Topic 1.4: What are NoSQL Databases?

The term NoSQL was originally used for non-relational databases (literally those not using SQ), which also meant that data storage and retrieval was carried out using means other than the table structures used in relational databases. In fact, some NoSQL databases are completely unrestricted by a formal schema (data structure) definition.

While such databases existed as far back as the 1960s, it was not until the explosive growth of data storage requirements for web-based companies such as Amazon, Facebook and Google in the twenty-first century that these database acquired the "NoSQL" moniker.

Most of these databases were not only optimized for web use, but also designed to run on computer clusters (also known as cluster servers). A ***computer cluster (Links to an external site.)*** is a set of loosely or tightly connected computers that work together in such a way that they can be viewed as a single system. Data is typically distributed across the nodes (computers) in the cluster in such a way that queries for large amounts of data can be run in parallel on each node, thereby reducing query run times.

As the concept caught on, the definition of NoSQL mutated to include a broader spectrum of databases. Some of these databases include a query language similar to SQL, or a full-blown implementation of ISO/SQL standard SQL. (HP Vertica, discussed later in this topic, falls into this category.) While some claim that using SQL disqualifies a DBMS from calling itself NoSQL, that has not prevented others from applying the term to such databases. Add a bit of marketing spin from vendors eager to present their products as part of the leading edge in technology, and the term blurs even more. Currently there is no formal, wide-accepted definition of NoSQL.

There is no formal, widely-accepted definition of NoSQL.

Not Only SQL?

As a result of the ever-broadening definition of NoSQL, many interpret the term to mean “not only” SQL.While this underscores the notion that NoSQL is not intended to displace relational databases, it has the unfortunate side effect of including object-relational databases, such as Oracle and Postgres. Clearly, this is overly-broad, so we must search further for a workable definition.

Authors Pramod J. Sandalage and Martin Fowler suggest the term “polyglot persistence”. In this context,***polyglot*** refers to different data storage mechanisms; and ***persistence*** (a term borrowed from object-oriented databases) refers to the permanent storage of data. This term fits well because so many organizations use a mix of data storage technologies based on the nature of the data and the requirements for manipulation of the data.

For example, a large grocery chain would likely continue to use a mainstream relational DBMS for applications such as inventory management (tracking inventory levels in the grocery stores and automatically re-ordering when stock falls to a predefined threshold). However, the customer loyalty application would be an entirely different story. Imagine a requirement for using the history of everything each customer purchased over the past 3 years in order to develop buying habits by store, brand, package size, department, day of the week, time of day, to name a few potential dimensions. I order to meet the requirement, it is necessary to store three years of sales data down to the line item (every item scanned by every register in every store during the last three years). While relational databases are certainly capable of storing tables containing billions of rows, they cannot efficiently handle queries that must sift through tens of millions of rows or more to answer analysis questions such has "how often are milk and yogurt bought together?" and "what hour of the day has the highest number of sales of bananas?".

The Objectives for Using NoSQL Databases

The primary objective of using NoSQL technologies is to fill gaps in the capabilities of more traditional database technologies, including relational, object-oriented, and object-relational. Analyzing these gaps leads to a series of more specific objectives.

First, data structures can be selected that better match the needs of the application, which increases developer productivity. For example, some NoSQL databases organize data into documents, while others structure the data in columns instead of rows, and still others have no predefined structure whatsoever.

Second, as already mentioned, NoSQL databases are specifically designed to make efficient use of cluster computers. This not only improves scalability up to "big data" proportions, but it also achieves reasonable performance for queries that must parse massive amounts of data.

Third, concurrent update issues do not require the rigid controls imposed by relational databases, which were designed for processing transactions where immediate consistency of changes applied by transactions is essential. Instead, "eventual consistency" techniques can be used. For example, on cluster computers, an update applied to data on one node in the cluster does not need to be immediately applied to other nodes in the cluster than contain the same data. As long as the DBMS remembers that one copy of the data is inconsistent, it can prevent incorrect query results from being delivered in query results until such time as the inconsistent data is updated. This technique improves data availability because queries do not have to wait until the data is totally consistent before being allowed to proceed.

Objectives Not Met by NoSQL Databases

Most NoSQL solutions fall short of relational databases in s number of areas. As discussed in the next section of this topic, NoSQL is not intended as a replacement for all SQL (relational) databases, but rather as technology that can be used in situations where relational databases are not able to meet the application requirements.

First, ad-hoc queries capabilities in relational databases are more flexible compared with NoSQL queries. In part, this is because relational databases handle simpler data structures (tables with rows and columns). However, it also has to do with the type of data stored in traditional relational databases, which tends to have simpler data types (numbers, character strings and dates).

Second, NoSQL databases typically do not support transactions where a series of database queries can be bundled together in such a way that a failure of any of the queries causes all of the changes to be rolled back as though the transaction never occurred. Also, as already mentioned, NoSQL databases do not provide immediate data consistency.

Third, if data from multiple applications or multiple data sources must be consolidated into a common database, relational databases may be a better choice. NoSQL are best suited to loading each data source independently. Matching and merging records from different sources for loading into a NoSQL database, such as consolidating patient medical information from doctor's offices, hospitals, pharmacies, laboratories and other facilities into a comprehensive medical record, would require a lot of custom programming because the data would have to be consolidated before being loaded into the database. With relational systems, the data can be loaded into different tables provided a common foreign key was used to link the records by patient.

Fourth, if robust security and privacy controls are required, then relational databases are a better choice. Most NoSQL databases contain only the most basic access controls that verify your credentials before giving you access to the database. However once you have access to the database, you can get to all the data. Relational databases, as you will learn, have robust features that allow users to be given privileges by table and, with the use of views, subsets of the rows and columns contained in a table.

Using Both Relational (SQL) and NoSQL

Relational databases are still the most common for mainstream business transactions (e.g. order management, customer relationship management, payroll).

NoSQL can be used to augment relational databases for requirements that exceed the capabilities of relational databases in any of several ways.

First, when data volume exceeds the scale normally handled by relational databases, particularly when data grows to "big data" proportions (billions or trillions of records), the NoSQL is a good choice.

Second, NoSQL databases are better suited to unstructured data. For example, storage of social media data such as that found in Twitter, Facebook, Instagram, and Vine is better suited to NoSQL databases than to relational databases. Also, non-scalar data such as audio and video data formats is better suited to storage in NoSQL databases.

Third, data prepared for display on the web, particularly graphical presentations such as charts and graphs, may be easier to store and manage using a NoSQL solution.

Finally, high speed data collection may exceed a relational database's ability to consume and store the data. For example, if a cellular provider collected real-time circuit usage data of its entire network on a continual basis (every connection to every cell tower in the country, 24 hours a day, 7 days a week), it is unlikely that a relational database could keep up.

Categorization of NoSQL

There are many ways to categorize NoSQL databases. One common method is based on the way they organize data. In this topic, we will explore four such categories: key-value databases, document databases, columnar databases, and graph databases.

The examples in this topic are based on the Northwind example used earlier. Here is a diagram showing the files that would be used to load the data into our NoSQL data structures:

Customer File

Customer ID	Company Name	Contact First Name	Contact Last Name	Job Title	City	State
6	Company F	Francisco	Pérez-Olaeta	Purchasing Manager	Milwaukee	WI
26	Company Z	Run	Liu	Accounting Assistant	Miami	FL

Employee File

Employee ID	First Name	Last Name	Title
2	Andrew	Cencini	Vice President, Sales
5	Steven	Thrope	Sales Manager
9	Anne	Hellung-Larsen	Sales Representative

Product File

Product ID	Product Code	Product Name	Category	Quantity Per Unit	List Price
5	NWTO-5	Northwind Traders Olive Oil	Oil	36 boxes	$21.35
7	NWTDFN-7	Northwind Traders Dried Pears	Dried Fruit & Nuts	12 - 1 lb pkgs.	$30.00
40	NWTCM-40	Northwind Traders Crab Meat	Canned Meat	24 - 4 oz tins	$18.40
41	NWTSO-41	Northwind Traders Clam Chowder	Soups	12 - 12 oz cans	$9.65
48	NWTCA-48	Northwind Traders Chocolate	Candy	10 pkgs.	$12.75
51	NWTDFN-51	Northwind Traders Dried Apples	Dried Fruit & Nuts	50 - 300 g pkgs.	$53.00

Order File

Order ID	Customer ID	Employee ID	Order Date	Shipped Date	Shipping Fee
51	26	9	4/5/2006	4/5/2006	$60.00
56	6	2	4/3/2006	4/3/2006	$0.00
79	6	2	6/23/2006	6/23/2006	$0.00

Order Detail File

Order ID	Product ID	Unit Price	Quantity
51	5	$21.35	15
51	41	$9.65	21
51	40	$18.40	2
56	48	$12.75	20
79	7	$30.00	14
79	51	$53.00	8

Key-Value Databases

As the name implies, key-value databases (also known as key-value stores) store data values using a key to identify each record. The value that is stored with each key can be thought of as a blob (binary large object), and it is common for hierarchies of data to be collapsed into tree structures and formatted using XML, JSON, BSON, or a similar markup or formatting language. The result is a single value that contains a hierarchy of data collapsed into a single data value (a blob). The key-value DBMS knows nothing about the contents of the blob. The application that uses the DBMS must pick apart the blob in order to make sense of it. Said another way, the application must take on the burden of transforming the raw data into information useful to the business users of the application.

Popular key-value database products include Riak, Redis, Memcached DB, Berkeley DB, HamsterDB, Amazon DynamoDB (not open-source), and Project Voldemort (an open source implementation of DynamoDB).

Looking at the Northwind data, although there would likely be other key-value stores, the most obviously useful one would contain order information. The key for the orders key store would be Order ID and the tree structure might look like this:

Customer ID

Company Name

Employee ID

Order Date

Shipped Date

Shipping Fee

Product ID

Product Code

Product Name

Unit Price

Quantity

The last four attributes will repeat for each line item on the order (each record in the Order Detail File for the Order ID). You need not worry about learning JSON or XML coding for this course, but for the purpose of illustration, here is the JSON code for Order 51 that would be stored as the data value in the key-value pair.

```
{

"id": 51,

"customerId": 26,

"companyName": "Company Z",

"employeeId" :9,

"orderDate": "2006-04-05",

"shippedDate": "2006-04-05",

"shippingFee": 60.00,

"orderItems": [
 {
 "productId": 5,
 "productCode": "NWTO-5",
 "productName": "Northwind Traders Olive Oil",
 "unitPrice": 21.35,
 "quantity": 15
 },
 {
 "productId": 41,
 "productCode": "NWTSO-41",
 "productName": "Northwind Traders Clam Chowder",
 "unitPrice": 9.65,
 "quantity": 21
 },
 {
 "productId": 40,
 "productCode": "NWTCM-40",
 "productName": "Northwind Traders Crab Meat",
 "unitPrice": 18.40,
 "quantity": 2
 }
]
```

```
}
```

As you no doubt noticed, JSON coding is somewhat tedious and complex. However, once the JSON has been formed by the application, the work that the key-value DBMS must do is quite simple. The only operations it needs to support is adding a new key-value pair, updating (overwriting) the value for an existing key, and deleting a key (which also deletes the value). Here how the key-value DBMS would store the data:

Key-Value DBMS Data

KEY	VALUE
51	Order 51 Data (JSON)
56	Order 56 Data (JSON)
79	Order 79 Data (JSON)

Key-value databases are useful for the following types of applications:

- Shopping Cart Data, typically with the key being the User ID
- Session Data, with the Session ID as the key
- User Profiles, with the User Name as the key
- User Preferences, with the User Name as the key

However, key-value databases may not be suitable for these situations:

- When there is a requirement to correlate data stored with different sets of keys. For example, you cannot correlate shopping cart data with the session data and user preferences that were active when the cart record was written.
- When there is a need to wrap multiple key saves into a transaction. When a key-store operation fails, only that operation is undone – there is no provision to undo previous operations that may have been part of the same business transaction. For example, if a shopping application fails, you can delete the shopping cart, but you cannot automatically delete the shopping cart and the session data.
- When there is a requirement to handle sets of data within a single operation, such as updating all orders for a customer when the customer has defaulted on payments. Key-value database can retrieve only one key-value pair per option. There is no provision for queries that find all keys or values that meet some criteria, such as finding all users who have English as a preferred language.

Document Databases

Documents databases store and retrieve documents, which are typically formatted using JSON, XML, BSON, or a similar language. Like key-value databases, records are stored using a key and a value, with the value. Unlike name-value databases, however, document databases are capable of parsing document contents, and many of them offer commands for searching document contents.

While the documents in a given database typically have a common structure, the individual documents need not have identical structures. For example, some records may have parts of the hierarchical tree omitted. It is also possible for attributes to have different names in different records.

Popular document database products include MongoDB, CouchDB, Terrastore, OrientDB, RavenDB, and Lotus Notes.

Document databases are useful for the following types of applications:

- Event Logging, such as user login and updates of sales leads
- Content Management, such as user comments, blogs, user registrations, and user or customer profiles
- Web or Real-Time Analytics such as web click-streams and page views
- E-Commerce Applications that handle offering products or services and managing orders.

However, document databases may not be suitable for these situations:

- When requirements include complex transactions involving multiple documents. However, some products such as RavenDB support complex transactions
- When requirements call for queries against varying document structures. In particular, when the document structure varies from record to record, such as profiles with different data elements at different points in time, queries against the document parts that vary will not work well.

Columnar Databases

Columnar databases organize data using rows and columns much the way that relational databases do. However, the columns can be split into column groups (Cassandra uses the term column families), each containing one or more columns, and these column groups are stored independently. Each column group has a row key, which can be used to put column groups back together to form complete rows.

Popular column databases include Cassandra, Hbase, Hypertable, Amazon SimpleDB, and HP Vertica. Note, however, that some argue that Vertica does not fit the definition of NoSQL because it uses ISO/ANSI standard SQL with some language extension.

Many columnar DBMSs, including Cassandra, store columns as name-value pairs, which permits missing columns to be omitted. The DBMS recognizes missing columns within a row and automatically treats the column values as missing data.

For example, let's assume that Northwind Traders accesses customer contact information far less frequently than the rest of the customer data. By splitting what would be the customer table into two column groups, processing is more efficient when the only the data in one of the groups is required. The following diagram illustrates the two column groups for two rows of data. (Obviously, there would be a lot more rows of data in a live database.) Note that corresponding rows in the column groups share the same rowKey value, which allows the DBMS to put the column groups together

Customer Data Column Group

rowKey: 1	customerID: 6	companyName: Company F	City: Milwaukee	State: WI
rowKey: 2	customerID: 20	companyName: Company Z	City: Miami	State: FL

Customer Contact

rowKey: 1	contactFirstName: Francisco	contactLastName: Pérez-Olaeta	jobTitle: Purchasing Manager
rowKey: 2	contactFirstName: Run	contactLastName: Liu	jobTitle: Accounting Assistant

Columnar databases are useful for the following types of applications:

- Event Logging, such as user login and updates of sales leads
- Content Management, such as user comments, blogs, user registrations, and user or customer profiles
- HP Vertica has many other use cases because it supports SQL. As already mentioned, however, some people don't consider Vertica to be a NoSQL database because it uses SQL for all queries.

However, document databases may not be suitable for these situations:

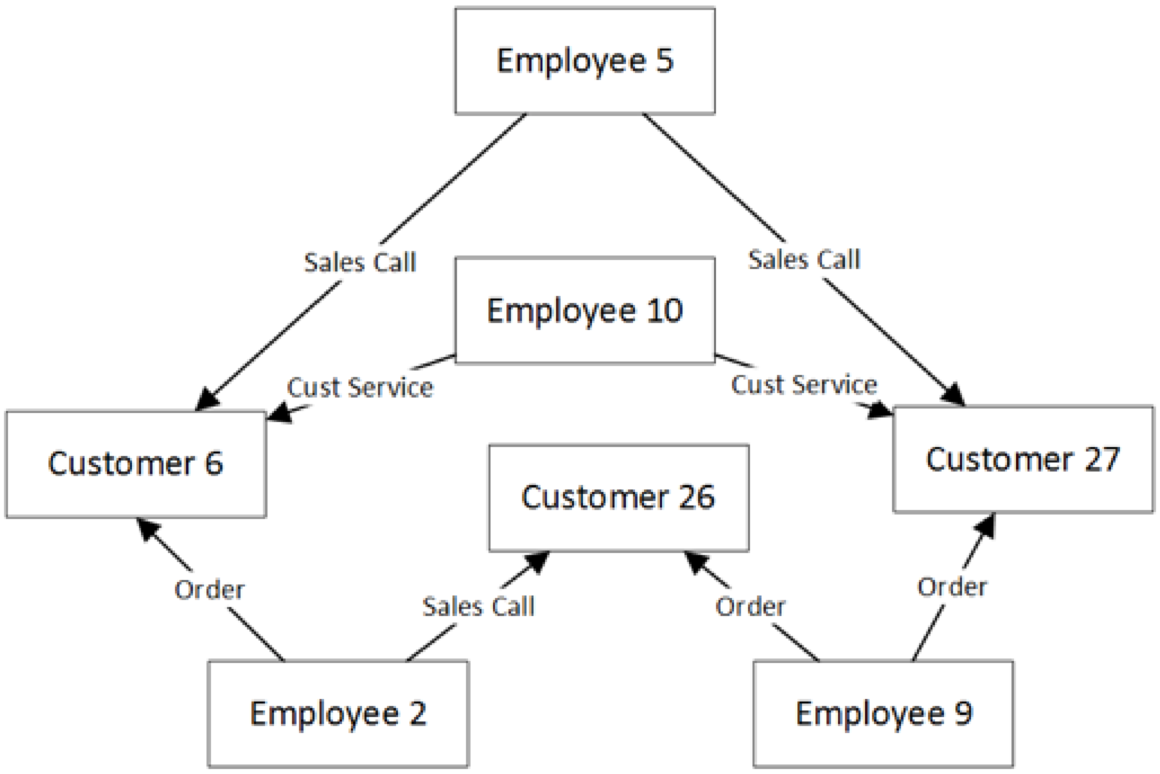
- Full ACID (Links to an external site.) transaction support for writes and reads
- Column family design changes are expensive in Cassandra. Therefore, it is not a great fit for early prototypes of applications where rapid changes are the norm. Cassandra is better suited to applications where the query patterns are well established because query pattern changes usually lead to column family design changes.

Graph Databases

Graph databases store entities as nodes and attributes as properties of the nodes. Relationships between nodes are stored as edges, and new edges may be created as needed. Unlike relational databases, adding edges (relationships) does not require adding properties (foreign key attributes) to nodes (entities). As you can see, graph databases are quite different compared to the other types of NoSQL databases presented in this topic. They address complaints that relational databases are not well suited to graphically displaying data relationships.

Popular graph database products include Neo4J, Infinite Graph, OrientDB and FlockDB.

As an example of a graph database, let's look at how we might implement a requirement from Northwind Traders to capture and store relationships between customers and employees, including placing orders, contacting customer service, and a sales call.



Graphs can be queried (traversed) by specifying desired properties and edges, such as “get all nodes where the customer is in California and participated in both a sales call and a customer service call”.

Graph databases are useful for the following types of applications:

- Connected data, such as showing diseases, treatments, and treatment side-effects.
- Routing, dispatch and location-based services, such as finding stores near a user’s current location which also have a particular product in stock.
- Recommendation engines, such as finding movie recommendations based on viewing history, or products that are usually bought together.

However, document databases may not be suitable for these situations:

- Any requirement to apply mass updates for all or a subset of entities (nodes) when attribute values change in the source data. Updating properties for existing nodes is generally not optimal.
- For very large volumes, operations that involve the entire graph may not perform well.