go from there. In a distributed environment, the issues compound with big data, your datastore spans thousands of deaths with replicated copies of data, and computers in the cluster are connected via networking with potential irregularity in transfer times. Even worse, consider the split brain scenario. Deposits switch in your network breaks, all the computers in your cluster remain active, but because of this partial network failure, different subsets of the computers lose visibility to one another completely. This situation is also called

network partitioning. It is a definite possibility in your cluster, and the system must address it in such a way that it doesn't give false results or corrupt the data by committing conflicting transactions.

Implementing multistatement transactions on distributed systems is hard, and is fraught with special problems that you must take seriously if you want an acid compliance system at large scale. The other challenge for SQL on big data, is the variety of data in big data stores. SQL is easy to use with structured data, but it is almost by definition weak with unstructured and semi-structured data. The practice of schema on write is an undesirable way to handle new content for big data, while traditional RDBMSs provide great guarantees of structure in data, their inability to store a new never before seen content, hinders their ability to retain new material for new unanticipated insight.

Though you have these two fundamental issues, difficulty of scaling transactions across a massive distributed data stores and schema on write being unable to store, never before seen datasets. Between these two issues, you had the reasons for a number of newer database technologies in the big data space. It's for these reasons, plus the expense of conventional database products that traditional RDBMS technologies have not scaled infinitely, and do not cover all the database needs for big data.



What are the two fundamental issues that prevent traditional RDBMS technologies from scaling up to the needs for big data? Check all that apply.

- Transactions are difficult to scale across large data

Correct

Correct. The necessity of distributed data storage for big data makes it difficult to ensure multi-statement transactions are ACID. Irregularities in data transfer times can cause conflicts among transactions.

storage by an RDBMS, so the schema-on-write method of RDBMSs severely limit the effectiveness of using an RDBMS for big data.

- Big data stores require a "split brain" configuration, which RDBMSs do not do

This should not be selected

Incorrect. The "split brain" refers to a scenario when a network connection is broken, which contributes to difficulties maintaining a consistent data store without conflicts. While a larger network increases the likelihood of such a scenario, this is not a "required" configuration.

- SQL would need to be refit for big data, but attempts to do so compromise the concise nature of SQL queries

4.7 WHAT WE KEEP

What We Keep

- SELECT statements, including multi-table SELECTs
- Seeing data as tables with column names
- DDL
- DCL

An assortment of software tools approach data warehouses at large-scale. But going forward, I will focus on the tools you use in this specialization. Apache Hive and Apache Impala. These open source tools are well established in the marketplace.

Hive was contributed to the Apache Software Foundation in 2008 by Facebook, when Jeff Hammerbacher was head of data there, and before he went on to help co-found Cloudera. **Impala** was initially developed at Cloudera, and has been generally available since May, 2013 and became a top-level Apache project in November, 2017. By now, you have a conceptual foundation in SQL as it was originally intended, and as it is still used as the principal language of conventional relational database systems. A primary strength of SQL is the straightforward way unless you express a wide variety of analytic queries on data using SELECT statements.

Indeed the expressive power of SELECT statements to answer so many questions from your data is the one feature you can reasonably expect from any system that claims at least partial support for some dialect of SQL. Some big data systems offer only very limited support, for example they **might not allow multi-table selects**. Other big data systems have much broader support for SELECT statements. Hive and Impala are examples of this. They support a wide variety of different SELECT statements, even more than some RDBMS's support. In order to write a SELECT statement, you must be able to view your data as residing in tables with named columns and of course Hive and Impala keep that ability. It's reasonable to let you manage table definitions with some dialect of data-definition or DDL that create, alter and drop statements. There are special variations of the create table

statement in Hive and Impala, but the point is, this functionality is available in Hive and Impala. For managing different users access to data, **Hive and Impala also support SQL grant and revoke statements**. They're familiar data control or DCL statements in SQL

What We Keep

- SELECT statements, including multi-table SELECTs
- Seeing data as tables with column names
- DDL
- DCL

4.8 WHAT WE GIVE UP

What We Give Up

- Unique columns
- Primary key constraints
- Foreign key constraints
- Synchronized indexes
- Triggers and stored procedures
- UPDATE and DELETE statements

A data warehouse is created by gathering data from one, or usually multiple data stores into one place for reporting and analytics. A great strength of big data platforms today is that they allow you to manage and use data stores of much larger sizes with a lower cost per terabyte than conventional relational systems. But these **big data systems give up transactions**, which combine **multiple insert, update, delete and select statements in a single atomic action**. A number of the features in a relational system, depend on this combination of statements for their implementation, though the loss of transactions has several implications.

In order to enforce uniqueness in a column, a conventional relational database system checks all rows in the column for any new row you try to add, and then permits the new row only if its value for the column does not occur in the column already. The check for existing values and the addition of a new value must be bound into a single transaction in order for this to be done correctly. Uniqueness is an important feature of a primary key. So, you can see that transactions enable a database to enforce primary key constraints on your database tables.

Similarly, a database system uses transactional lookups on related tables in order to enforce foreign key constraints. Without transactions, you might have tables with unique key columns, and you might have consistent foreign key relationships between your tables, but the database technology does not guarantee this consistency in your data. You cannot assume that rows are unique or that all foreign keys are correct. It's up to you to know about this, and to keep your data organized in whatever way you require.

Transactions are used by conventional relational databases to synchronize indexes with tables. Maintaining the entries in your index in lockstep with the DML you perform on your table. Without transactions, indexes will not be synchronized with your table automatically, and you would need to rebuild your indexes whenever you need them to be up to date. Database triggers also depend on transactions. Business rules and triggers and cascading DML statements and triggers both require the transactional ability to atomically combine multiple changes and queries in order to maintain database consistency.

Although you could conceive of some store procedures that perform only a single action in a database or that perform multiple actions but not atomically, these would give you a radically smaller subset of the store procedures you'd want to have in order to build up application logic in your database. Though as a practical

matter, effective stored procedure programming also requires transactions. **Without transactions, database triggers and stored procedures are not possible.**

Because of the difficulty of synchronization over multiple distributed copies of file data, dump file systems like HDFS, the Hadoop Distributed File System, **lack the ability to update file content in place.** Following from this limitation, big data warehouse systems give up SQL update, and delete statements that can change values on individual rows or delete individual rows. The workaround is to rebuild tables completely including desired changes in a batch process. As a side note, I want to mention that when you use Impala on Apache Kudu as your data store, you can manipulate individual rows with UPDATE and DELETE statements. It is this capability that makes Kudu a useful new system or a combination of operational and analytic work. But without this special feature of a storage system like Kudu, individual UPDATE and DELETE statements are not possible.

4.9 WHAT WE ADD

What We Add

- Table partitions and bucketing
- Support for many file formats
- Complex data types
- (Some detailed differences)

A feature that was added to SQL for querying bigger tables, is **table partitions**. This feature is actually not limited to big data systems. Table partitioning is a well-established feature of virtually every RDBMS that seeks to aid performance of queries as tables grow larger. By the way, writers and speakers often use the word partitioning alone to refer to either table partitioning or network partitioning.

These are completely different concepts and you need to pay attention to context to know which of these is being discussed. The idea of table partitioning is simple.

You take a bigger table and the system lets you specify some simple logic, the store's rows in physically separate file directories or partitions according to some value found in the rows. For example, a table of worldwide retail sales can store rows in a separate partition for each country. Now if you run a select statement to find say, the total sales figures in the year 2015 for Australia, the system would disregard all rows except those found in the partition for country Australia.

This quick dismissal of unneeded partitions from your query **is called partition pruning**. It means that the system needs to further process only a fraction of the data and the query result is assured to be nevertheless correct. This is especially helpful for query performance on large tables. Notice that partitioning on country does not help your query at all if the query does not include country in the qualifications for data you want to analyze. So, whether and how to use partitioning is a matter for you to decide based on the queries you plan to run most often on your large tables.

Table bucketing is similar to table partitioning. **It subdivides the rows in your large table into separate areas of storage** but usually in a random or pseudorandom way rather than a straightforward predictable way. Bucketing can help performance when you want an analysis based on an arbitrary sample of your table and not all the data.

Big data stores also support a variety of file formats, re-structured and semi-structured data. One simple file format is a CSV file, where records are stored as text and fields are separated by commas. Fields can be delimited by tabs or some other character as well. Other text-based format supported include **XML and JSON**.

A completely different format developed for use with big data is defined by the **Apache Avro** project. **Avro defines a binary file format for saving structured data to disk**. Binary files are not character-based and so there's no way to show you a useful example here. Avro files require far less storage space than a text-based format **especially when you have lots of numeric values in your rows**. But there isn't just one binary file format. Another format is **Apache Parquet**. Parquet files like Avro files, store structured records in a binary format but it's a different format from Avro and it can be even more space efficient than Avro for many types of tables.

Files can be compressed for even more space savings and encrypted for data security if needed. Your big data store can have different files and any mix of these formats and any other formats you can find useful or that may be developed in the future. So, you have no lock into one vendor or one kind of file.

Another major addition to SQL, is a set of additional data types called **complex data types**. One principle of normalized table design in most relational databases is to keep all your column types atomic. In other words, each field of a row should store exactly one thing. Here is a partial list of the datatypes supported in Impala. The datatypes in hive are similar with a few more on the list. The thing to note about these data types is that all of them with the exception of the character types, force you into using an atomic value in each part of a row.

The character types allow lists of words or unstructured text but that is an exception and good design dictates that you should be aware of this and plan your tables carefully, whether you keep your character columns atomic or not. The additional complex types in Impala, also supported in hive are these. The **array data type** let's you put multiple values of one tie into a column. For example, with the array type, the movie table might look like this. In this example, the actor's column is an array of names and the show time's column is an array of time.

A normalized design for this data might look like this. In these normalized tables, the primary key for actor and movie and movie show time is a composite containing both columns of the table. The table design with the array columns is a deliberately denormalized design. In fact array columns are examples of repeating groups in the design and with repeating groups, the title is not even in first normal form.

A map column is also a repeating group design except that each item in the column is a key value pair. Look at this table. The map data type for the phone's column allows the table to conveniently store any number of keyed phone numbers for one customer in a single row. One potential benefit of these complex data types is that they allow your selects to easily fetch all the contents from one table at runtime rather than having to compute results from multiple tables and this can result in improved query time.

The **struct data type**, also stores multiple values in one column and you will learn more about its use with hands-on practice in a later course in this specialization. I've introduced only the core basic statements of SQL throughout this course. There are a number of finer details that are specific to the SQL dialects used by Hive and

Impala and it is like that with any SQL dialect. You will learn these details as you practice Hive SQL and Impala SQL in the other courses of this specialization.

- Apache Impala Data Types
 - INT, TINYINT, SMALLINT, BIGINT
 - FLOAT, DOUBLE, REAL
 - DECIMAL (*precision, scale*)
 - STRING, CHAR(*length, up to 255*), VARCHAR(*length, up to 65,535*)
 - TIMESTAMP
 - BOOLEAN
 - ARRAY
 - MAP
 - STRUCT
- } **Complex Data Types**



Which of the following is a difference between *table bucketing* and *table partitioning*?

- Table bucketing provides a method for identifying rows with numerical data that falls into particular ranges, while table partitioning provides a method for identifying rows with common categorical data
- Table bucketing is good when analysis often filters by a particular column or columns, while table partitioning is good for analysis based on a sample
- Table bucketing allows you to work with only a small portion of a table, while table partitioning only controls how the table is divided into blocks for storage across a distributed system
- Table bucketing divides a table in an essentially non-predictable way, while table partitioning uses a predictable method to divide a table

Correct

Correct. Partitioning uses simple logic to separate rows, while bucketing uses a random or pseudo-random method. This makes the buckets samples of the data with rows that are not chosen by a common value or values, while the partitions

What advantage does the use of complex data types provide for big data analysis?

- They make some data use storage more efficiently
- They allow you to normalize big data, including semi-structured or unstructured data
- They provide deliberate denormalization, which can improve query time by containing all data in a single table

Correct

Correct. Complex data types allow all the information to be stored in a single row in a single table, so results do not have to be computed or correlated from multiple row or tables.

- They are the only way to access unstructured data in a big data system

4.10 WHERE TO STORE BIG DATA

Places to Store Data

- On premises
- In cloud
- Both (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, and keep in mind is not just that you want to store your data, you also want to analyze it to gain new insight.

Though you really have related decisions to make about the storage and processing of 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**. By the way, you might read about on-premise storage, this is a mistake in word usage. If you look up the words in a dictionary, you will see that premises and premise are two different words with different meanings, and **on-premises** is the right term in this case.

The first enterprises to store large volumes of data stored it on-prem because there was no other choice. They provisioned hundreds or thousands of servers with a dedicated network to join the servers together into a cluster, thus cluster configurations remain useful today. And for a typical set-up, each server in the cluster contributes both data storage with a set of disk drives and processing capacity with CPUs and random access memory. Within on-premise cluster like this, you can increase storage and computing capacity together by adding more servers, **also called node or host to the cluster**.

Today, you can store your data just as well using cloud services such as Amazon Web Services, Microsoft Azure, or Google Cloud Platform. 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 on-premises and some data in the cloud.

Cloud storage is attractive for many cases because it lets you start quickly, start small and easily increase storage as your data grows. With the separation of storage and computing in the cloud, you can support occasional processing needs with transient compute clusters that are elastic, meaning that they are sized on the fly to support the data processing involved and the response time you need at the time. This is especially attractive for **intermittent or bursty workloads**. You can even have different teams that access the same shared storage with different optimally-sized compute clusters. **Cloud providers let you pay for storage and processing only as you use it**. For cluster on-premises, you can start with a test

cluster of a few virtual machines or repurpose servers, but an enterprise grade cluster will usually involve dedicated hardware, needing major investment in real estate, equipment and personnel, and companies usually need long review times to make these investment decisions. Think of it this way. Suppose you need a means of transport like a car, or a truck, or a scooter, would you rather rent or buy?

If you plan to use the vehicle only one or two times a month, it is cheaper and probably more convenient for you to rent. But if you need the vehicle nearly every day for several years, then you most likely want to buy. Even though this means a higher initial cost and the extra effort of keeping and maintaining your vehicle, owning will cost you far less than renting in the long run. Of course, you could choose to rent even with constant use. This would let you change your type or style of vehicle with very little trouble, but you would pay a premium cost for this flexibility. It is similar with cloud or on-prem clusters.

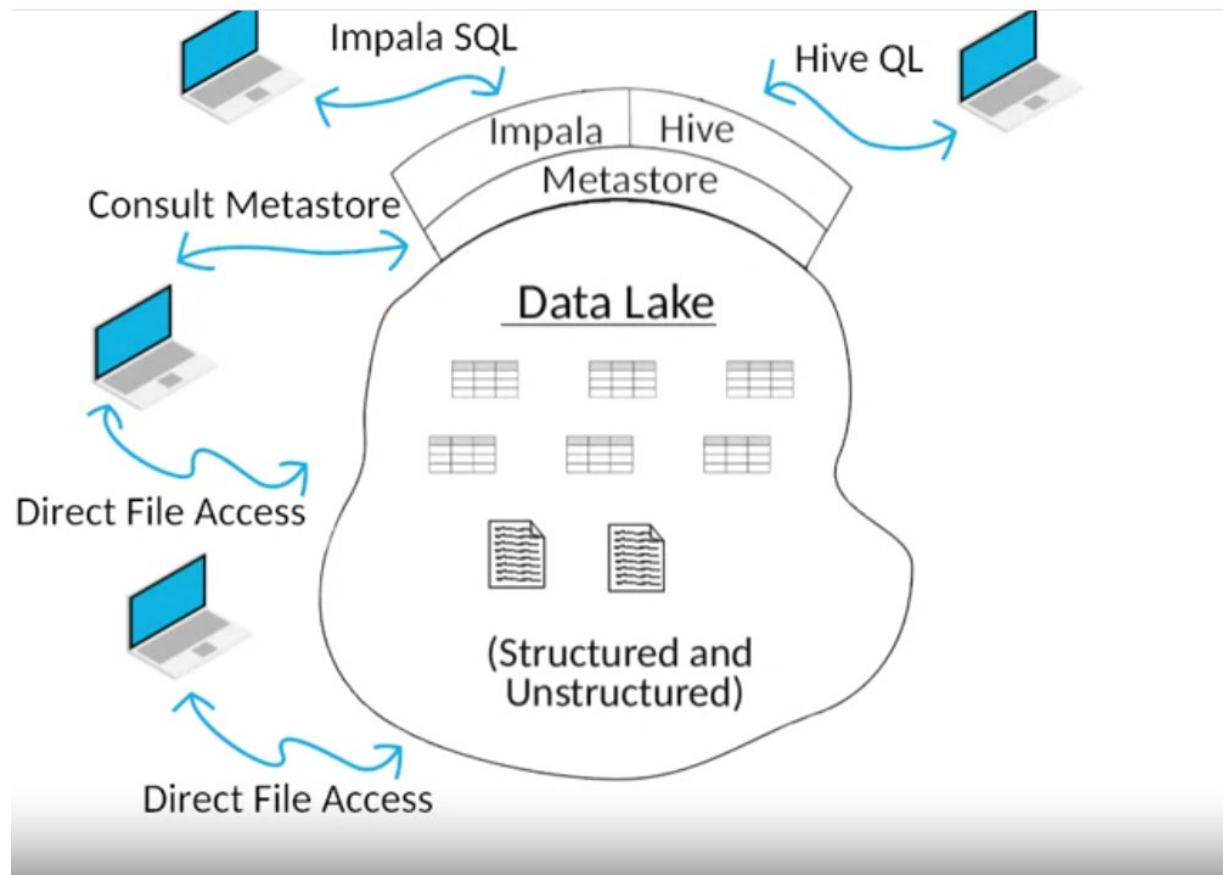
In the cloud, you rent storage and computing power from providers who take care of hardware and a mass of other details for you. They give you flexibility and convenience at a cost that makes the most sense for light usage. On the other hand, a data warehouse that will be used for more or two or more than two or three years with many users with ongoing analytic workloads can be more economical overall if your organization chooses to host it on-prem.

However, many organizations have approval and accounting processes that make it difficult to build or expand an on-prem data center. Datacenter investments are considered capital expenditures, whereas cloud service costs are typically considered operating expenditures. Capital expenditures are long-term investments, though they're subject to a lot of scrutiny and accountants are required to amortize the costs over multiple years. Operating expenditures are short-term decisions with less risk though they're usually subject to less scrutiny and fewer accounting requirements.

For this reason, many companies use cloud services even when the costs are higher. Midsize and larger enterprises may well take a hybrid approach to storage and computing because they have data of very different sorts, some generated online and some in-house, as well as a mix of transient and sustain compute needs. After all you may want to own a scooter, but rent a truck on occasion to do some hauling. With evolving offerings in hardware, software, and service providers, you can expect to see a growing variety of choices including new hybrid choices in the

future. Unfortunately, analytic tools like Apache Hive and Apache Impala work equally well with data on-premises or in the cloud.

4.11 COUPLING OF DATA AND METADATA



In a conventional RDBMS, when you issue a create table statement to create a new table in a database, the system handles all the implementation details. The system checks any foreign key constraints to make sure that they are legal. Sets up files needed to store the table, adds index files for any unique column constraints, and then records all these details about your table, in a special part of the database called the data dictionary.

Now, you can proceed to insert rows in your table, and the database system stores content in all the right files, to durably store your data and then uses those files to handle your subsequent select statements reliably, and correctly. It's great that the system lets you think in SQL, without having to worry about any of the lower level details. However, whether you like it or not, the system forces you to always think in SQL. 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.

Even with access, files are usually of a proprietary format that is not usable, except through the database software. The database system keeps the data dictionary, which is a set of internally maintain tables about your tables. With table names, column names, and properties, constraint definitions, and so forth. **The data dictionary tables, record your table definitions.**

So they comprise, data about your data, or **metadata**. The data dictionary is tightly coupled with your tables. It is always kept in exact alignment, accurately describing the tables you create. If you drop a table, the rows regarding that table in the data dictionary tables are automatically deleted. If you alter a table, the pertinent rows in the data dictionary are updated, and so forth. Through the mechanism, other relational database software, together with the encapsulation of file storage, and the data dictionary tightly coupled to data, RDBMSs provide SQL as the only way to access data in their databases.

In contrast, the data stores and big data systems, can have SQL as one way to access data, with other forms of access also available. The data store in your big data system can be called a **data lake**, or **data reservoir**, or **enterprise data hub**. All of these are terms you may see often. The data lake can retain large varieties of data of all sorts. Some contents may be structured, and so easily usable by a SQL engine like Hive or Impala, and some not structure. While the traditional RDBMS only supports 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. Some

Coupling of Data and Metadata

What does it mean that a RDBMS's data dictionary is tightly coupled with the tables?

- The contents of the data dictionary accurately describe every table in the database

Correct

Correct. The descriptions in the data dictionary govern the data in the tables, so the data in the tables must be only what is described by the data dictionary.

- The contents of the database are compressed for optimal storage space
- The data dictionary allows only one table to exist at a time
- Each table has its own data dictionary, so they come in pairs

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 or 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.

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 lockstep with all your data. In fact, some files may reside in the data lake without any information about them in the metastore at all. 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.

This lesser data lake accept files of any sort, and you can still analyze your file contents with the SQL engine like Impala or Hive, using SQL statements like those familiar to so many other systems. 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.

What does it mean that a big data system's table definitions are loosely coupled to files?

- The table definitions provide a way for non-SQL access to the files
- The table definitions describe what is expected in some files, but even those files may not match exactly

Correct

Correct. The table definitions provide a structure, but there is no guarantee that any of the files will follow that structure. In addition, some files may not be associated with any tables at all.

- Only Apache Hive and Apache Impala are allowed to access the files using SQL.
- The data files are governed by the table definitions in the Hive metastore rather than in the data dictionary

WEEK 5

Learning Objectives

- Install a working environment that can be used for hands-on exercises with big data
- List and describe existing databases, tables, and data in a big data system
- (Honors) Answer questions about Cloudera's Data Analyst certification using the web page

To use the hands-on environment for this course, you need to download and install a virtual machine (supplied by Cloudera) and the software on which to run it. Before

continuing, be sure that you have access to a computer that meets the following hardware and software requirements.

5.1 HARDWARE AND SOFTWARE REQUIREMENTS

- **Windows, macOS, or Linux** operating system (iPads and Android tablets will *not* work)
- **64-bit** operating system (32-bit operating systems will *not* work)
- **8 GB RAM** or more
- **25GB free disk space** or more
- **Intel VT-x or AMD-V** virtualization support enabled (on Mac computers with Intel processors, this is always enabled; on Windows and Linux computers, you might need to enable it in the BIOS)
- *For Windows XP computers only:* You must have an unzip utility such as **7-Zip** or **WinZip** installed (Windows XP's built-in unzip utility will *not* work)

5.2 INSTALLING THE ENVIRONMENT

Installing the hands-on environment has three major steps: First, download and install the software that runs the virtual machine (VM)—there are two choices, as noted below. Then download the VM that you will be using. Finally, install the VM. You can then adjust your settings as needed.

The VM can run using products from **VMware** or Oracle's **VirtualBox**. For Mac systems, VirtualBox is free but VMware requires a license; for other systems, free versions are available for both. The VM tends to run a bit faster on VMware, so we recommend choosing VMware if you can.

NOTE: *These instructions appear in each of the courses in this specialization. If you have already installed the VM for a previous course, you do not need to reinstall. If something goes wrong with your VM, you can delete it and reinstall, but any work you have done inside the VM will be lost.*

5.3 A. DOWNLOAD AND INSTALL THE SOFTWARE

Choose the software you wish to use, VMware or VirtualBox, and follow the appropriate instructions.

VMware

For Windows and Linux systems, download the [VMware Workstation Player](#). For Mac systems, download [VMware Fusion](#). Open the downloaded file and following the instructions provided, but please ***decline all updates and upgrades***.

For Windows and Linux:

- Decline the upgrade to the Pro version. The free version is sufficient for this course and this specialization.
- Decline to enter a license key. The license allows you to use VMware Player for free without a license key as long as it's for non-commercial purposes.

VirtualBox

On the [VirtualBox downloads page](#), click to download the platform package for the system you are using. Open the downloaded file and follow the instructions provided, but please note: The installer might prompt you to accept that the network may be disconnected during the installation, and to install USB device software. Accept both of these prompts.

B. Download the VM

We have created a VM specially for this course. You need to download this VM by clicking one of the following links, depending on which software you are using (VMware or VirtualBox). These two VMs themselves are identical, and they include the data and the applications you will need for this course. The only difference between them is which software they run in (VMware or VirtualBox).

NOTE: *The VM is large and might take a long time to download.*

- [VirtualBox VM](#)
- [VMware VM](#)

Open the downloaded file to unzip it. You will now have a folder with several files within it.

We recommend that you also **keep the downloaded zip file** in case you need to reinstall a fresh copy of the VM. You can also come back here and download it again if needed.

C. Install and Start the VM

Follow the appropriate instructions below depending on which software you chose to use.

VMware

To start the VM in VMware:

1. Open the VMware software.
2. In the **File** menu, click **Open**. (You might need to click **Player** to see to the File menu.)
3. Navigate to the directory where you unzipped the VM file.
4. Select the file named **Cloudera-Training-CourseraDataAnalyst-VM-cdh5.13.3b.vmx** and click **Open**.
5. The VM will appear in the list of virtual machines in the VMware window. (A new window might also pop displaying the VM.)
6. With the VM selected (or in the window that popped up), click the **Play** button to start the VM.
7. When the VM starts up, VMware might display prompts to update VMware Tools. Click **No** or **Never** to decline these updates.
8. Move the VM window and resize it by dragging the corners as needed.

VirtualBox

To install and start the VM in VirtualBox:

1. Open the VirtualBox software.
2. In the **File** menu, click **Import Appliance...**
3. Under **Appliance to import**, click the folder icon and navigate to the directory where you unzipped the VM file.
4. Select the file named **Cloudera-Training-CourseraDataAnalyst-VM-cdh5.13.3.ovf** and click **Open**.
5. If you see a **Continue** button, click it.
6. Under **Appliance settings**, do not make any changes.

7. Click **Import**.
8. Wait for the import process to finish.
9. The VM will appear in the left panel of the VirtualBox window. Double click it to start, then give it a few minutes to fully start.
10. When the VM starts up, VirtualBox will display messages about keyboard capture and mouse pointer capture. Click **x** to dismiss these messages.
11. Move the VM window and resize it by dragging the corners as needed.
12. In the **View** menu, adjust the scale factor to make the VM display as desired. For example, if you are using a Mac with a Retina screen, select **View > Virtual Screen 1 > Resize to 200% (autoscaled output)**.

D. Connect the VM to the Internet

First, ensure that your computer is connected to the internet. Then, in the upper right corner of the VM, look for an icon like this:



If the icon appears just like in this image above, then the VM should already be connected to the internet.

But if the icon appears with a red **x**, then the VM is not yet connected to the internet:



If you see this icon with the red **x**, click the icon and choose **Auto Ethernet** to connect to the internet.

Changing Settings on the VM

You can change the settings inside your VM to your liking using **System > Preferences**.

A common setting to change is the keyboard. The default keyboard layout is English (US). If you use a different keyboard, you will need to set it in the VM, even if it is already set on your computer. Use **System > Preferences > Keyboard** to add another keyboard layout and set it as the default. After setting the keyboard to another default, you can remove the English (US) keyboard, if you like.

Logging into Hue

Throughout this course, you will use a browser-based interface called Hue. Hue is installed in the VM; although it's accessed using a web browser, you don't need to be connected to the internet to use most commands on Hue.

To log into Hue:

1. In the VM, open Firefox by clicking the **Firefox Web Browser** icon in the menu bar..
2. Click the Hue bookmark or go to **<http://localhost:8888/hue/home>**. (For this training, you *must* use the browser within the VM. Do not try to use the browser outside the VM on your computer.)
3. Sign in using username **training** and password **training**. You can agree to remembering your username and password if you like. You also can go through the tour of Hue 4.0 if you like, but these courses introduce all the parts of Hue that you will need, when you need them, so you can click the **X** and ignore the tour.

Troubleshooting the VM

As you work with the VM, you might from time to time come up against some odd issues. Please consult this document as needed to help you troubleshoot and resolve the issues.

If you have worked through all the suggestions here and still have trouble, please reach out to your fellow students or the instructors through the forums for this course. Help your fellow students as you can, but your instructors will do their best to help you as soon as possible.

Please try to resolve the issues on your own first. We understand how frustrating it can be, but you'll learn more if you try it on your own!

Note: Many issues can be solved by restarting the VM, so this should be the first thing you try. Restart the machine by clicking **System > Shut Down** from the menu bar, then click **Restart**. (Do not just pause or suspend the VM or quit the software running it.) Many times the issue is a service (such as Hive, Impala, Hue, HDFS, or

one of the underlying services those rely on) going down, and restarting the machine will restart all these services.

VM CPU and RAM Requirements

The VM for this class is designed to use one processor core (CPU core) and 4GB RAM. Reducing the amount of RAM to below 4GB is not recommended and is likely to cause failures when running some queries on large tables. Increasing the amount of RAM to some amount greater than 4GB is unnecessary, but it will not cause problems so long as your computer has sufficient RAM to allow it. You should always leave at least about 4GB available for the operating system outside the VM to use. For example, if your computer has 8GB total ram, you should never configure the VM to use more than about 4GB. If your computer has 16GB total RAM, you should never configure the VM to use more than about 12GB.

However, you should *not* increase the number of processor cores (CPU cores) used by the VM. If you do increase the number of processor cores used by the VM, then it is absolutely necessary to also increase the amount of RAM. For example, with two processor cores, you should use at least 6GB RAM. Increasing the number of processor cores without also increasing the amount of RAM is likely to cause failures.

VM Is Slow

If the VM is running slowly, it might be that you are using too many resources for the memory available to the VM. If you're using Hue, first try closing the browser and reopening it. This sometimes clears out the resources.

If that doesn't help, then in the VM, go to **System > About this Computer > Resources** to see how much CPU and memory (RAM) is being used.

If your RAM usage is high, close all applications and browser windows or tabs except the one you're using. Avoid having Hue open in multiple browser windows or tabs, because this can use a lot of RAM.

Services Not Available

Occasionally you might find that a service or process on the VM has failed and needs restarting. The simplest way to do this is to restart the VM. See the note above the table of contents, above.

Errors in Hue

Clicking around in certain parts of Hue that are not part of the exercises might result in error messages. The Job Browser is one such example. Clicking it might show a red popup layer in the browser with an error message similar to this:

HTTPConnectionPool(host='localhost', port=11000): Max retries exceeded

with url: /oozie/v1/jobs?len=100&doAs=training&filter=user%3Dtraining%3Bstartcreatedtime%3D-7d&user.name=hue&offset=1&timezone=America%2FLos_Angeles&jobtype=wf (Caused by NewConnectionError('&requests.packages.urllib3.connection.HTTPConnection object at 0x7f4f1c53f090>: Failed to establish a new connection: [Errno 113] No route to host',))

This occurs because the Job Browser depends on a component called Oozie, which we do not include in the VM. Oozie is not used in any exercises for this course; installing it would make the VM larger and require more memory to run, which would reduce performance.

Note that the job browser is not the only place where errors like this might occur. Other areas of Hue might yield errors related to other components that have not been installed. In general, we have tested that the exercises related to Hue work without error. If you deviate from the exercise instructions, then you might encounter errors such as the one described above.

5.4 DIFFICULTY CONNECTING TO BEELINE

Your first step when a command doesn't work as expected should always be to check *carefully* for typos! Be sure the command you are using to start Beeline is

beeline -u jdbc:hive2://localhost:10000

It's very easy to type **jbdc** instead of **jdbc**, for example, and easy to overlook that typo.

It's also easy to use the wrong number of 0s at the end. (There should be four 0s.)

5.5 DIFFICULTY CONNECTING TO IMPALA SHELL

Your first step when a command doesn't work as expected should always be to check *carefully* for typos! Be sure the command you are using to start Impala Shell is

impala-shell

It's very easy to forget to use a dash and instead use a space, for example.

5.6 DIFFICULTY CONNECTING FOR S3 OR OTHER INTERNET SERVICES

Some commands (such as any commands that interact with S3, the cloud service we're using for these courses) require that the VM itself be connected to the internet. In the upper right corner, find one of these icons to determine the connection state and what you should do:



The VM is connected; if there is a problem, check your computer's connection rather than the VM's connection.



(Animated) The VM is trying to connect; give it a moment until it resolves to one of the other two icons.



The VM is disconnected; click the icon and choose **Auto Ethernet** to reconnect. If the problem persists even after you reconnect the network, then restart the VM.

5.7 APACHE HIVE

I've heard it said that HiveQL, the SQL dialect of Apache Hive is not really SQL, but is MapReduce for people who know SQL. The statement is a little harsh and provocative, but it does make a point. Hive was first developed for clusters that at the time primarily ran the core components of Apache Hadoop, HDFS for files and MapReduce for data processing.

Here's a quick review. MapReduce programs read and process data using multiple distributed tasks that run in parallel across minicomputers. A MapReduce program can, if needed shuffle intermediate results into some new grouping for another stage of distributed tasks. This illustration shows three distributive processing stages with two shuffles. But a Map Reduce program can contain one stage or any number of stages, which shuffles interpose between the stages. Finally, a Map Reduce program may write a potentially massive result back to storage, using multiple parallel write operations as shown here.

Or, it may deliver a modest result to a display for you to examine. Hive adds a meta store describing some groups of files stored in the cluster as tables and describing files in the file records as table columns. With this enhancement to the cluster data with table definitions, analysts familiar with SQL can write select statements.

Hive automatically translates a select into a suitable MapReduce program. Hive can translate any SQL select statement you can write into MapReduce. Using Hive, you cannot process all possible data, but you can process all data that has suitable structure. You cannot write all analytic programs, just those that can be expressed in SQL. However, that capability, SQL statements against big data with structure enables analysts to **cover a great portion of all big data processing needs.**

At one time, Facebook even publicly stated that fully 80 percent of all their big data processing jobs were Hive queries. From its beginning, **Hive has translated SQL statements into MapReduce programs.** Since late 2015, Hive has had the option to **produce Apache Spark programs instead of MapReduce.** Spark programs operate essentially in just the same way as I've described for MapReduce. In fact, I'm using the terms stage and shuffle from spark to properly describe both Map Reduce and spark.

There are differences, in that **Spark programs use more memory and reduce the use of disk drive for temporary storage,** and so can improve response times for long running programs compared to the equivalent MapReduce programs. Hive remains an excellent choice **when you want to process large amounts of data in a cluster using SQL,** and especially when the data you produce will be large as well, and you want to store the result back in your cluster. For instance, you may have a table with a few billion rows and you want to produce a new scrubbed copy of the table, with names converted to uppercase, and date and time formats made consistent, and rows with invalid foreign key references removed.

This kind of data transformation is **a common step in regular ETL pipelines** for data warehouses. You can write a command for this in HiveQL easily and the cluster will run the resulting program reliably, whether it takes minutes, hours or even days to complete. Hive distributed programs are fault tolerant. Even if a worker machine in the cluster fails during the program run, the cluster will automatically redo any work lost using the remaining workers.

- Hive translates SQL statements into other programs for actual execution.

This should be selected

- Hive is generally fast, but it can be open to failure for longer running programs.

Un-selected is correct

- Using Hive, you can input SQL statements, MapReduce programs, or Apache Spark programs.

This should not be selected

Incorrect. Hive accepts SQL statements, and translates those into either MapReduce or Spark programs.

- Hive is a good choice for processing large amounts of data as part of an ETL pipeline.

Correct

5.8 APACHE IMPALA

Hive translates SQL statements into programs in MapReduce or Spark to general purpose systems for big data. In contrast, Apache **Impala is built from the ground up as a distributed SQL engine for big data**. The concepts and design behind Impala, originated with Google when they realized they needed to build a high speed query system, specifically for big data. Google published a paper about their project called Google F1, in 2012. Marcel Kornacker, the technical lead on the F1 project, joined Cloudera to create similar functionality in open source software.

Cloudera contributed Impala to the Apache Software Foundation, and Impala became a top level Apache project in November 2017. Impala runs as a collection of Impala daemons running in a cluster. The word D-A-E-M-O-N can be pronounced damon or demon, either way. I find it more fun to say daemon. A daemon is a continuously running server program that awaits and serves requests as they appear. A web server, for example, is a kind of daemon. This diagram shows how Impala performs your interactive select statements. Your client connects to any Impala daemon in the cluster, and sends it your select statement. This daemon

coordinates with the other Impala daemons, and collectively, these daemons read and process the data needed to perform your query.

Finally, the daemon you contacted gathers the results, and returns them to your client program for display. Impala has a long list of special internal tricks, designed to process your big data queries as quickly as possible. An Impala query can run **10 or even 50 times faster** than the same query in Hive.

And the performance differences are even more striking with multiple analysts querying your cluster data at the same time. If, as will happen on occasion, a worker machine in the cluster fails while your Impala query is running. Your current program will receive an exception, and you will need to restart your query. This is a principle difference between Hive and Impala. Unlike Hive, which gives you strong assurances that your queries will complete, **Impala always opts for speed as a priority, even a priority over fault tolerance**. By the way, this diagram is slightly simplified, here is a more detailed version. There are a couple of other Impala processes called the **catalogue service** and states tour, that run in support of the Impala daemons. These processes have the sole duties of consulting the Hive metastore and data file storage, keeping the Impala demons informed with this metadata.

The SQL dialects, Impala SQL, and hive QL, are about 95% the same, so you can run most queries using either Hive or Impala. As a general rule, when you need to produce a new large data setting or cluster, Hive is a good choice for its reliability and fault tolerance. Requires that analyze your big data but return a small set for you to examine, Impala is the better high-speed choice. Impala is also good for business intelligence programmes or dashboards that query your cluster. It's no surprise that I had a large corporate customer say, **we use Hive to prepare our data, and Impala to serve our data to users**.

- Apache Hive
 - Reliable
 - Fault tolerance

- Apache Impala
 - High-speed
 - Business intelligence or dashboards

Suppose you want to run several quick queries on a large data set (in Hue is acceptable) to answer some specific questions. Which of the following gives the best choice and rationale for that choice?

- Use Hive, because you don't need a business intelligence program.
- Use Hive, because it's more reliable for large data sets.
- Use Impala, because it's faster.

Correct

Correct. For quick queries, looking for specific answers (and not, for example, to process and save the results for further analysis), Impala is a good choice.

- Use Impala, because it's not fault tolerant.

5.9 EXPLORING STRUCTURED DATA IN HUE

This is an exercise environment like the one you will use this week and for the remainder of this specialization. It contains all the software and data and everything you need for this exercise and for the remainder of the courses in this specialization. It's a simple Linux desktop, a few things on the desktop, and some

useful invocations in the menu bar that we'll use; a web browser, a command line prompt, and a text editor, all useful in this specialization. So, here I am invoking a web browser.

If you look on the lower right, you can see that you can click to invoke a different workspace or different desktop so you can switch between workspaces or desktops as you wish. Now, I'm going to invoke Hue, an open source project in our big data environment that acts as a web page over a lot of other useful applications like Impala and Hive. I'll log on with user training password training, and I'll let the browser remember that if I wish. So Hue comes up, and it comes up with a little hint window that I can dismiss if I wish.

There's the main Hue web page. It has a menu bar across the top with a few useful indications that the menu on the left, a drop list, a search window, and a few other invocations on the right. There's a quick browse area on the left that I can hide or display. I'll leave that displayed. There's a main application area on the right. Here in the quick browse area, I'm looking at database-type native sources, so table-like data sources including some databases. I'll go to the Impala data source.

There you can see Impala has already supplied with five databases that I can use; the default database which is a collection of related tables; customers, employees, and so on, the fly database, a different set of tables, and other databases. So a database is it's just its own little collection of related tables. I want to create a report on the default database, but first, I'm going to perform an important action for Impala. You see that little refresh icon here? I'm going to click that, and choose the bottom item in the radio list there, and click refresh. This causes Impala to synchronize all of its information about these tables and databases with what is in the big data store, so everything is lined up.

Now, going back to the default database, and I'll also choose in the menu, browsers and the table browser. So then the table browser comes up on the right. It's redundant with the quick browser on the left. I'm going to have both of those open for my use. So you can see I can navigate around different databases, different tables. There I can see in the path at the top, I'm in databases default, and there's the list of the five tables in that database; customers, employees, offices. There's a bit more information on the screen that I can ignore. Now, I'm opening a text editor because I want to create a database overview document about the default database. I'll call it "Database Overview." I'll sign it and date it. This is going to be useful for me later. Maybe I'll come back in six months and start working with this

database. It's good to know who wrote it and when it was done. So I'll name the database default.

This is what the report is going to be about. I want to start by listing the tables in the database. There's a customers' table, employees', and others. I can type those or I can just select and copy and paste into the text document, and then clean up the layout a little bit. Now, I want to get some overview information about each table individually. So I'll start with the first table, customers. I can click on the customers' table in the table browser, and I'll see some information about that table. Again, there's some information I can skip over for now, some properties, but what I really care about are the two things you see here; columns, that's the names of the columns and their data types, and a sample of a few rows in the table.

This is just so that I can start to intuitively get to know what's in that table. I can select, and copy, and paste that information into my text document. This is a little bit of a messy extra information and alignment. I'll clean up the document a little bit. So there I have my column names and their datatypes, and here's some sample rows with column headings and three sample rows. Now I want to go to my document and add some column comments where I think those are suitable. So I'm going to write that the cust_id column is customer ID, and the code PK says that I think that column is a primary key unique for each row in the table, and that the country column is a two-letter country code.

Name is pretty much self-evident. If I wanted to make a note, I could, but I'll leave it for now. That's my overview of the customers table. I'll go on to the employees table, put a heading in the document, go back to the table browser, go to the employees table, and select, and copy, and paste, and clean up. I'll make a few comments. The empl_id column is a numeric employee id, and maybe make another comment, the salary. There's a number there, but the unit of what the salary is is not known. So I'll make a note there, unknown units. Maybe that's US dollars. But I might want to go back to the original data source and confirm with the originator of the data what that unit is. I'm going to come back to the empl_id and office_id columns later on.

I'll leave that for now, and on to create a summary of the offices table. That's the basic cleaned up data from the office's table. I'm going to go on and create the summary of the order's table. But wait, let me put my comments on the office's table. So office_id is I think a primary key. I can put the comment it's an office_id or maybe say that's obvious. I might note that this state province column is a state

or province depending on what that is. Remembering again here that country is a two-letter country code. Now on to create a summary of the orders table. I'll put some comments on the orders table.

The order_id column is I believe a primary key. The cust_id column is a foreign key. I can go over to the Quick Browser area, look at the column names in the other tables, expand customers and employees, and it stands to reason that the cust_id column in orders refers to the customers table, its cust_id column. So table name.column name. So orders.cust_id references, customers.cust_id. You can see up here looking at a sample of the data for customers cust_id is a simple letter. Those letters for cust_id appear in the sample data for orders cust_id. Similarly, empl_id is a foreign key that references employees.empl_id.

We made it easy here by using common column names between the two tables. Total, that's an order total, and I'll make a note again, unknown units. I don't know what monetary unit that might be, might make a note maybe it's US dollars, maybe not, I'm not sure. That's the orders table. Finally, the salary grades table. While I'm copying and pasting this, I'll talk about it. This is what I would call a lookup table. It will only have just a few rows of some static information that I don't really expect to change very much. But that has useful information that I can put into my database to go with other data that might be changing more often.

What it does is that it shows ranges of salaries, and maps those two salary grades. So looking at the data, you can see that grade one is a salary grade with a minimum salary of 10,000, and a maximum salary of 19,999, and grade two is 20,000 up to but not including 30,000 and so on. You see these three sample rows. So then for the comment for min_salary, that's a minimum salary for a certain grade. Grade one is minimum salary of 10,000, maximum to 19,999. I'll make a note, and it's not really a foreign key, is not an exact value that will occur exactly as such in the employee salary. But that I'm giving ranges.

You'll learn later how to combine tables with information like this, where values don't occur exactly. But I'm just noting that's a minimum salary for the range, see employees.salary column, so on for the max salary column. Though, so both are the table in the database. I'm going to go to the top and add a couple of important notes to my database overview document. Note one is important. All the comments that I'm making, the notes that I'm making about these columns are not authoritative. They're my estimates of what I see on initially examining these

tables, just looking at some sample rows and the table definitions, and I'm putting my little short notes about the columns, they're not authoritative.

So I'm pointing that out. This is not from the original data source where these descriptions are coming. At a later time, I may be able to interview the data source and come up with more authoritative information about these columns, like the monetary units for example. My other note is this, the primary key, foreign key, and references notes that I'm making are not actual database constraints. Because an impala Data-store doesn't have database constraints rigorously enforced in the database, the way a real relational database would do. Instead, this is relational like, and I'm really describing data relationships than I'm assuming based on the data that I see. If my maintenance of this data is disciplined, then those relationships will hold.

So there's my report, my database overview document, and that's what I wanted to produce for an overview of this database. Going back to the employees table and remembering that `empl_id` acts like a primary key, and `office_id` in the employees table is a foreign key referencing `offices.office_id`. At least it behaves like a foreign key in the data. So this whole report took about 20 minutes to produce, and I think that was time well spent beginning to get to know the data. I'll save this report in a document. I'll give it a decent name. I'll save it on the desktop in this simple case, and there's that report.

Now, if I have DDL, that's data definition authority on these databases and tables, then I can actually add comments using you, where the table browser says comment, where I can click and add a comment as I'm doing here. Noting where I think something is a primary key, simple notes about columns. Maybe not a note about every column, but notes where I think they're called for. I can put a comment on the table, and I'll put a short description of the table, these are international customers for some company that this database refers to. There's comments on the customers table.

I'll go on to the next table, and describing all the tables. Now, I can use a desktop utility and just take a screen grab of the tables and their descriptions directly out of here, and go to the customers table, and take a screenshot of those columns with their descriptions and the sample data from Hue. I can create similar screenshots for all the other tables, and put those into a nice pretty looking document using Word, or Google Docs, or something like that. Then, here's the original text document of a database overview, or if I have DDL access to the tables,

I could go in and create comments on the tables and the columns in Hue, and then use screenshots and create a prettier document. Whether the document is this text document, or the prettier document, does not matter as much as the information in the document. Is the information good? Is information clear? The database overview is the important thing. This exercise of creating it is the beginning of my Exploratory Data Analysis, EDA database, and will be useful going forward.

5.10 WELCOME TO THE HONORS TRACK

In this course and throughout this specialization, you'll find the optional honors track as a focus on one concrete goal. Preparing you to **become a Cloudera certified associate data analyst**. Their certification is a separate credential from the ones you receive from Coursera. It is a professional certification and meant to verify your ability as a data analyst of big data. We cannot guarantee that if you go through this Coursera training, you will pass the Cloudera certification. Of course, we can't guarantee that. But we will help you to develop the variabilities that the exam is designed to test.

We'll give you practice with these skills you'll need to demonstrate, and familiarity with the resources you'll use both in the exam and in your professional work. The rest is up to you. We have designed this specialization and the certification with the same ultimate goal to help you build **proven mastery of data analysis in the big data world**, and to help organizations of all sorts to find people such as yourself for this work. For this week, I'm assigning you the task to read Cloudera's webpage on the certification program. Because that contains the core information you need to enroll and take the examination. I'll test you on that page because I want you to be secure in knowing how you will go about the certification process.

The Honors Track of this specialization focuses on the knowledge and skills you need to become a Cloudera Certified Data Analyst. If you wish to pursue this certification, you should familiarize yourself with the certification examination process. To that end, you will read the main web pages about the certification, and will answer questions about how Cloudera's certification works.

Study the public web pages about

- [Cloudera's certification program](#)

- [Cloudera's primary certification exam for data analyst](#)
- [The certification frequently asked questions \(FAQ\) page](#)

You may refer to these pages as you answer the quiz questions.

(Although the questions in this assessment are answered in the public web pages, you can learn more about how the certification exams work by taking Cloudera's free OnDemand course, CertPrep 101: Preparing for Cloudera Certification. You may find this short course—consisting of about 1 hour of video—especially useful when you get closer to taking the certification exam.)

If you're watching this video, then I assume you've read the web-page about Cloudera certification process, and you've verified your understanding of the web-page by taking our quiz. I'm glad you're interested in Cloudera certification. The honors track in the rest of this specialization, will give you the **extra practice with the tools and experience with the documentation**, so that you'll be largely self-sufficient and well-qualified to work as a big data analyst. I hope you'll then go on to take and pass the exam, and I look forward to seeing you on the roles of Cloudera certified associate data analysts.