Big Data refers to data that traditional computing approaches can't deal with because the data is:

Too large = "High Volume"

Too complex = "High Variety"

Arriving too quickly = "High Velocity"

Gartner analyst Doug Laney produced this quote in 2001 and refined it in 2012, giving the "what, how and why" of Big Data. The "why" explains that Big Data is about getting VALUE out of your data.

Many articles that talk about Big Data only refer to the "What" of Big Data, notably the three Vs in the first part of the quote, "Big data is high volume, high velocity, and high variety information assets", which pose a challenge for companies to store, analyze and gain insight from this data. The three V's refer to:

Volume – the increasing amount of data – Terabytes, Petabytes, Exabytes
Variety – the increasing types of data - Text, photos, geolocation, sensor, binaries, etc.
Velocity - the increasing speed of data arriving

The How and the Why are essential in understanding a Big Data problem. The second part of this quote, "assets

requiring new forms of processing", and the third part, "enabling enhanced decision-making, insight discovery and process optimization" both refer to the how and why of Big Data: Why do we have this new technology why do we need these new tools.

The reason that companies want to engage in big data projects is to get additional insight out of their data. They want to gain insight into their customers behavior, to identify new products, and gain competitive advantage from their products to increase revenue. You'll see some examples of this as we go through the course. There are many different terms floating out there in ways of defining Big Data and I find this definition to be the best, the most descriptive.

Some journalists and bloggers also refer to "Veracity" of data (from the Latin word for Truth) – referring to the uncertainty of the quality of the data. Much data is poor quality, unreliable. Another V is "Voracity" – i.e., hungry. Other words beginning with 'V' are used in different sectors, for example "variability" and also value – which is a good way of describing the latter part of the quote.

Data Volume is particularly a problem when performing many forms of statistical analysis. For many statistical applications, it is necessary to read the whole dataset to do the analysis – it is not sufficient to read a sample, or a part of the data. Statistics is the basis of Data Science, a growing area of data analysis today. This is especially problematic for large data volumes, above tens of Terabytes, when using traditional systems.

The Velocity problem of Big Data is of particular concern to those businesses involved in web-scale applications. For example, those involved in processing social media information from millions of users, and projects concerned with the "Internet of Things" (IoT) – connected devices such as smart-phones, smart-meters, and even sensors in groceries or other goods. The latter has the potential to generate vast quantities of data at a data flow rate that is likely to overwhelm most traditional systems.

Variety encapsulates a number of different issues surrounding data, including the following:

- Much data today is not in a simple "tabular" form, which can be easily fit into relational database tables. Including Much "unstructured" data (i.e., there isn't a predefined "schema" that allows a computer system to un-ambiguously interpret the data). Turning unstructured data into structured data isn't difficult, but often involves human intervention to interpret ambiguities.
- In Relational Databases, the "schema" of the data is pre-defined into the database before data can be loaded. The way you store the data is influenced by what you want to do with it. This means the "schema" must be defined before storing data, and before interpreting the data. The schema cannot be changed to reflect new datasets, new use-cases, etc. In many Big Data projects, it is necessary to store the data before the schema is known. Many Big Data tools allow the analyst to build the "schema" into the application, to change it as often as the application needs to change (without affecting the underlying data) and even apply multiple different "Schemas" to the same dataset for different purposes.
- Finally, there are new data sources, multiple formats and different systems that need to be considered. More on this coming up.

"Schema on Read" is the big issue that the Relational Model doesn't handle. The idea of Schema on Read is to store it once, and not worry about the schema or structure of the data at write-time. We can work out what to do with the data at analysis time (which might be hours or years later). And it is also possible to apply different schema to the same data at read time for different projects.

There are different types of clustering methods:

- Connectivity-based (or hierarchical) clustering, e.g., single-linkage clustering, complete linkage clustering…
- Centroid-based clustering, e.g., K-means, Lloyds algorithm…
- Distribution-based clustering, e.g., Gaussian Mixture models
- Density-based clustering, e.g., DBSCAN
- Others

The obvious solution to these bottlenecks is to use lots of disks in parallel. Rather than attaching lots of disks to a single machine, we use lots of separate computers, connected by a fast network. We call this "Scaling Out"

In this module we will discuss some of the modern big data frameworks and architectures that are based around this idea of scaling out.

These big data frameworks are mainly build around two design patterns:

- A data lake design pattern (Built using Hadoop Apache Cassandra, Amazon S3, and even relational databases)
- A data warehouse design pattern (Built using relational databases e.g., Teradata)

Both of these designs can be built from one or multiple technologies. But:

- **Data Lakes are predominantly Hadoop**
- **Data Warehouses are almost always a database.**

Data Warehouse design patterns were formally established in 1990s and restated by Gartner Dec 2005. Only built using relational databases.

Subject oriented means there is a model of the business data and processes for tables and their relationships. So customer tables connect to customer transaction tables.
Integrated data means extensive data quality investments. It also means using a consistent data format. Format and content of dates, addresses, account types are agreed by the business user. This prevents male and female from being "M & F" in one table but "1 & 2" in another.
Nonvolatile means we don't change the data, we add new versions and retain the old information. **Persisted** means pulling together and integrating all data at query time is not sufficient. It must be pre-integrated and stored on disk.
Time variant means we store a new record for every change made, we don't over write the original. This is like the nonvolatile intent.
High concurrency and SLA levels are not in the original definitions. But they are the requirements and expectations of all customers. High concurrency means 100s or 1000s of users can access analytic data simultaneously. SLA levels include performance speed expectations and high availability uptime expectations.

Parsing Engine – Parsing Engine is responsible for receiving queries from the client and preparing an efficient execution plan. The responsibilities of parsing engine are:
Receive the SQL query from the client
Parse the SQL query check for syntax errors
Check if the user has required privilege against the objects used in the SQL query
Check if the objects used in the SQL actually exists
Prepare the execution plan to execute the SQL query and pass it to BYNET
Receives the results from the AMPs and send to the client

Message Passing Layer –Message Passing Layer, called the **BYNET**, is the networking layer in a Teradata system. It allows the communication between PE and AMP and also between the nodes. It receives the execution plan from Parsing Engine and sends to AMP. Similarly, it receives the results from the AMPs and sends to Parsing Engine.

Node – A node is the basic unit in a Teradata System. Each individual server in a Teradata system is referred to as a Node. A node consists of its own operating system, CPU, memory, own copy of Teradata RDBMS software and disk space. A cabinet consists of one or more Nodes. Currently Teradata's largest customer system is 50 Petabytes of storage and 4096 CPU cores. There are over 100 customers in the "Petabyte Club".

Access Module Processor (AMP) – AMPs are the parallel workers. AMP nodes usually have 128 to 256 disk drives or a dozen large capacity SSDs for high IO throughput. There are usually 20-40 AMPs per node for maximum parallelism. AMPs store the data in the disk array during data loading. They later execute the SQL plan from Parsing Engine, reading in the data, performing data conversions, aggregations, filtering, sorting and generating an answer set. Records from the tables are evenly distributed among the AMPs and across the disk arrays for the best performance. Each AMP is associated with a set of disks on which data is stored. Only that AMP can read/write data from the disks.

Cabinet - Consists of 1 or more nodes
Nodes up to 2048 nodes

- Linux operating system
- Largest: 256 nodes, 4096 CPUs, 55PB

Relational database

- Parsing engine nodes
- Access Module Processors (AMPs)
  - 20-40 AMPs/node

Storage

- Random access database pages

BYNET Interconnect

- Scalable Infiniband bandwidth

Submit code to run in parallel

Name node

- Cluster configuration, metadata
- Yarn: start/stop of batch jobs

Data nodes

- Mappers access, process data
- Reducers sort & summarize
- Runs Spark and other software

Hadoop Distributed File System

- Spreads files across all disks
- 64MB chunks
- Batch file scans

A more precise definition of **scalability** is familiar to users of the Teradata Database:

Expansion of processing power leads to proportional performance increases.
Overall throughput is sustained while adding additional users.
Data growth has a predictable impact on query times.
Because a shared nothing hardware configuration can minimize or eliminate the interference and overhead of resource sharing, the balance of disk, interconnect traffic, memory power, communication bandwidth, and processor strength can be maintained. And because the Teradata Database is designed around a shared nothing model as well, the software can scale linearly with the hardware

All servers yield a diminishing returns curve of performance as you add processors. This is not unique to any one server nor vendor. Competition for memory and I/O resources inside an operating system and RDBMS cause occasional “collisions” where two CPUs need access to the same data. One waits while the other proceeds. This is very much like a traffic jam where there are just so many spaces on the road for cars (requests). When this occurs, some productivity or scalability is lost.

Most SMPs have around a 15% degradation. This means that as you add CPUs, you lose approximately 15% of the total sum of all CPUs. Consequently, it is very common for 4 CPUs installed to yield something close to 3.5 CPUs of actual performance. The most advanced SMP server designs use exceptionally fast memory busses and highly tuned operating systems to do better. They may only degrade at 8-10%. Regardless, this means that a 32 CPU system may only deliver 20-25 CPUs of performance boost. A 64 CPU system may only deliver 32-40 CPUs of boost.

To circumvent this, we must partition workloads into isolated groups. This is done with server clusters. **Clusters can usually scale close to linearly forever.** They may only scale at 9-95% linearity but they do so continuously. Clusters do not exhibit the diminishing returns curve. So, if one node in the cluster handles 500GB of data, to get to 5000 GB of data management, you need 10 nodes.

Subject oriented. The data warehouse organizes records into subjects like 'customer', 'account summary', 'account transactions.' Data is not organized in a data lake, we just capture the files.

In an EDW, the data is **integrated.** That is customer records are matched to accounts, accounts to transactions using key fields. The data is normalized meaning we only have one format for dates, names & addresses, gender, and so on. So data from multiple applications is merged together providing the best and most understandable data to the users. With the data lake, this is done by the programmer when the data is read from the file. The programmer will deliver a result, but there is no guarantee the programmer will deliver consistent information compared to another programmer. One programmer may use M & F for gender, another displays 1 & 2.
The importance of **raw data** was mentioned before so don't forget: the data lake must keep the original file indefinitely

Relational databases can add new **datatypes** but it's a little harder when the data is semi-structured. So today, we don't find documents, videos, audio, and other ornery data layouts in relational systems.

Most databases are compliant with **ANSI Standard SQL 2011 or 2016**. It's a robust, complex language. SQL-on-Hadoop uses a subset of the ANSI 1992 standard. This makes it difficult for business intelligence and data integration ETL tools to access the data lake. There is a lot of missing functionality.

Fast queries and joins are a result of the **cost based optimizer** in the database. But with Hadoop, most all queries are a form of table scan: the system must read all or almost all the data blocks to find 5 or 10 records. When a table scan is needed, its very fast. But databases with random access only do table scans for 10% of the queries.

ACID consistency means that any update to the database is sacrosanct. For example, a transfer of funds from one bank account to another is a single transaction. Its complete in both accounts or its backed out. With Eventual consistency, the update to the first account can occur and the second one gets lost. Eventually it will be corrected but there is a period of time when the data is actually corrupt. This works OK with Facebook photos, not OK with money transactions.

The relational database foundation of a data warehouse is expected to support 100-500 **concurrent users**. Some Teradata sites boast over 1000 concurrent users. Hadoop is much less efficient, generally topping out at 10-20 concurrent batch jobs.

The data warehouse value to the corporation means its on lock down by a **governance committee** of users, DBAs, operations, and bosses. Changes to the system are carefully planned weeks or months in advance to prevent mistakes, downtime, and willy-nilly changes. With Hadoop, there is little to no governance. This has its downsides but it also provides flexibility. Data scientists and Java programmers want to change anything at anytime. And there are workloads where this is vital.

Data lakes

Hadoop is a DIY open source system. Many Hadoop tools are not well integrated with the Hadoop core and the total collection are not tested together. This is a programmers world and mostly DIY implementations. Think of it as building your own house or buying one already built.

Hadoop is a batch file system which means it processes data in bulk. Random access to specific records is slow. Yes, there are many SQL-on-Hadoop implementations but there is still no database to send the SQL to. And that is why SQL performance is poor. A Teradata system –as well as Oracle and IBM– can run 1000s of queries/second. During the most complex "joins" of 2-4 huge tables, a relational database optimizer can run it in seconds/minutes while Hadoop takes hours.

Hadoop SQL products –called Hive, Impala, or Presto– support limited SQL functions. Which means that BI and ETL tools have trouble using Hadoop data files. Often they cannot access the data. Performance is often 2 to 200 times slower on Hadoop. And the number of business users accessing the system is rarely more than 10 or 20.

When you have millions of files, there must be information on finding them, their sizes, the number of records, and where the file came from. This is largely incomplete in Hadoop meaning there is a higher chance the programmer will choose the wrong file to process.

Apache Spark is now the favored tool of many programmers since its easier to use than Hadoop. Similarly, all the HDFS data is migrating to cloud storage, especially to AWS S3 storage.

Security has been improving but it is still primitive. There are more holes and leaks than secure perimeters.

Data Warehouses Although ALL database vendors have added the highly popular JSON, XML, and Avro data formats to their SQL language, its not enough. There are many data formats that need to be added such as audio, video, email, etc.

While the cost per TB of storage costs a small premium versus a commodity Hadoop system, the cost of EDW software still remains a lot higher than 'free'. And don't be fooled that Hadoop is free. Once you download it, you must augment it with 10 commercial products and dozens of programmers.

Governance committees consist of DBAs, line of business, auditors, purchasing agents, and programmers. Their 1st priority is to protect the data warehouse from self inflicted failures or corruption. Thus they delay implementation of database changes for weeks or months. Its possible to change a relational table in five minutes, adding or deleting a column. But governance committees schedule such changes weeks or months in advance. This limits the agility and flexibility of programmers and data scientists. Sometimes this is good, sometimes bad.

Speaking of agility and flexibility, there is the discovery zone process. Discovery means finding the questions you need to ask. It means exploring data we've never seen before. Teradata Data Lab takes care of a lot of this. But most relational databases do not a Data Lab equivalent.

Gartner prescribes that about **80% of all analytics** will run through traditional data warehouses, marts, and cubes. For the most part, 80% of the analytic ecosystem is engaged in this kind of analysis. When you realize there are 4-5 data warehouses, 200-600 data marts, and assorted operational data stores and multidimensional cubes, the infrastructure investment is enormous for most companies. Teradata recommends consolidating data marts but that's a tale for another day.

Data federation is largely the data lake use case of fast ingest and unstructured data handling.

Similar to Data Federation, Gartner sees the **distributed process** as a data lake with schema-on-read data preparation workloads. This could be called big data integration or ETL on steroids.

The discovery zone stands alone. While the Hadoop community intensely wants to claim this role, its been the domain of tools like SAS and other data scientist technologies for decades. Indeed data science is done in Hadoop – and the data warehouse, and the data marts, and assorted data stores.

First, a data lake captures raw data. Extracting the useful data and applying refinements can be thought of ETL or data integration. The raw data is like iron ore mixed with tons of dirt. Imagine the iron ore smelter sifting through the dirt, and producing ingots of iron which are then converted into rolls of sheet metal. Thus, the data lake takes raw data and transforms it into its first new form for later use.

Next is the data warehouse. Data from the lake it passed to the data warehouse where it is further refined to a final state in the data products system. Data products are what the business user consumes with their applications or BI tools. We can think of this as taking the rolls of sheet metal an transforming them into a car body. The business users take the new data and combine it with the clean, integrated data to form a report, graphics, or a dashboard. This is the goal of the data manufacturing process.

Lastly, data R&D is the discovery zone. Data from the lake, the data warehouse, and numerous isolated sources are given to the data scientist. They are looking to invent something new, to come up with questions we didn't even know we needed answers to. This is the 5% area of data science and discovery that all corporations need to have. They work with raw data and refined data to understand the new analytic questions that will help the corporation.

In contrast, Spark was designed by the Berkeley AMPLab, not as an add-on, but a complete software stack. At the heart of Spark is the Spark Core. It contains the core processing. This engine was originally designed to use its own cluster OS, Mesos. However, Spark was also designed in a modular fashion, meaning it can now happily run on top of the traditional Hadoop components of YARN and HDFS. And while Spark was designed to use HDFS as its primary storage, the AMPLab also added an in-memory accelerator to that called Tachyon. And Berkeley didn't just work on the infrastructure of clusters. They also thought about the stack above the Spark Core and developed several libraries: Spark SQL, Spark Streaming, Spark ML. Spark is unlike Hadoop because all its components were designed to work together instead of simply evolving. Diagram https://amplab.cs.berkeley.edu/software/

ElasticSearch (engine)- Logstash(ingest)– Kibana(visualization): combination of open source Big Data platforms marketed by the company Elastic.co. This stack integrates with other tools, including Hadoop.

"By combining the massively popular Elasticsearch, Logstash, and Kibana (ELK/Elastic Stack), Elastic has created an end-to-end stack that delivers actionable insights in real time from almost any type of structured and unstructured data source. Built and supported by the engineers behind each of these open source products, the Elastic Stack makes searching and analyzing data easier than ever before.

Thousands of organizations worldwide use these products for an endless variety of business critical functions."

Documents are mostly text, but the format varies immensely. Documents often get captured in a data lake where parsing tools can extract the meaning data such as creation date, author, subject, and text body. Then using sentiment analysis algorithms, we can pick out the meaning and value in the document. In 2015, this was limited to the data lake. But Aster Database collected many algorithms that are now part of the Teradata Analytics Platform. So you can do text analysis in the data lake and the data warehouse.

Audio and visual data may seem esoteric. Less than 5% of companies analyze this kind of data today. And because of the file sizes, this kind of data is usually captured and processed in the data lake. Now think of some use cases. Surveillance video is being used to spot cashiers who do favors for friends by not scanning the bar code on pricey items. It's evidence. eBay analyzes merchant photographs to determine if the search text matches the image. If not, they send emails to vendors saying "the red dress is actually crimson. You can sell more if you categorize it right, including accessories that match a crimson dress."

Web pages were historically analyzed in relational databases. The data looks scrambled since it is really a collection of key-value pairs. I might have a key-value pair like "name=willcox" and "purchase=crimson dress". It turns out this is actually easier to parse and extract useful data than in documents. But because of data volume (as in 100MB per web page) these clickstreams are captured and parsed in the data lake. The other reason the data is kept there is A) we sometimes need to analyze the data in seconds and B) the data perishes in value in hours.

Sensor data is a date-time stamp plus and array of sensor values. There can be one or hundreds of measurements, usually in numeric format. Because of the high volume of data and arrival rate, this data is usually captured in the data lake. Data preparation and sensor data quality techniques can be used to refine sensor data in the data lake. About half of all analysis on sensor data is done in the data lake, aka Hadoop. But often the data is pulled into the data warehouse, aggregated, and combined with other data like supply chain inventory or customer purchase history for warranty tracking.

Key-value stores These records contain a search keys with a payload. It's essentially a design from the 1960s that started the mainframe era. The payload is considered a blob – it is an unknown structure except to the programmer who created it. The common programming technique is to put an identifier in the beginning of the blob to tell whoever reads the data "Expect format ABC in this blob". So the logic for what the data looks like is only found in the application program. There is very little metadata about key-value storage blobs.

Documents are mostly text, but the format varies immensely. Documents often get captured in a data lake where parsing tools can extract the meaning data such as creation date, author, subject, and text body. Then using sentiment analysis algorithms, we can pick out the meaning and value in the document. In 2015, this was limited to the data lake. But Aster Database collected many algorithms that are now part of the Teradata Analytics Platform. So you can do text analysis in the data lake and the data warehouse.

A *graph database* is generally built for use with transactional (OLTP) systems. Most graph databases allow for a single Btree index into the records. Within each record are pointers to related records. Like a relational secondary key, these pointers show relationships between records. The classic example is hereditary links. Susie is Cindy's Mother. Cindy is Katie's mother. So there are pointers from mothers to daughter records. Now anyone can ask the question "Who is Katie's grandmother?" and the software can inference the answer by chasing the pointers.

Column family stores are modeled on Google's **BigTable**. This technique is NOT a columnar database. Instead, it is a collection of name-value pairs, similar to the JSON structure. What is different is that you can have many name-value pairs with the same name in the same record. In essence, it's a simplification of the JSON format. Also, since there is no relational schema, there can be holes in the data for a given record. If the customer did not give us a phone number, nothing is stored in the records.

JSON is a classic example of a semi-structured data format. It may be hard to believe, but this bracket weary notation is the least complicated of all the NoSQL unstructured data types. One benefit is that people can easily read JSON and figure out what's going on. At least up until about 3 pages of text. But with Key-Value stores it's much harder to read more than half a page of text.

These data structures map into the schema-on-read concept. That is, the schema is embedded repeatedly in the data. For example "streetAddress" is a column name in a relational table. Notice that Ptype and Pnumber get repeated for each phone. Ultimately this is a data structure that chews up a lot of disk storage and memory. So why use it? Well, object oriented languages like Java, Ruby, and Python expect the data in this format. They have built in parsers to fetch the data elements within this format when the program asks for it. So its easy for programmers to work with. THAT's the main reason for the demand for 'unstructured data' and 'schema-on-read'.

Nodes and keys are almost identical to relational database records. Nodes have 'attributes' which is simply saying there are additional fields in the record.

Edges are the word for keys that link the two records together. So the "account number" would be in the account record and the customer record. Those fields are both edges.

RDF triples are a special data structure shrouded in complex words. There is a subject, a predicate, and an object in each data record. Imagine this statement: Paul's mother is named Lisa. Paul is the subject, mother is the predicate, and Lisa is the object or target of the sentence. Now, if we have a second record showing Lisa's mother is Julia, we make the inference that Paul's grandmother is Julia. Inferences can be extremely complicated such that only a computer can match up all the RDF triples to get an answer.

In 1970 SQL did not exist. IBM invented it in the late 1970s and it caught on with emerging relational database vendors. **By 1984**, knowing SQL was the hot ticket to a great career.

In the early 2000s, the rebellion began with Java programmers who – to this day – despise database administrators. The DBA controlled the database and wouldn't let the programmers do what they wanted, when they wanted. It didn't help that the object oriented languages like Java didn't work well with relational tables. So by 2009, he rebellion against SQL was growing fast among programmers.

2010. As the NoSQL products emerged, it became clear that IT divisions and users didn't want to learn YET another language for accessing their data. All the tools and skills that had built up around SQL for data integration, queries, design, and administration didn't work with the NoSQL products. So, with limited humility, we changed the phrase to mean "Not Only SQL".

2013=New SQL. Many NoSQL vendors began adding pieces of ANSI SQL 1992 clauses to their products to capture more sales and business partner tools. They didn't get very far since SQL was only an access language. It is not a relational database. But the marketing chants began to say 'New SQL' as if it were not only equivalent but newer and better. (Its not. No by a wide margin).

2015. So we come back to the need to know how to code in SQL. Even if your full time job is to work with the NoSQL database, you will need SQL skills and tools to do your job. Because all the clean curated data must be gotten from the data warehouse. But many NoSQL tools truly do not use SQL, inventing their own coding standards.

As Big Data systems evolved, the benefits of a relational interface were later realized for analysis of their data, and SQL-lite functions became available in products such as Hadoop Hive, Cloudera Kudu, and Apache Spark. So today, the term "NoSQL" has evolved to mean "Not only SQL" or "New SQL" emphasizing that such systems may include support of SQL.

Schema-on-read means we just store the data and clean it up later. Or maybe we don't clean it up at all. This has the advantage of fast data loading.

When an application program needs data, it fetches the raw data and begins parsing it into useful structures. This is the natural way Java and Python languages work with "objects" stored in the database. So if the data is "Object Oriented", the programmer is happy because its easy to work with. This is especially useful for transient data or data that is used by only 3-4 users. It's also ideal for data that arrives for the very first time. This approach prevents the data from being curated and integrated into a data model. The programmer has to do all that manually which means all too often it is not done correctly.

While all the NoSQL vendors claim great analytic performance, they are light years behind Teradata, IBM, Oracle, and others when it comes to analytics. NoSQL systems can do simple analytics and joins.

NoSQL programmers prefer relaxed governance: no committees or DBAs limiting their ability to change the data. This allows them the flexibility and agility to make changes as fast as the business demands them.

Schema-on-write . The Schema (tables and constraints of the columns) must be established in advance of
loading data. For data that is shared among many people, especially business users, this is perfect. This data is every day operational information that is known and cleansed prior to users ever seeing it. Cleansing may be overnight, hourly, or in rare cases every 5-10 minutes. Most important – it is integrated with other data structures. So if there is a lot of data being shared amongst employees and it is used daily, this is the only way to support those users.

RDBMS governance is strong meaning programmers cant put changes into the system at whim. This protects the 100s of business users from outages caused by a programming mistake. It also ensures the accuracy and cleanliness of the data. So strong governance is in opposition to programmer flexibility.

Schema-on-write is not always practical. In many Big Data applications, the schema of the data may change in real time. In others, it is necessary to load data into a system prior to any knowledge of its structure, allowing the data scientist to explore the raw data.

Document databases are built primarily using JSON which is much easier to read and use than XML. Each document is identified by a unique key, in this case Customer account number. We have simplified the bracket weary JSON structure because PowerPoint formatting won't hold enough information for this discussion. So imagine the so called documents on the left are JSON objects.

The JSON objects have a lookup key in the database called account. If we search on account number 101, we get Sarah's record, everything about Sarah, and a history of all her purchases. For high speed transaction systems this works out well most of the time.

On the right is the same data in 2 relational tables. It wasn't obvious in the JSON example that there are two important key fields in the data: namely customer account number and productID. With relational, the two tables are related on a common key – the account number.

Now look at the inquiry request for all persons who purchased ProductID=1897. With Document storage like JSON, the product ID is not the primary access key. This means the NoSQL software has to read every document in order to find all names and account numbers of people who bought that product.

With the RDBMS software, the better way is to index the productID table and fetch only the records for ProductID=1897. Now use this subset to index into the Accounts table and fetch the name and address of the people who bought those products.

NoSQL products have added secondary indexes and other techniques to cope with their inefficiencies. And it works great. Except when you have to join data from two types of documents and the data size is in the Terabyte range. At that point, document databases and key value stores slow down severely.