Databases: Executive Briefing

by Simon Allardice

Start Course

Bookmark Add to Channel Download Course

Introduction: The Problem Databases Solve

Course Overview: What to Expect

These days it doesn't matter if you're part of a Fortune 500 organization or a brand new startup, your knowledge of databases and your ability to improve that knowledge is one of the most valuable and long lasting business skills you can have. And I stress it's a business skill because unlike many technologies, what's most important here is not having specialized technical expertise that only applies to one particular product, instead what's valuable is a generally accepted set of concepts and approaches, that have been around for decades, can be learned quickly and then applied in multiple situations regardless of which vendor and which database you have. And simply being able to ask more informed questions about databases can shape the success of your project, even the success of an entire organization. So, in the next few minutes we will get clear on the most important ideas here. We're going to cover the jargon, the terms and phrases you'll encounter again and again, not just database, but a bunch of other data terms that come along for the ride, like data analytics, big data, data warehousing, data modeling. We'll talk about normalization, schemas. We'll go over relational versus non-relational, and more besides. Cover acronyms like SQL versus NoSQL versus NewSQL, and why do some people say S-Q-L and

some people say "sequel". We'll then use these ideas to make sense of the current marketplace of database solutions, all the major players who have been around for decades and why they've been so successful for so very long. We'll go over recent trends in the market, the emerging technologies, and what you should expect in the near future. And whether you already have a team or you need to build one, I'll talk about the roles, skills, and expertise you'll want to have. So let's begin. But, first I need to deal with a very common misunderstanding about databases and it's a misunderstanding that happens because databases are so common, because they're ubiquitous, databases are everywhere. Most people know they've been using databases in their life and work day in, day out, for years. And because of this, most people think they know what a database is. Most people think they understand the problem databases were designed to solve. And most people are wrong. Let me explain what I mean.

The Problem Databases Solve

If you've ever gone through any kind of introduction to databases before, you may have encountered phrases like a database is a repository of data or a database is a location to store important information. And I cringe whenever I see definitions like these because they're technically true, but they don't explain anything. It's as if a 10-year-old child asks me, what is an engine? And I answered by saying, oh, an engine is a collection of metal components. Again, technically true, most engines are made of a collection of metal components, but it's a useless explanation because it doesn't say what's important about engines, why they're valuable, why they're useful. And likewise here, yes, a database can certainly be described as a repository of data, a place to store your information, and that is how a lot of people think about them, but it's not why they're valuable. It's not why they're useful. Whatever data you have, data about customers or invoices, web activity, products, purchases, inventory, details about patients or about employees, financial transactions, website content, documents, images, video, whatever it is, if all we cared about is having a place to put it, to store it, to save it, we don't need a database to do that. You know this. If you have some simple data, you could just open up a file, like a text file or a spreadsheet, and just type it in there, save your files in a folder somewhere. Done. There are lots of people, even small businesses, who work this way. They make their own location to store important information, they make their own repository of data. There's nothing magical about having a place to store information, we've always been able to do this. So the fact that we have data is not why we need databases. We need databases for what happens next. What comes after the words "we have data", like we have data and it will grow. We start off with a little and suddenly we have a lot of it. And it's one thing to use a text file or a spreadsheet to store the

details of 100 customers, it's quite another thing to store 100, 000 or a million or 10 million. Then there's we have data and it will change over time. The address of an employee, the current inventory amount of some item, the price of a product, and when that data changes, we often want to know not just what it is now, but also still keep track of what it was yesterday, last week, last month, last year. And of course, it won't be just us, it won't be just one person looking at this information. So we have data, which must be shared. Now shared might mean a handful of people on a team need to be able to edit and work on the data without overwriting each other's changes, but shared data could also mean information used on the website made available across the world to anyone and everyone, and if we're going to do that, another concern might be we have data and it must be fast, we have data and it must be searchable, we have data and it must be reliable, oh and there's a word. Reliable could mean uptime. Will the data always be reliably available anywhere in the world, 24 hours a day, 365 days a year? But some people might say reliable data means trustworthy. If I ask, how many seats are still available for this performance? I will always get one answer, it will always be accurate and up to the millisecond. And in any real world business situation, our data will be interdependent, data in one part of our system that is separate, but related to data in another part of our system. So if we were keeping track of, say, seat allocation for an upcoming concert, and the seat allocation data tells me 37 seats are currently reserved for this performance, can I then look in another part of the data and find the financial transactions for a corresponding 37 ticket sales? So we have data that needs to be consistent within itself. Something that's true with any data, but particularly with things like financial transactions, personal information, healthcare information, is that we have data that must be protected and secure. Protected from internal hardware failures, backed up constantly, recoverable from disaster, but protected from external threats and attacks, even protected within our own organization, data that must be confidential and controlled with defined access levels. Who is allowed to edit this data? Who is allowed to just read it? And who isn't allowed to even look at it at all? But let me stop there, because while I haven't tried to cover everything we might possibly say after the words "we have data", it's enough to make the point. The reason we use a database is not to help us deal with the problem of just having some data, the reason we use a database is to help us deal with all the other problems that having data always brings with it. It's going to change, it's going to grow, it needs to be shared, it needs to be fast, it needs to be reliable, it needs to be kept consistent, it needs to be secure. And that's why thinking of a database as just a repository is data is focusing on the wrong thing. Because a database is not about simply having a place to put our data, it's about having an organized system to put our data into. And when we put our data into that system, it's going to help us do all these other things. But to have this organized system, to have a database to put our data into, it must first be

defined. It will be up to us what exactly is most important about all these other factors. How much do we expect our data to grow? How do we intend to share it? How much structure? How many rules do we want to or need to wrap around our data? And we'll do this using a Database Management System, or DBMS. And DBMS describes an entire category of software products. Microsoft SQL Server is a database management system, Oracle Database is a database management system, as is IBM's DB2, Apache Cassandra, Mozilla Firebird, MySQL, MongoDB, and many, many others. Now we're going to explore more about the different options a little later. Right now all we need is the idea that if we were starting from scratch, we would need a DBMS of some kind. Now with many of these database management systems, you can either download and install the software locally and on-premises, or you could choose to use a hosted, cloud-based version. We then use that DBMS software to create and define and then work with one or more databases. Each database within the DBMS is self- contained, it will have its own data and its own set of rules about that data, about security, about internal consistency, however we choose to define that. And the database management system will continually run in the background to surround and manage our databases to make sure that those rules are applied. And one of the reasons I don't need to drill down into the differences between these different database management system products is that many of them are built on the same set of core ideas and concepts. So next, I'll talk about the different steps of defining a database, and we'll cover many of those core concepts.

Databases: Core Features

The Importance of Relational Databases

Databases are one of the few technology areas where products released 30 or even 40 years ago are still entirely relevant today. As I said, this is one of the most long lasting technology skills you can have. I started programming full-time in 1985 and one of the technologies I used in my first job was IBM's DB2 database management system. Now the programming languages I used back then are no longer in demand, the operating systems have long gone, the actual computer hardware I was working on can occasionally be found in the vintage section on eBay, but DB2 has remained a viable product and a relevant skill ever since. And it isn't the only one. PostgreSQL, a popular open source DBMS was first released in the 1980s, as was Microsoft SQL Server, and Oracle was released in 1979. Now sure, all these products have been improved, updated, refined,

but they've also been continuously relevant for decades. And the products I've just mentioned, and quite a few other popular DBMSs were built on the same set of ideas, the same concepts. They share a remarkably similar approach to what a database should be. And they're considered not just database management systems, but a particular type of database management system called a relational database management system, or RDBMS. And while there are other types, the relational ones have been the most popular, most common, more prevalent databases over the last several decades. Now, if you're thinking, I don't want to hear about any of these old school database management systems, I only want to know about the newest cutting edge database products. The reason the relational DBMS products have stayed relevant for decades is they're very good at what they do and what they do is perfectly suited for many typical business scenarios. And while the last few years has seen an incredible amount of new products released in this space, understand that many more recent database technologies were designed to deal with specific situations that the standard relational databases weren't great at. And what that means is it's genuinely difficult to explore the newer ones without directly comparing and contrasting them against classic relational databases. So we'll start with the classics. And so I don't have to say it 1000 times, just assume for the next few clips every time I say database, I mean relational database.

Defining a Database, Tables, Rows, and Columns

A lot of the work of defining a relational database can be done before you've even decided on or installed a DBMS. Much of this can be and should be done on paper or at a whiteboard. And the first question, what's it for? If you think you need a database, what data do you intend to put in it? And the answer, most of the time, is not, well everything. Yes, a few organizations try to have one database to rule them all, a single corporate database containing every piece of company wide information you might ever have or ever want. But it's much more often the case that you'll end up with multiple databases in an organization. Each database designed to hold the data for one well defined, bounded area. Now that might sound a little vague, but okay, there are different reasons. Sometimes a database is to support one business unit or one department, like having a human resources database or a finance database. Other times it might be organized around a business task, like purchasing or customer relationship management, or perhaps having one database just to hold the content for your public website. But we need a scope, we need a reason for the database so we can meaningful talk about what stuff belongs in it. What do we want to store data about? Is it people or objects or ideas or raw data, like website analytics, or all of the above? But we don't then just gather all our data and dump it in the database, we first describe

the different pieces, the different elements, the different things that we want to store data for. Now when we're at that whiteboard or paper or even just talking about this stage, the word entity is often used, because it sounds a little better than saying thing or object. What are the separate describable entities that belong in this new database? If we're creating a database to support online purchases, then entities might include things like a customer, a product, an order, an invoice, an payment. And a database to support classroom scheduling for a training center, we might begin with entities like instructor, student, classroom, course, event. We then define a table for each entity, and a table is the fundamental building block of every relational database. We must have at least one table, we will typically have multiple. And until you define tables, a relational database is useless, there's nowhere for your data to go because all our data interactions, saving data, reading it, updating it, deleting it, they're not done with the database, they're done with a table inside the database. A table for each entity. Now to be clear, that doesn't mean we make a new table for every single customer we have. No, it means we define a table to hold customer data, another table to hold order data, another table for all the product data, and so on. And each table consists of well-defined, structured, and repeated data. We can begin by thinking of a table as something like a spreadsheet, it is a grid of columns and rows. Columns like columns in architecture running vertically, rows running horizontally. And each row in a table will represent a single individual item of whatever your entity is, simply put that means if we've created a custom table to hold all our customer data, then one row in that table is one specific customer. Or one row in the order table is one order. In a patient table, a row is one patient, and so on. And the columns are describing the distinct pieces of information we want to keep track of for that entity, like a customer email or a customer phone number or a customer zip code. Now some tables might only have 2 or 3 columns, some tables end up with 50 or 60 or more. so when we're defining a database, our job is first about figuring out what tables need to exist, and then defining what columns we need in each of those tables, because we can't start adding rows of actual data until we've said exactly what those rows are going to be made of. Because within each table, every single row has to have the same structure and follow the same arraignment of columns. So where this differs from the idea of a spreadsheet, is the database tables are intentionally not flexible, they are not freeform. When you define a new table in a database, you don't get the equivalent of a blank spreadsheet, you just start typing random data into. No, we have to define every single column that we want to have, and not only give that column a name, but also say exactly what type of information, what type of data is allowed to be in that column. So from the very beginning, we begin to set rules and constraints for our database to make sure it's always meaningful. If we were defining, say, an employee table for a human resources database, we might want columns to represent first name and last name and a column

to represent the date they joined the company and another one for if they have a bonus percentage, and are they currently fulltime or not? But we would also say, these columns must be text, this must be a date, bonus percentage must be a number, and this column should only allow either true or false in it. And when we make these choices, the database management system, the DBMS itself, will enforce them. It won't allow data that doesn't conform. And this is great so that later we don't have to worry about having a bunch of missing or garbage data in the database, we can trust it, we can rely on it. And the most important column we will need to define in any table is the key.

Sidebar: Database Symbols

A quick sidebar, when planning a computer system, we often draw diagrams showing how the different pieces will interact, and we use different symbols to represent things like web servers and load balancers and firewalls and databases. Now there's nothing that stops you from drawing the symbols any way you want to, but if you do an image search for database symbols, you will see that a database is typically represented by a cylinder, and usually divided into several horizontal segments. And if you wonder why we ended up with this particular symbol, well, it's because this is what a lot of databases used to physically look like. The thing is, all your data has to be stored somewhere. It's on some physical piece of computer hardware, commonly on a hard drive like this or an array of hard drives in the server room. Even if you're saving all your data into the cloud, well, it still ends up on a hard drive in somebody else's server room. Now go back a few decades and the predecessor of a hard drive was called a disk pack. Several large circular disk platters all enclosed in a protective cylinder, like this. This is an early removable disk drive, it's the kind of a think a large organization would have used to store database data back in the 1980s. You place it into a large drive enclosure and screw it in. This particular one is a digital RA60P, it held 205 megabytes and cost $15, 000 plus a monthly maintenance contract. Obligatory conversion into today's numbers, to get the capacity of this rather unremarkable 2 terabyte hard drive using these, you would need a stack of them twice the height of the Empire State building and costing $150 million dollars. But, this is why a database is traditionally shown as a cylinder with divisions. End sidebar.

Using Keys and Indexes

Whether we have a customer table with a million customers or a patient table with 100, 000 patients, or even just a user table with only 5 users in it, we always need to be able to get directly

to one specific row. Whether it's to get the data for one customer or update one particular patient or delete one specific user row. And the typical way is that in every table, we will have one column to be used as the key, or more explicitly the primary key for that table. And for a column to be a primary key, it must contain unique values where no two rows in the table hold the same value for this column because the primary key must be able to take us to one, and only one, row. We can't have duplicates. Now sometimes a piece of data we've already decided to store is good enough. We might decide that for our user table, the email address should always be unique, so we will use that as the primary key. Or if we're making a table to store vehicle data, we might say that the existing VIN number column can uniquely identify each row. But often there isn't a piece of data we can guarantee to be unique. So for many tables, we'll define a new column just to have one to be the key. This is often an ID column of some kind, like customer ID or order ID or a patient number. Now we can also tell the DBMS to automatically generate a unique value for that primary key for every new row that's entered. You're adding a new customer, well the last one had customer ID 442, so this one will be customer ID 443. And by defining a primary key, you're not just doing one thing, you're not just telling the database what column can uniquely identify any row in the table, because that by itself wouldn't fix everything. Because if we had 5 million customer rows in our customer table, we still don't want to have to scan through all of them just looking for that particular customer ID. So the other benefit of defining a primary key is it will also generate an internal index within the database. Think of an index in a text book, we can look in the index and it'll help us jump quickly to a particular point in the book without having to leaf through every single page. And in a database, an index serves the same purpose. The DBMS will maintain an internal map of the data, so if later we come to it looking for customer ID 742956, it knows where to jump to get to that row in the table.

Foreign Keys and Referential Integrity

And one of the huge benefits of having unique primary keys to identify a single row in a table is we can then use that key from other tables as a reference. For example, let's say we have a customer with the unique customer ID 567. When creating a new order in our orders table, rather than copying and duplicating any data about the customer, which we don't want to do, we define a column in the order table for customer ID, and this column will be a reference from the order table back to the customer table. Now customer ID is unique in the customer table because it's the primary key. But over in the order table, it doesn't have to be unique. Now we could have several order rows with the same customer ID of 567, and that's because the same customer can create multiple orders, and that's okay because customer ID is not what makes them order

unique, that would be something like order ID, and that would be the primary key on the order table. So here, customer ID is still kind of a key, but in the order table, it's not the primary key, instead, it's called a foreign key. That means it's a key to a different table. And we tell the DBMS that this column is a foreign key so that it can check the validity of whatever data we put in that column. What that means is that DBMS won't let us create a new order row for our customer 724 if there is no customer of 724. It will check to make sure the reference is valid. From the other side of things, if we have a bunch of data, then the DBMS will stop us from doing something silly like deleting a customer in the customer table if that customer still has existing orders in the order table. The term we use for this is referential integrity. It's back to this idea that the data will be kept consistent within itself, always meaningful, even when multiple different tables are involved.

Database Design Stages and the Role of SQL

It's important to understand there are very separate stages when working with the database. There's all the upfront work of defining it and creating it, deciding on the different tables and columns and keys and relationships, there are a few different names you might hear for this work. We can use a loose term like database design, but another common term is data modeling. And if get to the point of actually describing every table, every column name, every column data type, every key, and every relationship, this is referred to as the schema for the database, the structure, the plan, the definition. Now the schema does not include any actual data, but it says what that data has to be. Now if we want to visually represent a database schema, we don't use that pseudo spreadsheet grid view because the schema doesn't include rows of data. Instead the schema view of a database design would show each table with their names, their columns inside those tables, and show the primary key for each one, as well as the foreign keys that create relationships between different tables. If you've done this work on a whiteboard or graphical application, and it doesn't actually exist in a DBMS yet, you might also hear it referred to as the logical schema. You may not have implemented this yet, but it is well defined, it's a plan, it's specific. And this is all work that has to be done before we can get to the very different stage of actually using the database. And once you've defined the table, there's really only four things you can do with it. You can create a new row in that table, you can read a row in that table, you can update a row and change a value, and you can delete a row. That's it. Every additional complexity is either doing several things at the same time or doing these things with multiple tables at the same time. And there is a acronym that can be applied to any option for data storage, not just databases, but it does apply here, CRUD. You can create, you can read, you can update, or you can delete. And with relational databases, we do these CRUD operations using a language called

SQL, or Structured Query Language. SQL is a little different, it's not a general purpose programming language. And what I mean by that is you can't make a web application in SQL or make a mobile app in SQL. You'd write those kinds of apps using a programming language like Java or Swift or Python or C#. But within those applications, you could then use SQL just for that part where you need to talk to a relational database. SQL is a very small, very focused, domain-specific language, meaning it's focused on one thing only, working with relational databases. And it's why you see SQL in so many database management system products, like MySQL, PostgreSQL, Microsoft SQL Server. Even products that don't have it in the name, like DB2, Oracle, and Teradata use SQL. Trivia sidebar moment, this language was originally called "Sequel", but was renamed S-Q-L back in the 1970s due to some trademark issues, but the "sequel" pronunciation stuck around with some people, with some organizations, with some products. So it's quite common to hear the popular Microsoft DBMS referred to as "sequel" server, although some people do call it S-Q-L server. It doesn't mean anything. It's simply a difference in common usage. End sidebar. Now I'm not going to try and teach you SQL in this course, but it is important to understand the part that it plays and why it is so ubiquitous in the world of relational databases.

Organizational Business Usage

Using Transactions

In the relational databases we've been focusing on, splitting data across multiple tables can help us avoid needlessly duplicated information, and it makes it easier to update one part of our data while leaving other parts completely untouched. But in any real world application, we won't always deal with just one table at a time, we'll often pull a bunch of information from several different tables as if they were all connected, which is called a table join, and a very common thing to do. We'll also need to make changes that affect multiple tables at once. Now this is more challenging when you consider that a database could be in use by hundreds or thousands or even hundreds of thousands of people simultaneously, and there is potential for conflict. Even if it's just two people, you and I, we might be trying to update the same row at the same time, or you're making several updates at once and I'm reading data from the database while you're still in the middle of making those changes. And to help with problems like these, most relational database

management systems have a built in feature called a Transaction. And the word transaction is applied to databases as a similar meaning to how we'd say it in real life. If you enter into a transaction with me to even just buy a sandwich, that interaction has at least two essential parts to it. You give me some money, I give you a sandwich. Now we're okay if neither of these things happen, we're okay if both of these things happen, what's not acceptable is if only one part happens. If you give me the money and I drop the sandwich on the floor, I don't say, oops, keep the money and expect you to walk away. No, the expected transaction did not happen. We need to return things to their initial state. You get your money back. Now when developing an application that uses a relational database, when a programmer realizes there is a task, a single unit of work, which needs to make changes to two or more tables at the same time, they can write some simple code to tell the database, these two or three or four things I'm about to do should be considered a transaction. If they all complete successfully, great, but if there's an issue with any of them, the data gets automatically rolled back to how it was before the transaction even began. And the programmer doesn't have to write the code to roll everything back, technique DBMS itself will take care of that. And the idea of a transaction applies to an incredible number of real world situations, so having this feature built into many relational database systems is one of the reasons they've been such a consistent part of computing.

ACID Qualities

There's an acronym often used around database transactions, ACID or A-C-I-D. And this is an not an esoteric, technical term that you'll only find deep in the database documentation. No, you'll see this ACID word even in things like feature comparison web pages and basic sales literature for a product. And it stands for Atomic, Consistent, Isolated, and Durable. And these aren't four different things, they are simply four perspectives, four things to consider about what makes a successful transaction. Now atomic is the original Greek definition of an atom, meaning a thing that cannot be cut, cannot be divided. It's the core idea that whether there are 2 or 3 or 10 different parts to a transaction, you can't split it up. You either do all of them or none of them. Consistent means the transaction must take the database from one valid state to another valid state. A transaction can't violate any of the other rules you've defined within the database, like what types of data are allowed in columns or having valid primary and foreign keys. Isolated means that while the database is in the middle of executing the different operations within a transaction, then no one else using the database will see any of the changes until they are all complete. No one will ever see the database halfway through a transaction. And durable means that once the DBMS confirms the transaction is complete, then everything about it is persisted

and saved. So if in the very next millisecond there was a massive power failure, when everything reboots we won't have lost that confirmed transaction. The idea of transactions and these ACID qualities is not functionality we need a programmer to plan and implement, if transactions are important to us then we'd expect to choose and use a database management system that already included this functionality, and where our programmers use the built in transactional features of the DBMS. Now it's true there are database products which don't support transactions because they focus on other things. We're going to talk about some of those products in a moment.

Limitations of Relational Databases

The relational database management systems have been such a consistent part of computing because the features they provide, like having strict controlled schemas and ACID transactions, work very well in a lot of general purpose business scenarios. But they're not ideal for every situation, and the same database feature that can be hugely beneficial in one scenario can actually be detrimental in another. I talked earlier about defining a schema for a database, describing specific tables and columns with explicit rules about the data that's allowed in each column. Now this has the benefit of enforcing a format for your data, you're only allowed to add new data that conforms to the schema and you know exactly what you're going to get out of it. And this is great if your data is inherently well defined and predictable, but if what you need is flexibility, if what you have is data that's unpredictable, that's volatile, well you don't always know what you'll get. Then a strict controlled schema isn't helping, it's going to get in the way. And generally speaking, once you've defined and implemented a classic relational database, and you have applications using that database, and you want to avoid making changes to the schema of it as much as you can because everything that uses it expects that fixed schema. So if you start adding and removing columns or changing table names or altering your primary and foreign keys, things are going to start breaking. There are many phrases to describe classic relational databases, but flexible is not one of them. Now for a long time that was okay. Yu go back 2 or 3 decades and a typical software project lifecycle may have been 12 to 18 months, or 2 years. If you're on that length of time, then changes to a database schema are predictable and controllable. But in today's world of Agile development where we're implementing and releasing new features and updates in cycles of weeks or even days, making continual changes to a database schema can become a real liability. And beyond that, a couple of decades ago, we simply didn't have the massive, massive datasets and real time web applications we routinely expect to deal with now. We didn't have the same expectations for constant availability and global deployment, to be able to take a database and replicate it across multiple international

data centers. And over the last several years, we've had multiple new database products intentionally created to deal with these situations that relational databases weren't really designed to deal with, like flexible schemas, massive scale, geographic distribution. So, let's see a few examples of those.

The Marketplace and Your Team

The Marketplace of Database Solutions

To understand the current marketplace of database solutions, let's begin with a simple split between several relational and non-relational options. In the relational category, I'll begin with a few of the well known players and I'll also begin with a few products in the non-relational category, MongoDB, Redis, Cassandra, and Neo4J. Now, all of the products in the relational category share the same basic vocabulary and concepts. They have tables, they have well-defined columns, primary keys and foreign keys, they use joins, they have ACID transactions, they use SQL as a language to work with the database. And skills here are loosely transferable. Once you know how to work with one of the RDBMS products, it is pretty simple to move to another. But that is not true about these other products. They're not just very different from relational databases, they're different from each other and the skills don't just carry over from one to the next. And the main way we gather these, and other products, into any kind of meaningful category is describing them more by what they are not rather than what they are. And the phrase that's often used is these are NoSQL databases. What's common across the is they don't use the classic relational database model. Generally speaking, they don't use tables. Generally speaking, they don't use SQL. Generally speaking, they don't support transactions. Generally speaking, they don't require a fixed schema. And I have to say generally speaking a lot because there are so many products here that there's a lot of little caveats of, well, if you want a fixed schema, you can kind of do that here, or well, some of these do have a similar idea to transactions. So, yes, there is a lot of variety here. But even though they're referred to as NoSQL, some of these products do allow SQL in certain contexts, so it's better to read this as "not only SQL" rather than NoSQL whatsoever. And within this rather wide and loose category of NoSQL solutions, there are a few meaningful subcategories, including key value stores, document databases, wide-column stores, and graph databases. And the products I'm showing here are just one example of each of these. There are

other products in each of these subcategories. Now I could do an entire executive briefing course on each of these subcategories. They are beyond the scope of what we can cover here. But, we can still explore the key question you would ask before drilling deeper into any of these products, why would we, or why does anyone choose a NoSQL solution over a relational database? Or vice versa? Now I'll explain in a moment why this isn't always a great question, but let's just go with it for now. First, speed. They are typically faster than the equivalent relational database. Because relationship database management systems have a lot of overhead to provide the features we enjoy about them, ACID transactions, enforcing schemas, checking referential integrity between tables, allowing multiple table joins on the fly, this can all have a hit on performance. So if you have a data store that just doesn't do any of that, it's going to be faster. Two, they often scale better. The more recent NoSQL solutions are built with the cloud-type infrastructure in mind. They are designed to be deployed onto multiple machines. They're often very fault tolerant, where one part of it failing doesn't bring the whole thing down, and they have easy and often automated options to scale out onto more hardware the moment that you need to. And this can be very important if you're expecting very large amounts of data. Three, they're often cheaper, at least directly. One significant point is that most of the NoSQL categories are open source with no required payment for installing and using. Now this, of course, doesn't mean zero cost of ownership. We still need machines to run them on, we need to pay for storage, we need people to administer and maintain them and to program against them, but it's certainly a factor when comparing to licensing costs of closed source commercial products, like Oracle, DB2, or SQL Server. So speed, scaling, cost, these can be very compelling arguments, but still, I said it's not always a great question to ask, should I choose a relational database or a NoSQL solution? Because it can be both. Don't fall into the trap of all or nothing. These days it is not unusual to have an architecture that uses two or more database products, both relational and non-relational. And not just within an organization, but in use by the same application. Sometimes this is called polyglot persistence. The idea that rather than try and select just one product and then try and work around any of its limitations, we will intentionally use multiple products, choosing each one where it's the best fit for that particular problem, that particular part of our system. A relational database for the part of our system where we have well-defined, predictable data, where transactions are important to us, where we find value in keeping the relationships between our tables. And we'll use a NoSQL option where we have unstructured information, where pure performance is more important than ACID transactions, and where we may need to scale quickly. And beyond the idea of having an approach that uses multiple products, there are also more and more database management systems that kind of cross the line, where the same product has features taken from both relational and non-relational approaches. Let's see a couple of those.

Multi-model and NewSQL Approaches

So you might wonder with this recent popularity of NoSQL databases, did the major relational vendors, like Oracle, Microsoft, and IBM, just sit back and watch calmly as their market share got eaten into? No, not at all. With the increased competition, which has been a good thing in a couple of ways, one is that large vendors now also offer some kind of NoSQL solution within their own product suite. For example, Oracle now have a NoSQL database product, IBM support multiple approaches, as do Microsoft, like Azure Cosmos DB, which is considered a multi-model DBMS. It can be used like a document database or like a graph database, these are both examples of NoSQL solutions, and it has global distribution features built in, but it also supports transactions, it supports using SQL as a query language. It's not identical to a conventional relational database, like Microsoft's own SQL Server, but it has many of the same features we find useful in those databases. And beyond that, there's also some new database products released that take a lot of inspiration from the NoSQL focus on performance and focus on scalability. But, without throwing out all the relational features that have been useful for so many years. Now these products, which include Google's Cloud Spanner and Apache Ignite, are sometimes referred to as NewSQL rather than NoSQL. So they do support SQL queries, they support ACID transactions, they generally follow the relational database model, but they have features oriented around scaling and speed that are much more likely NoSQL products. Now it remains to be seen whether this NewSQL terminology will stick, but the products themselves are a very welcome addition. Now beyond making decisions between the different products, we may also need to decide whether to install and manage our database management software on our own machines, on-premises, or use some kind of hosted or managed cloud-based solution. Most of the database products that began as only on-premises installable software are now available in some kind of hosted managed version. And some of the newer database products are only available if you're using a particular cloud platform, like Cosmos DB on Microsoft Azure, Cloud Spanner on Google's cloud platform, or Amazon Neptune on AWS. So if you still have to decide on a DBMS, then yes there is now an agony of choice. There are more options than ever and with more and more products starting to adopt a multi-model approach. One of the things you'll need to consider is not just the pure feature set of any database product or the cost of it, but also the skills on your team.

Building a Team: Roles and Skills

Now when you're thinking about the team, the different roles and responsibilities you need, do consider the different stages of working with the database. We have the initial data modeling and

database design, and here it's not just about the technical knowledge of one database product, it's vital to have that institutional business knowledge around what data you have and how it's going to be used. Now there may or may not also be the decision about what database management system is best for this problem. And then there's the creation and implementation of a database, which will require good technical knowledge of whatever DBMS you have. And once a database is implemented and up and running, there's the different skill set of programmers writing applications that use the database or that integrate it with other products. Now these programmers may or may not have been involved in the earliest stages, it's often helpful, but it's certainly not required. And then there's the ongoing and repeated administration tasks. This is not just a vague idea of maintenance, but the important continuous collection of specific database responsibilities. This is the database administrator, or DBA, role. Now if you're installing your database software on-premises, on your own machines, then the DBA is involved in every stage of this, from making the initial hardware and capacity decisions, installing and configuring the software, creating the database itself, setting up authentication and security, so the right people and the right applications are allowed access to the database, monitoring how much space it's taking up compared to our projections for it, monitoring the performance and response times, figuring out if we need to improve everything with new indexes or perhaps normalizing or de-normalizing a database, making sure we have the backups and disaster recovery in place, making sure all the security patches are applied, helping make any needed changes to the database. Now there's sometimes an illusion that if you use a cloud-based, managed platform, all this goes away. It doesn't. There are a few tasks that might disappear. For example, when using a managed database service, you shouldn't have to worry about always applying the latest security patches. But most everything else, performance, authentication, capacity monitoring, verifying backups, making any needed changes to a database, that's still all entirely up to somebody on our team. And there are also new concerns if we go to the cloud. Do we have the correct monitoring and reporting in place? So if anything happens, we're actually informed about it. Do we have the right policies and alerts so if the database goes down at 3:00 in the morning, we know who gets the phone call? Do we have the correct configuration to scale this database when it hits a certain size or response time? Are we actually getting the Service Level Agreement, or SLA, that we're paying for? And if the data we're hosting is subject to legal or regulatory requirements, like financial or healthcare data, does this service actually meet those requirements? All of this is still up to us. Now it's true that a cloud-based service can give us a lot of agility and flexibility to grow and often great features for improving availability and performance, but it still requires attention and skill. And if you are making decisions about using newer, more cutting edge database products, and also building a team, do consider the

availability of skills. It's reasonably easy to find people with years of SQL Server, DB2 or Oracle expertise, not so simple to find folks with substantial experience in Google Cloud Spanner or Amazon Neptune, at least not yet. But with that, I'm going to bring this executive briefing on databases to a close. We have quickly covered a lot of concepts, examples and terminology and I hope you found this useful. Whatever feedback you'd care to share, please do let us know. Tell us if there are things you'd like to see more of or less of. We read and pay attention to all of the feedback. Thanks for joining me. See you next time. We hope you enjoyed this course. If you're interested in more content for technical leaders, content we keep short and focused with the up-to-date information you need to be informed and make decisions, but without getting buried in the details, find other courses like this at plrsig. ht/exec.

Course author

Simon Allardice

Simon is a staff author at Pluralsight. With over three decades of software development experience, he's programmed in every discipline: from finance to transportation, nuclear reactors to game...

Course info

Level	Beginner
Rating	★★★★★ (123)
My rating	★★★★★
Duration	0h 48m
Released	1 Nov 2018

Share course