4 Sept 2024

Jai Guru!

Course Overview

Hi, everyone. My name is Gerald Britton. Welcome to my course, SQL for Data Engineers. I’m a senior SQL developer and Pluralsight author. This course is part of the data engineering core skills path. In this section, we’ll be exploring SQL techniques useful for data engineering, including optimization strategies, utilizing SQL to build data pipelines involving extracting, transforming, and loading data for analysis, and leveraging your SQL skills in big data environments. By the end of this course, you’ll know how to utilize and optimize SQL for data engineering tasks. Now before beginning the course, you should be familiar with the basics of creating SQL tables and writing queries. From here, you should feel comfortable diving into related content with courses on database design for data engineers, data warehousing for data engineers, and ETL processes and tools for data engineers. I hope you’ll join me on this journey to learn these powerful techniques with the course, SQL for Data Engineers, at Pluralsight.

SQL Querying Techniques for Data Engineers

Leveraging Subqueries to Simplify Complex Queries

Hi there. Welcome to the course, SQL for Data Engineers. My name is Gerald Britton. This is part of the data engineering core skills path at Pluralsight. In this course, I want to introduce you to some of the key skills you will need for using SQL in various data engineering tasks and environments. Since this course is not designed to teach you SQL from the ground up, I’ll be jumping right into the topics and keep the SQL backstory to a minimum. First up, let’s look at subqueries. You might hear them called nested or inner queries, but the idea is the same, get results based on the output of another query or even multiple queries. This technique lets you build modular queries where logic is packaged with the data it works on. Let’s take a look at a few examples. For a simple example of a subquery, the first query here gets a single value from the subquery. That’s the manager ID of an employee named John, assuming there’s only one employee with that name. Since the subquery is used in the WHERE clause, this filters the result set to John’s manager, and the other query gets the manager’s name. The second example can return multiple rows. Imagine a manufacturer that makes electronics along with other things. Likely, there are many products in each category. Note the use of the in keyword. That means this version will work if more than one category ID is returned or even none at all. For a slightly more complicated example, say you want to get a list of customers with recent orders. Here, two tables are involved, a Customers table and an Orders table. This is called a correlated subquery since the customer and order tables are related by the customer ID. Using WHERE EXISTS, as shown here, is considered a best practice since it can handle null results better. Another way to use a subquery is in the column list of a query, as shown in the second example. Here, all customer names are returned along with the count of each order for that customer. That COUNT function you see is an example of an aggregation function. There are more available. Let’s look at some aggregation examples.

Incorporating Data Aggregation for Analysis and Insight

Aggregation in SQL can be used to summarize and analyze data, which can help to get important insights and statistics. Let’s look at a few of the options available. There are many aggregation functions available, and some may be unique to vendors like Microsoft or Oracle. But these five are supported by most database makers. SUM, as the name implies, adds up a numeric value and returns a total. Similarly, AVG returns the average of a value across a dataset. COUNT is the function we just used and does just what it says. Using the asterisk here means to count the rows, not just a particular column. Now tell me, what do you think MIN and MAX do? Right, return the minimum and maximum values, respectively. That works on both numeric and non‑numeric data. There are many more functions for statistical analysis like standard deviation, variance, and covariance among others. Now sometimes you want these aggregations over an entire dataset, but sometimes you want it broken down somehow. Let’s see how to do that. Here’s a simple way to do aggregation at a lower level than the whole dataset. We’ll use the GROUP BY clause to do that. This query will return the aggregate value, the average of salary, in this case, but report the average per department rather than the whole dataset. I’ve also added an ORDER BY clause to sort the results, although that is optional here. Other times you want to do this sort of aggregation and also filter the results somehow. In the second query, I’m filtering out departments where the average salary is less than or equal to 50,000. You can also combine aggregation with subqueries as in the third example where I ask for employees whose salaries beat the average. All these examples work on the whole dataset as they do their aggregation, but there’s a way to break this up even further with window functions.

Enhancing Aggregation with Window Functions

Window functions take the aggregations we just looked at and made them even more granular. They work on subsets of rows and could do more than just aggregate data. A big difference is that they add the function’s value to the result set returned instead of returning just one row per aggregation, as happens with GROUP BY. Here’s a simple example of a window function in use. The query starts off like any other. Then there’s a call to the window function, ROW NUMBER, in this case. Every window operates on a subset, and the OVER clause defines that subset. The whole dataset then is partitioned according to the columns specified in the PARTITION BY clause, department\_id, in this case, though there can be more columns if required. The ORDER BY clause is required for some operators. Here it makes sense since we want the employee with the highest salary in a department to have row number 1, the second highest, row number 2 and so on. For other window functions, ORDER BY is not required and really makes no logical sense, as in the second example. In addition to the ranking functions like ROW NUMBER and aggregation functions like AVG, there are other analytical functions and statistical functions available.

Structuring Subqueries Using Common Table Expressions

A few minutes ago, we were looking at subqueries and how they could be nested within other queries. One big disadvantage with that approach is that the subquery is more difficult to see and comprehend on its own. There’s another way. Using Common Table Expressions, or CTEs, as they’re usually called, you can write your subqueries ahead of time, unshackling them from the main query for readability and easier maintenance. Here’s how you can write subqueries ahead of the main query using CTEs. The magic is the SQL WITH statement. The general structure of the WITH statement is not particularly complicated. It starts with the keyword WITH. After the WITH keyword come one or more Common Table Expressions. I’ve just called them cte1 and cte2, but really you should give them meaningful names that reflect their purpose. A CTE may optionally have a list of column names to rename and future proof the results from a query that comes next. Also, if the query in the CTE contains expressions without aliases, you can put those aliases in the CTE header. In fact, I recommend that you do that. The query definition in the CTE is a select that can be as simple or complex as needed, pulling data from tables, views, and other CTEs and subqueries. Such a query can also contain set operators like JOIN, UNION, and ACCEPT. And you can have additional CTEs as required or desired. After the list of CTEs comes the final outer query, which can be any valid SELECT, INSERT, UPDATE or DELETE statement. This example uses that layout. It defines two CTEs that pull filtered data from two different tables, the employee and people tables, then combines that data in the main query, which joins the two CTEs together on employee ID.

Navigating Hierarchical Data Using Recursive CTEs

Recursive CTEs can be used to navigate hierarchical or graphical data using recursion. That means they reference themselves and iterate until some terminating condition is met. The basic layout of a recursive CTE is pictured here. It consists of two queries. The first is called the anchor member and is normally a query pulling data from a table view or even another CTE. The WHERE condition filters the data. After the anchor member, the next thing we need is a UNION ALL operator. This is required by the ANSI SQL standard for recursive CTEs, and it combines the results of the anchor and recursive queries, including duplicates. The second query is called the recursive member, and it pulls data from itself. Note that recursion stops when the recursive member stops returning results. Practically, you need to figure this out in advance to prevent runaway recursion, which will depend on your data. Applying this to our Employees table, here is a recursive CTE to navigate the hierarchy. The anchor member only returns top‑level managers since they have no manager. This could be a CEO, for example. A hierarchy level number is set initially to 1. The recursive member finds all those who report to managers and increases their level by 1. This ends when the self‑reference returns no results. The main query calls a recursive query to get the final result. One interesting thing about this query is how it uses an INNER JOIN, in this case, to the recursive CTE to find a manager’s employees. Well, this may be a good time to look at other join types and when to use them.

Selecting the Best Join Operation

The last example used an INNER JOIN. There are, however, more types of joins available, each with its own use case. Let’s dig into them. Let’s say that you have two tables that have some column in common that you can use for joining. You’ve probably already used an INNER JOIN. INNER JOIN returns rows where there is at least one match in both tables, but drops rows where there is no match. There can be anywhere from no rows returned to the row count of the largest table. The vertical bars in the table are the math notation for the size of a set. FULL OUTER JOIN returns all rows from both tables. If there’s no match from either the left or right table, those column values will be null. LEFT OUTER JOIN returns all the rows from the left table and any matching rows from the right table. If there is no match from the right table, those column values will be null. RIGHT OUTER JOIN returns all the rows from the right table and the matching rows from the left table. If there is no match from the left table, those column values will be null. You can think of left and right join as the same basic operation where the positions of the tables in the query are swapped. Which one you use should best reflect the business purpose involved. Note that you can usually omit the keyword OUTER since it is implied by the FULL, LEFT, and RIGHT keywords. One special join is a CROSS JOIN, which returns the Cartesian product of the two tables. That is all possible row combinations. The row count of the result set will be the product of the row counts of the two source tables.

Overcoming SQL Limitations Using Dynamic SQL

Sooner or later, you’ll hit a situation where you cannot write a single SQL statement to cover the business requirements. Even writing many statements in advance may not do the trick. This is where Dynamic SQL comes in. The idea is to compose the statement at runtime and then execute it. Let’s look at a few examples. For an example of Dynamic SQL, suppose you’re running on SQL Server and have an Employees table where you need to write a query, but the column name is in a variable. You can’t use a variable in a normal SQL statement for columns and tables, but Dynamic SQL is able to do it. Here, the query is built in a string with the shell of a SELECT statement and the ColumnName variable inserted as text in the column list. Then a special system function is used to compile and execute that statement at runtime. Note that Dynamic SQL is not limited to replacing column names. You can use it for any part of a statement, including things like condition testing in a WHERE clause. Now here’s the same sample we looked at before, but written in PL‑SQL for use on an Oracle database. And not to leave anybody out, here’s another favorite, the same query written this time for use in PostgreSQL. Dynamic SQL is powerful, and with great power comes great responsibility, according to Voltaire. There are three main things to look out for. First of all, Dynamic SQL is complex since you are thinking about the query that will be created, not the one you’re writing. This makes maintenance harder. Second, Dynamic SQL may not perform as well. In particular, it can thwart query plan caching and even optimization since the query can look different every time it is run. Finally, you run the risk of a very serious issue of SQL injection. In the preceding example, the column name was received in a variable. Now, imagine this comes from user input and the user typed in a valid column name followed by a semicolon and a DROP TABLE statement, as shown here. Whoa! That’s nasty. If you use Dynamic SQL, you probably don’t want an end user to be able to enter such things directly. Forewarned is forearmed.

SQL Query Optimization

System Factors that Influence SQL Query Performance

After you have created SQL queries that produce correct results, are you finished? Not at all. There’s one thing you need to do and do and do again, optimize your query for performance. Often you’ll be thinking about how fast you get results back to your users, and that depends on what is happening behind the scenes. There are four big factors that affect the performance of a SQL query or any server‑based activity, for that matter, that happen out of sight. The first is CPU utilization. Now most databases are deployed on powerful server infrastructure, whether on‑premises or in the cloud. Powerful, but still limited. A poorly written query can consume so many CPU cycles that other users cannot get any work done. RAM capacity refers to the amount and use of RAM available in the server to service the query. RAM is used for many things in a database system with buffering being one of the main uses. If a system is short on RAM, performance will suffer, and the whole system will slow down, and not just your query. The I/O subsystem needs sufficient throughput to handle the work given to it. Most modern systems use high efficiency data layers such as Fibre Channel, but you can still cause I/O bottlenecks with poorly written queries. The network between your users and your database can have an important effect on the performance they experience. Writing your queries with network transmission in mind can keep that concern and the traffic to a minimum. Now you may not have deep visibility into these or the power to do much about them directly. Still, poor queries can result in nasty calls from DBAs or network engineers. Much better is to get ahead of these issues by working on the things you can control. Let’s look at some of the top areas.

Reducing Query Time Using Indexing

One of the top considerations for performing queries is table indexing. Indexes can reduce query runtime from hours to minutes or even seconds. Let’s see how that works. So, what is an index anyway? Let’s look at an online resource, the endangered species directory maintained by the World Wildlife Fund on their website at the link shown. The list begins like this and had two pages when I recorded this course. A quick glance at the list tells you that the species are arranged alphabetically by their common name. Imagine though that you are a biologist looking for the scientific names in Latin. How would you find the entry for thunnus albacares? Well, you’d need to read right to the very end of the list since that has the common name, Yellowfin Tuna, like the one in your sandwich, and it’s the last one in the alphabetical list by common name. Well fine, you think. Not so hard to read through two pages to find what I’m looking for. Well, now imagine that this list holds all the known species in the world. That’s about 8.7 million arranged the same way. Reading through that list to find a species by its Latin name would not be much fun for you and would need a lot of I/O on a computer. In a database, I/O is often, if not always, the single greatest bottleneck. Now if we can reduce the I/O required to find the information we want, we can effectively give that bottle a bigger neck. This reduction will also lead to secondary benefits such as reduced CPU time, waits, RAM use and more. The goal of an index on a database table then is to reduce I/O. To understand how this works, we need to first look at a table that has no indexes.

Understanding Database Query Execution Plans

For a simple example of a table without any indexes, check out this Customers table. It has 12 typical columns, including interesting ones like CustomerID and LastName. Probably there will be queries on this table that look for particular customer IDs or names like the SELECT statement here. What do you think happens when this query runs? We can see that by asking the database to show us what it will do by looking at its execution plan. A query execution plan is a way to see how a database breaks down your query into units of work to get you the results you’re asking for. If I look at the query plan for this query on the Customers table, it’s pretty simple. However, look at the explanation at the bottom. This query will be done using a table scan. Alarm bells should be going off in your head. Why? Because a table scan means read every row in this table and return rows satisfying the WHERE condition. Well, what if there are a million rows in this table? There will be lots of I/O to read all those rows, CPU cycles to handle the I/O and look at each row, RAM to buffer the rows being read, and possibly network usage if the database is remote or in the cloud. An index can reduce all this work to a fraction of the I/O used by a table scan. I can easily change the Customers table to add an index on the CustomerID column by adding the PRIMARY KEY constraint. A benefit of this is that the table will no longer allow duplicate customer IDs, although the first version would. Now what about that other column, the LastName? This CREATE INDEX statement will put an index on the LastName column. With these two indexes in place, the execution plan should be better, and indeed it is. Now the database does an index scan. That should be much smaller than the whole table, so now we have an efficient query that minimizes system resources. Note that since this plan is from SQL Server, the word clustered here means that the index is embedded with the data. Now here’s a portion of a larger query plan that shows relative costs and estimated row counts. Each icon shows an operation that will be performed like compute scaler for the first box. The percentage cost of the total plan is shown above the box. Note the one at the bottom right, which shows 58%. This is the largest cost for this plan. Here, it’s an index seek, which means looking for a key in an index using the index tree structure. And that’s usually better than an index scan, as in the last example. The numbers on the arrows are the estimated row counts. These counts come from the statistics maintained for the tables and indexes and are used in the cost calculations.

Reusing Execution Plans with the Query Cache

I mentioned that the database optimizer produces an execution plan. Actually, it may produce several such plans before executing a query. A complex query may have many solutions. Each one will have an associated cost based on what it needs to do to get the job done. Statistics on indexes and tables are a big help here. When the optimizer is ready to run a query, it chooses the plan with the lowest cost. You might wonder what happens to a plan once it has been executed. Current DBMSs do not simply throw them away, but keep them in a cache for future use. Though it sounds counterintuitive, performance can be optimized by using cached plans since the plan optimizer can be bypassed. For every query, the optimizer uses system resources to do its work. Complex queries take more work to plan and use more resources. By using a plan that is saved from before, the database can bypass the optimizer, freeing up those resources for other work. Plans may be stored temporarily in memory or saved permanently to disk. Memory is faster, of course, but plans would then need to be rebuilt after a reboot. Parameterizing queries when possible can lead to greater plan reuse since the cache takes the parameters into account. Parameterization is often done using variables in the query, depending on the DBMS vendor. If plans are rarely used or are no longer valid for some reason, the storage used for them can be freed by invalidating them in the cache. Some systems give database admins tools to do this ad hoc, as well as other cache management tasks.

Designing a Database for Optimal Querying

Database design is a discipline all by itself, and database architects are in high demand. There are important principles to master though to set the stage for efficient queries. Few things affect database performance as much as normalization. That simply means to reduce redundant data as much as possible by separating out unique sets of data into their own tables. Check out this first try of a Customers table. It has all the things you might expect, name, address, email, phone number; however, often people in businesses share their addresses. If we duplicate that info into each customer entry, we waste space, cause extra I/O, and use extra memory and maybe network resources to process it. Normalizing this table, as on the right, separates out the address info into its own table. These two tables are then connected by a foreign key reference. This will take less space to store on disk and less I/O at the cost of needing an INNER JOIN to the Addresses table when you need that for your query. Normalization also reduces the work to keep those addresses up to date since any given address is found only once in the Addresses table and won’t be duplicated in the Customers table. The Customers table on the left has one index, the customer\_id, which has the PRIMARY KEY attribute. That’s great, but likely some queries will search by customer name and not ID. Now check out that query on the right. It’s looking for the customer ID based on the last name. Unless we do something, the whole table will have to be scanned to find Hobbs. The CREATE INDEX statement that follows does just that. The index uses both the first and last name columns, starting with the last name. So queries like the one above it can use the index and avoid table scans. Indexes like this are often called business key indexes since a business will think about customers by name, not their number. Systems often use similar queries to get results from normalized tables. To make this easier and less error‑prone and to encourage planned caching, you can encapsulate the basic query in a view. The table definitions on the left are the normalized Customers and Addresses tables. Certainly you can write a query using an INNER JOIN to pull them together. It’s likely though that this will happen a lot. So why not make it easy on yourself and everyone else? Create a view like on the right. This does the join so you don’t have to. The final query shows how to get the data you want from the view. Now if you like views, you’ll love functions. These take the logic of a view and add a parameter to make it even easier to get the results you want. The left column is the view from the last example. I can make it a function by wrapping the query in a CREATE FUNCTION statement, here for SQL Server. Now the query calls the function with a customer\_id as its parameter, making that query even simpler. Note that creating functions like this is not part of the ANSI SQL standard, although you will find them supported by major database vendors; however, the precise syntax will vary.

Query Writing Habits to Adopt and Avoid

Writing SQL has much in common with coding in other languages. It’s part engineering and part artistry. Let’s look at some good habits to learn and bad ones to avoid. The query on the left selects four columns from three tables joined on other columns if some conditions are met on two other columns. Now, which tables do those columns come from? I don’t know, and that’s the point. All 10 columns could come from one table or be spread around. The database query compiler can figure it out, but you’ll need to keep the table definitions handy or have them memorized. Worse yet, if the table definitions change, the semantics of the query might also change, causing incorrect results. Now consider the query on the right. Each table has an alias, and each column reference uses its table alias. Now it’s clear when reading the query which column comes from where. This will help the next person who needs to work on your query or even yourself a few months later. The left‑hand query here includes two nested subqueries. Parsing this out on your head is difficult. You need to think from the inside out. The right‑hand query uses CTEs for all the tables involved. Each CTE is self‑contained and does one thing. The main query is simple joins, easier to read and maintain. To the database compiler though, this is semantically the same, and the results will be the same. Avoid nested subqueries as much as possible, and let the compiler do the work for you. And P.S., if you also work in object‑oriented languages like Java or Python, the CTEs follow the single responsibility principle. SELECT \* has its uses, especially when writing and debugging queries. However, it should not be used in production code for a few reasons. When you use SELECT \*, the database gets all the columns from the table, including some you don’t want or need. This can impact CPU utilization, RAM consumption, and network latency. SELECT \* affects readability because it’s not clear what the query really wants to get from the database. Very rarely is that all columns from all tables. Finally, your query may work for a year and then start failing or returning erroneous results all because columns were added or dropped from one or more tables or even moved around. There are a few other habits to avoid or break if you’re already using them. First, avoid scalar functions in WHERE clauses or join on clauses. Function calls may prevent the use of table indexes or even stop the database from performing parallel operations when it otherwise could. Second, avoid Dynamic SQL. Even though I just talked about it, this should really be a last resort option. It can harm performance, readability, query caching, and lead to SQL injection. Parameterized queries and procedures are much better options. Third, avoid unnecessary GROUP BY and ORDER BY clauses. In this query, the GROUP BY clause is redundant since the query uses the DISTINCT keyword. For that matter, avoid DISTINCT since this may cause an internal sort operation, and sorts are expensive. By the same token, avoid ORDER BY if you can. For example, if a query result is used in another query, the order of the results from the first query is irrelevant and you don’t need ORDER BY. A couple of other things to avoid when you can are cursors and programmatic looping. These tools are usually used for Row By Agonizing Row processing, or RBAR, as Jeff Moden calls it. These are performance killers. Well, I’ve only covered these topics briefly, I know. Really, each one needs an in‑depth study to master the concepts. Thankfully, Pluralsight has lots of other content on these, some of which I even worked on.

Data Workflows for Data Engineering

Understanding Data Ingestion

By this point, you should have a good understanding of the basics of utilizing and optimizing SQL queries for data retrieval and analysis. Now it’s time to look at the bigger picture, building workflows that leverage SQL as a powerful partner for efficient data ingestion in engineering workflows. Data ingestion is all about getting data from diverse systems into a target system for analysis and decision making. The process involves one or more of the following steps. Data extraction means getting the data from some source system external to the target system. Typically the target system is a database, and the source system may be other databases, documents in JSON or XML format, flat records like CSV files, spreadsheets and others. Data validation ensures that the extracted data conforms to the business standards, for example, discarding information with dates out of a certain range, non‑numeric data when it should be numeric or addresses that can’t be found in the real world. Data transformation makes data conform to some standard, for example, converting floating point to fixed precision numeric, making dates conform to day, month, year or year, month, day standards or removing leading and trailing spaces. Data aggregation involves summarizing some data before loading it into the target system. Although this potentially loses some level of detail, it can greatly reduce target storage requirements. Data enrichment refers to enhancing the data extracted with other data to make it more complete. For example, looking up a customer name, given a customer ID. And data loading is the process of storing the data in the target system, often referred to as a data warehouse. Now you shouldn’t necessarily consider this a logical order to follow. There are two common approaches you will find in use.

Comparing ETL and ELT

There are two main approaches to data ingestion. The first is called ETL, which is an acronym for extract, transform, and load. It usually follows the order we just looked at. Data is extracted from source systems, optionally goes through validation, transformation, aggregation, and enrichment. Then is loaded to the target system. This approach is appropriate for processes with complex cleansing and enrichment requirements and where powerful systems are available for these steps before loading the data or where the target system has computational or capacity limitations. The second approach swaps the load and transformation steps. Here, data is extracted from source systems and then loaded directly without much transformation. The validation, transformation, aggregation, and enrichment steps then are performed on the loaded data. This strategy works well when the target system has sufficient power and capacity compared to the extraction system. Modern data warehouses frequently meet this requirement. Note that these steps are not rigidly followed and that some systems may follow a hybrid approach, perhaps extraction and validation before load, then cleansing and enrichment once loaded. Let’s look at a few examples of these steps.

Validating, Transforming, Aggregating, and Enriching Data

For an example of extracting data using SQL, consider this snippet using SQL Server as the system. First, I create a table to hold XML data. In this case, I’m using a table variable for convenience. Next, I load the document using the system OPENROWSET function and insert it into the new table. Finally, I can extract the value from the column using XML path notation. Note that I extracted and loaded it as is before working with it. Now if the data was not valid XML, I would expect an exception. Oracle lets you treat a JSON document like a table by first creating an external table, defining the format and fields, setting up the path to the file, and then you can read from the file as if it were a table. Similar facilities are available in other systems and for other file types. You can also do data validation, transformation, and enrichment in pure SQL. Here are a few examples from popular database systems. On SQL Server, you can use the TRY\_CAST and TRY\_CONVERT built‑in functions to test for valid data. This SELECT will return null since the date is not valid, but would return a valid date if the year were set to 2024. If you are running Oracle, the REGEXP\_LIKE function will return false if the data does not match the regular expression. Here, since it’s in the WHERE clause, the row would be omitted if the column does not contain a valid number. On PostgreSQL, you can check if strings are valid using the is\_numeric function. Special functions like these are often vendor‑specific, so be sure to check their documentation. ANSI SQL specifies the cast function for data conversion, and most vendors implement it for data transformations, though without validation. We looked at aggregation earlier in this course, but not enrichment. So, here’s a simple example where the source data is enriched by adding the customer name using a lookup from the customer data table.

Automating Data Pipelines

You’ve just seen some of the possibilities for ingesting data from source systems using SQL that can be an adequate solution for many problems. On the other hand, if you’re working on a project for an enterprise, you should be thinking about automation and scalability. Solutions like this are often called data pipelines. Let me give you an idea of what a data pipeline is. Data can come from many sources, cloud storage, a commercial database, documents, maybe formatted as JSON or XML or a website and others that data is then piped to some storage system that can handle SQL queries since we’re concentrating on SQL here. Other targets are usually available. The pipeline is where the work of ETL or ELT takes place. Just like a real‑world pipeline, this needs to be dependable without the need for constant hands on. You could design your own pipeline, of course. There are, however, many commercial offerings available that have proven themselves over decades of real‑world use. Let’s see what they can do.

Breaking Down Data Ingestion Solutions

Modern ETL systems have many functions and capabilities in common. In addition to the ingestion steps we already looked at, you will find other important components that help build robust, scalable, automated data pipelines, including workflow scheduling and automation, which provide for interval or event‑based ETL jobs, manage job dependencies, and optimize resource utilization. Data quality goes beyond validation to encompass cleansing, standardization, and deduplication. This helps companies maintain data standards and find and fix anomalies and inconsistencies. Often data pipelines are complex when multiple sources are involved and multiple stages are required to ready the data for analysis. ETL systems can orchestrate these to manage dependencies and handle errors with little to no human intervention. Traditionally, ETL systems have entailed batch processing where data is ingested on a daily basis often overnight. That is changing, and real time or near real time processing is now available for time‑sensitive analysis and decision‑making. Many business types are highly regulated, especially financial industries like banking, investing, and insurance. To help with regulatory reporting, metadata management captures the data about the data, enabling data lineage tracking and governance, making regulatory compliance and reporting easier. Finally, ETL systems can monitor themselves to track pipeline execution, show job status, and measure performance usually with dashboards, alerts, and loggings to keep things running smoothly.

Surveying Data Ingestion Vendors

There are many ETL systems vendors today that implement the functionalities we just looked at. Some of the top ones you might run into are Informatica. That’s one of the oldest that dates from the early 90s. It has a large set of connectors for all kinds of sources, including databases and cloud‑based systems like Salesforce, Google, and Amazon Web Services. It includes an easy‑to‑use graphical design tool to create data pipelines. IBM Infosphere DataStage is another one that has been around for more than three decades that you might find in more IBM‑focused environments. The Oracle Data Integrator is a heavyweight in the ETL world and coming up soon on its 20th anniversary. Like Informatica and Infosphere, ODI has multiple connectors and a graphical development environment. Apache Airflow has the distinction of being open source and is maintained by many independent developers around the globe. It is a natural selection where tools like Hadoop, Hive, and Spark SQL are in use. Microsoft’s SQL Server Integration Services, SSIS for short, has been the go‑to solution for SQL Server databases for two decades. It has less built‑in automation than some of the others, but pairs well with other job schedulers. Talend Data Fabric is a robust and popular data integration solution with a graphical user interface and easy‑to‑configure components. Built on an open‑source base, Talend is now a commercial product. Amazon Web Services Glue is a serverless option for AWS users. Their data factory product is a managed cloud‑based solution. Microsoft also offers a data factory product built on their Azure platform with a similar feature set as the rest of the pack. You’ll likely encounter several of these in your work as a data engineer.

SQL Meets Big Data

Introducing Big Data

Big data emerged as an idea in the early 2000s due to the expansion of the internet and the explosion of digital data, creating new challenges for organizations to capture, store, manage, and analyze the vast amounts of information coming in. Data engineers need to understand the big data technologies and be able to use SQL skills to drive business decision‑making. Big data is the name we give to the ever‑exploding wealth of digital data in our world. That can come from web pages, mobile devices, social media and more. Handling this is the challenge. Advancing technologies, including computation, storage, and networking enable distributed frameworks and file systems. Affordable commodity hardware makes the supporting infrastructure cost‑effective. Open source is everywhere, and collaborative development drives innovation in big data. Some of the biggest names like Hadoop, Spark, and Hive came not from giant corporations, but rather developer communities working on the big data problem. Businesses see the value of analyzing this data to make data‑driven decisions, often called business intelligence. Now, there are tools to explore vast datasets to drive strategic decision‑making and gain a competitive edge. Data science has emerged as a key discipline, and advancements in machine learning and artificial intelligence are helping analysts build predictive models and perform advanced analytics. The ever‑growing Internet of Things includes devices ranging from smart thermostats to industrial machinery and more, generating huge real‑time data streams, requiring timely analysis.

Understanding Apache Hadoop

By far the biggest player in the big data space is Apache Hadoop. It really is the elephant in the room, as the logo suggests, and it’s no baby anymore. Worldwide, it dominates its competition with 35% market share. Also, Hadoop is the foundation for Hive, which provides a SQL interface for querying and managing data in Hadoop. The four pillars of the Hadoop architecture are HDFS, the Hadoop Distributed File System. This is a storage system that operates on a master slave basis. YARN, an acronym for Yet Another Resource Negotiator. I touched on this a bit in the previous module. YARN manages and schedules jobs that run in Hadoop, managing resources and distributing work to processing nodes. MapReduce, a programming model and engine for parallel processing. MapReduce jobs are run on Hadoop using YARN. And Hadoop Common, which holds the libraries and utilities used by other Hadoop components.

Querying Hadoop Using Hive

Now that you have a basic idea about Hadoop, it’s time to learn about querying Hadoop using SQL. Apache Hive is an open‑source data warehousing and SQL querying tool built on top of Apache Hadoop for managing and querying large datasets like those stored in Hadoop’s HDFS. Some of the key aspects of Hive are you can use SQL for writing queries on data stored in Hadoop, usually in HDFS. Queries are converted internally to Java for compilation and running, for example, using MapReduce. The Hive is not like a traditional RDBMS, and schemas are not defined in advance. Instead, data can be stored in any format, and the schema is added when the data is read. So it can read both structured and unstructured data with ease. To support schema on read, Hive supports many storage formats, including text files, row column‑formatted files and others. The Hive is primarily designed for batch processing rather than real‑time querying. It is very efficient running large batch jobs, but not ideal for real‑time querying. Since Hive is integrated with Hadoop, it can use other Hadoop components like HDFS and YARN and Hadoop’s built‑in security. Here is an example of a Hive query using schema on read and the CSV file as the source stored in an HDFS warehouse. First, the CREATE TABLE statement starts like a normal SQL table definition with columns and data types. Then the next section says that this is delimited data, in this case, comma delimited. The statement continues by pointing to the location of the data on an HDFS file store. Finally, a simple SQL statement can be used to get the average salary per department. Now suppose that the data was stored as JSON. Only the row format clause would change and would now look like this snippet here at the bottom. SERDE stands for serialization, deserialization and points to the Hive metadata for JSON‑formatted files.

Utilizing Apache Spark for High Performance Analytics

Hive seems pretty cool and complete, don’t you think? Well, what if you’re not satisfied with its performance or need to perform advanced analysis, handle real‑time data as it arrives or integrate with other big data frameworks? That’s where Apache Spark comes in. Spark offers several compelling advantages when compared with Hive. Generally, Spark performs better than Hive, thanks in no small part to in‑memory computation, which can substantially reduce the need for disk I/O. Spark has more flexibility and can be directly used in high‑level languages like Python, Java, Scala, and R. You can also use Spark from an interactive shell for immediate results. If your data streams in at all hours from real‑time feeds, there’s no need to wait for high style batch processing. Spark can stream data as it arrives for immediate analysis. Spark provides a rich set of libraries for advanced analytics and machine learning like Spark MLlib and Spark ML. With these, you can do complex analytics directly inside Spark without moving to another system or platform. Spark integrates well with many big data tools and platforms, including Hadoop, Kafka, Cassandra and others. For an example of querying Spark using Python, check out this code fragment. First, I import the Spark module containing the API I need, then I start up a new Spark session, which sets the variable Spark as a pointer to the session object. Now I can easily load a CSV file with headers using the built‑in read.csv method. I can then query the data using Spark SQL with very few lines of code. You can run these commands interactively in a Jupyter Notebook, for example. Great for quick exploration and analysis of data without the need for a full Spark job. Now here’s the same query, this time run from a Spark shell interactive session. This contains the essentials from the Python example, but it’s all done in an interactive shell. That means I don’t need a separate call to start the session. The first part loads the sample CSV files with headers. The second part queries the data, filtering on people under the age of 30. Alas, I passed that milestone quite a while back. You can execute these commands one by one in the Spark shell, and you’ll get immediate feedback and results. Very handy for exploring data with Spark.

Swimming in a Data Lake

A logical extension of the concepts behind big data is the data lake. This lets you keep all your data structured or not and work with it there. This is a good fit in a world of data coming in from all sorts of things where structuring it before storing could be expensive and time‑consuming. Some of the key characteristics of a data lake are they are highly scalable by adding storage nodes as needed without hitting hard limits. They’re flexible, accommodating structured and unstructured data without any transformation cost. They’re cost‑effective, often built on typical big data platforms like Hadoop HDFS or Microsoft Azure. Since they use schema on read, there’s no need for typical database schema definition and management. Data lakes are ideal for exploration using tools like Apache Hive and Spark, as well as machine learning frameworks. Proper governance and security are essential for any data solution, and data lakes can employ access controls to safeguard their contents. You’ll often find data lakes integrated with ETL systems like the ones we looked at earlier for automation. Remember when we looked at data pipelines? Well here, I’ve got a visualization of data coming into a lake, same inputs, similar SQL access. The big difference is that the pipeline is gone, and instead we have all the data in one lake. Now, there will likely be other pipes involved pumping the data into the lake. The key difference is that the data is piped in as is without the usual transformation.

Decentralizing Data in a Mesh

A new approach to managing data in large organizations is that of the data mesh. It is a response to the challenges of centralizing data storage in traditional databases, Hadoop clusters or even data lakes, which can lead to scalability and governance issues. Let’s look at some of the characteristics of this emerging trend. Data meshes are now appearing with characteristics like decentralization. Instead of a central team managing all the data, it’s managed by the parts of the business that own or generate the data. After all, they are the experts. A data mesh organizes data around business domains rather than technical or business silos, making the data more relevant to the people who use it most. Data in a mesh is treated more like a product that is used across the business. This then encourages data producers in each domain to produce high quality data, accessible through well designed APIs. Domain teams have the autonomy and tools to manage their own data, which may include data pipelines and even their own data lakes, thus reducing dependence on centralized tech teams. Even with a data mesh, there’s no getting away from centralized governance to ensure quality, security, and compliance. A data governance team sets up policies and standards to meet these requirements. Data meshes work to meet today’s and tomorrow’s challenges of scalability and agility through a decentralized architecture and a product‑oriented mindset. This is an active area of development and research, working to find ways to unlock and exploit a company’s data assets.

Course Summary

This course has been a bit of a whirlwind through classic, current, and emergent technologies for data engineers using SQL. I think you can see that each topic and subtopic deserve a deep dive on its own. Thankfully, you’ll find them as you work through the Pluralsight paths available for each area. It’s been great having you with me on this brief journey. Hope to see you again soon on pluralsight.com.