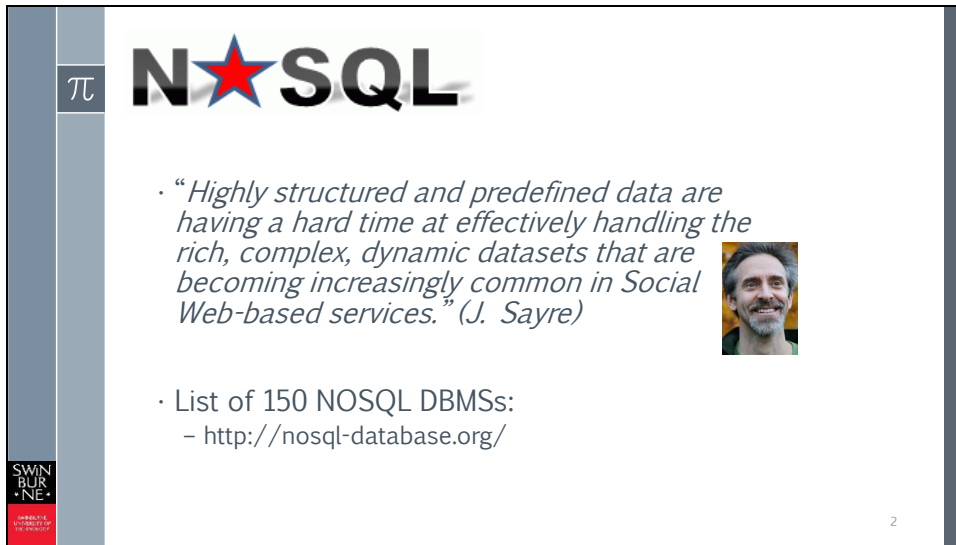


In this module we look into very recent database technologies which do not follow the relational concept.



The slide features a vertical grey bar on the left with a white circle containing the Greek letter  $\pi$  and a small red square with the text "SWINBURNE" and "UNIVERSITY OF TECHNOLOGY". The main content area has a white background with a dark blue border. At the top left is the "NoSQL" logo, where the "o" is a red star. Below the logo is a quote in italics: "Highly structured and predefined data are having a hard time at effectively handling the rich, complex, dynamic datasets that are becoming increasingly common in Social Web-based services." (J. Sayre). To the right of the quote is a small portrait of a man with a beard. Below the quote is a list item: "List of 150 NOSQL DBMSs:" followed by a link: "http://nosql-database.org/". A small number "2" is in the bottom right corner.

$\pi$  **NoSQL**

· *“Highly structured and predefined data are having a hard time at effectively handling the rich, complex, dynamic datasets that are becoming increasingly common in Social Web-based services.” (J. Sayre)*

· List of 150 NOSQL DBMSs:  
– <http://nosql-database.org/>

2

What is NoSQL? The acronym stands for not only SQL. This suggests that SQL is not going anywhere – it has its applications.

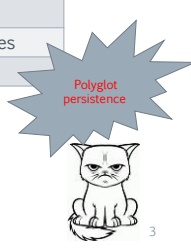
NoSQL has existed since around 2009. Google and Amazon faced problems with very large datasets and the performance limitations of relational databases. Relational DBMSs do not work very well in a clustered environment such as Cloud – they were often priced on a per-server basis; for example, you would be charged more if you used several instances of an Oracle DBMS on different servers for faster processing.


Although not all companies have to deal with as much information as Google or Amazon, companies have become interested in collecting data about the way users use social media or even games. To do this, they like to store content they find on the web. Most contemporary web documents do not have a predefined structure. Think of a Facebook page – no one can predict the structure of a newsfeed. NoSQL Databases do not require a strict schema for the data, and they were designed to deal with very large datasets and to use as many servers as available, making them ideal for Cloud and other clustered environments.

$\pi$

## What database does Facebook use?

MySQL	relational	page feeds
Haystack	object	pictures
Cassandra	column	personal messages
Hadoop	column	data analysis





Facebook serves at least **570 billion page views per month**.

More than **3 billion photos** are uploaded every month.

Facebook's systems serve **1.2 million photos per second**.

To cope with this volume, Facebook uses NoSQL – a relational and several non-relational databases. The MySQL database keeps the data for users and page feeds.

Accessing data from MySQL is slow, therefore Memcached is used as an intermediate in-memory storage. This means the FB pages can be built in memory from a multitude of fragments.

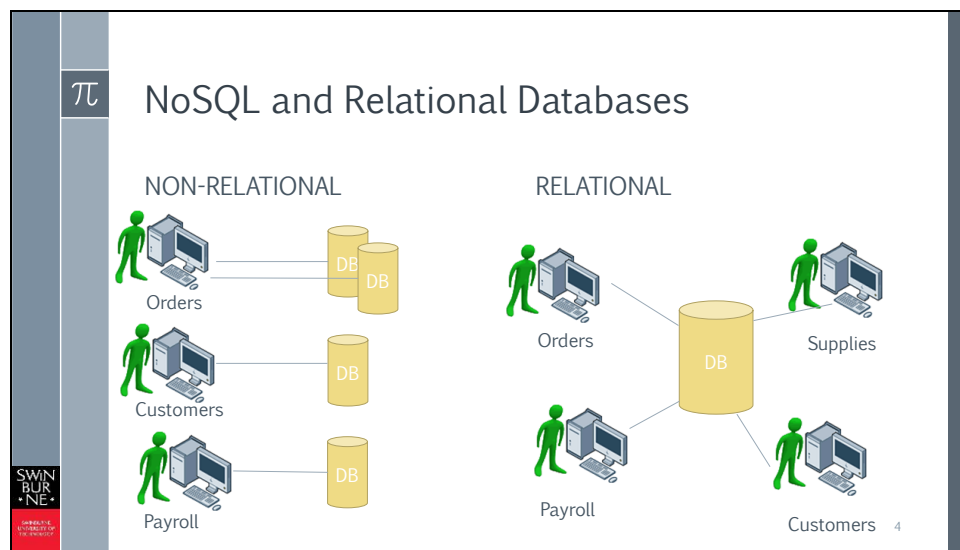
Haystack is an object database that stores all the pictures.

Cassandra is used to store the Inboxes (personal messages)

Scribe is a scalable logging system developed by FB

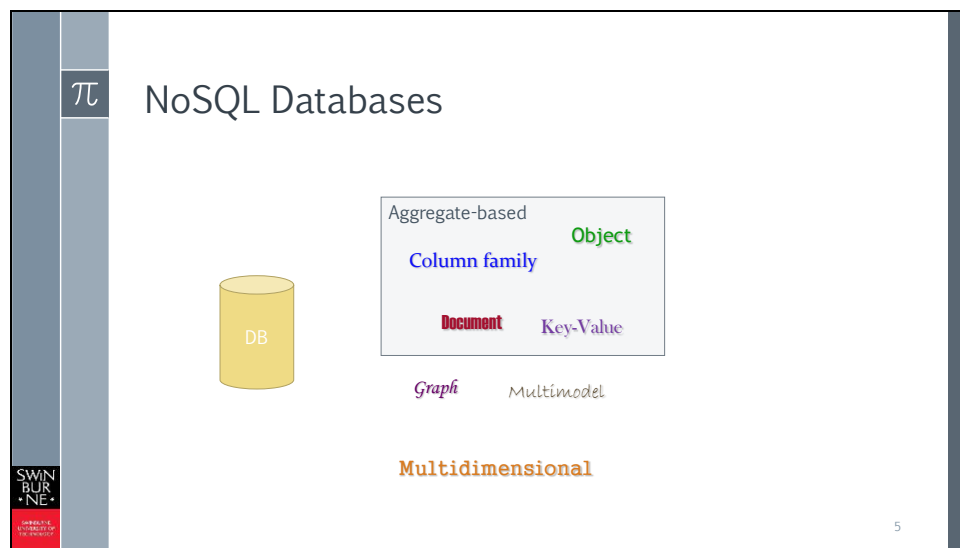
Hadoop is another column database, which is used for data analysis. Another product involved is Hive - a data warehousing facility which works with to Hadoop to enable SQL querying.

Such a diverse data storage model is known as polyglot persistence.



Relational databases are designed to be integration databases – central data stores for applications that work on related entities. When a customer is updated, the customer's new data is instantly available to all applications that use the same data store, because relational databases like to keep a single copy of each entry in a consistent state. But if all applications compete for this one customer entry, some of them may have to wait, and the database takes a long time to respond. In the relational world, distributed models are often set up to remedy this. Replication is used to maintain several copies of the same data. All copies can be made available to the outside world. Setting up replication can be very difficult if the expectation is that the data remains consistent.

In the world of NoSQL, databases are assumed to be application-specific. Each application maintains its own data, and we accept that there may be overlap between the data stores. The same customer may exist in several databases, and the entries may not be consistent with each other. Non-relational DBMSs can easily be deployed to a cluster of computers, because they were specifically designed for such environments.



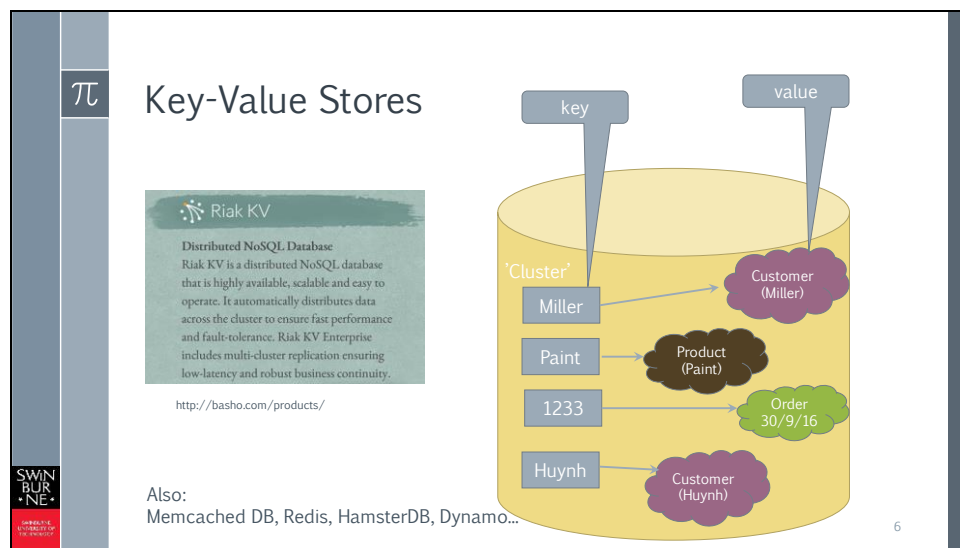
A great number of different data structures have been developed by different NoSQL products. Graph and multidimensional databases focus on the relationships between entities, which can be numerous and complex.

The most popular NoSQL databases are aggregate-based. An aggregate is an entity that consists of several entities that would probably be stored in different tables in a relational database.

The differences between column family stores, key value and document stores are not very clear-cut. Key-value stores can only retrieve entities by key. If we add an id field to a document in the document store, it becomes similar to key-value stores, in that we can also retrieve this document by its key. Column databases also have keys for the entities they store. You can picture the id fields in column databases like primary keys in a relational table. Column stores also have attribute names which are very similar to attribute or column names in relational databases.

One feature that all of them have in common is that they do not enforce a rigid schema. If we sell paints and lawn mowers, we can specify the viscosity and colour of the paint, its waterproofing abilities and time required to dry, and we can also note that the lawn mower comes with a 2-year guarantee and an electric engine. Such details are hard to accommodate in relational databases because with such different products, most attributes would be different, and it would be very difficult to accommodate them in a common Product table. In NoSQL, we can create a collection of aggregates and add both the lawn mower and the paint. Each aggregate can have its own attributes.





Key value stores use hashing to store an entity as a blob defined by its key. Hashing uses a mathematical function to transform the key into a number that is used as a memory address. Because this number points straight to the value to retrieve, hash maps are very quick to search, but only if we search by primary key. Any other part of the entity cannot be used as a search criterion. Some key-value stores, however, have more functionality and let you store objects by type and then you can search by the object's content.

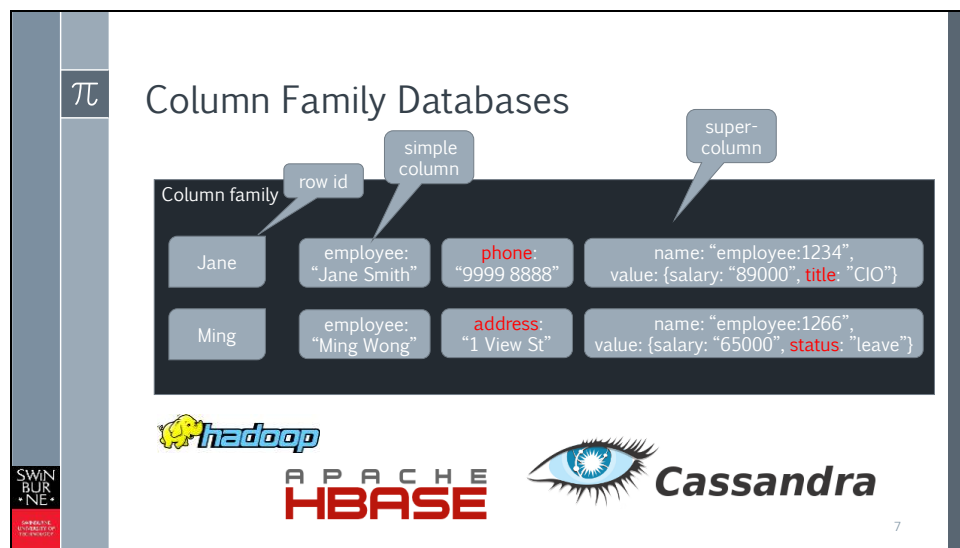
Basic key-value stores put the data into buckets regardless of type. Unlike document and column family databases, they don't suggest you distinguish between types of objects. They can all go in the same bucket. However, having different buckets helps organise the data and key-value databases give you the option of having a number of buckets each with a certain type of content. So you can use a bucket like you would a table in the relational model.

Most key-value stores don't let you query for a list of keys, so if you forget the key to a value, you can't get the entity back. In most products, you can't search the entity by except by key, and relationships between values can't be constructed. Therefore key-value data stores are often used to store temporary data such as shopping carts and session data.

Transaction support is generally limited, but varies from product to product.





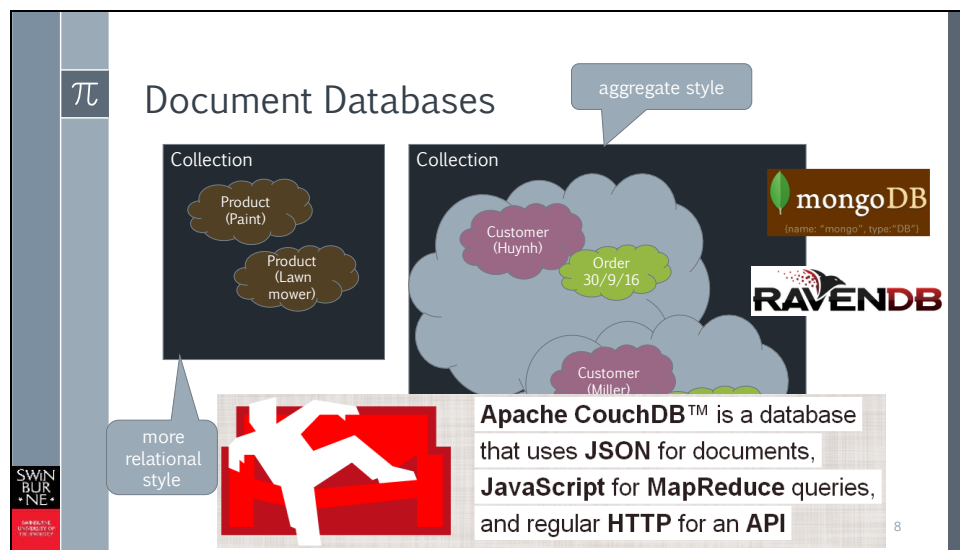


Column family stores group data into entities that look like row data in a relational database. Unlike in the relational model, for each row there can be different columns. The columns of all rows form the column family.

The example shows a column family as implemented in Cassandra. The first names in the leftmost box are the row keys that define the entity. Each row has a number of name-value pairs. The name is the column name and the value is the actual data. When we have a simple column, only the name of the column and the data is mentioned. We can also have supercolumns, which have subcolumns – meaning we can have nested columns. For supercolumns, you have to mention the words name and value to tell the DBMS which is which. For Cassandra, we use JSON notation to do this.

Supercolumns are a handy tool to group related data, but the data of the entire supercolumn has to be fetched into memory when one of the subcolumns is needed. Therefore columns should only be grouped if they are mostly accessed together.

Cassandra has a sophisticated write model to update data that is designed for performance. It also has its own SQL-like query language, CQL.



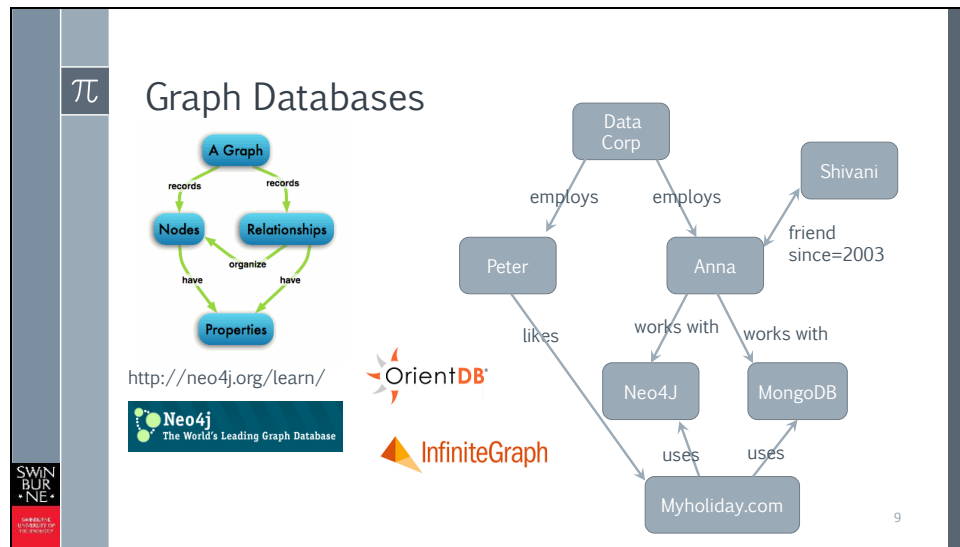
Document databases store collections of similar data. The entities stored in a collection can have relationships to other entities, which makes the design similar to a relational model. The entities can also be stored as larger aggregates. Aggregates are larger entities that include smaller entities, which, in a relational model, would be accommodated in separate tables.

The entities are described in JSON, BSON or XML format and then stored in a collection. A collection is the document equivalent of a relation. As you know, JSON and XML have a tree structure, so documents that use these formats have an implicit schema, but you also know by now that these formats are semi-structured and do not necessarily define every small piece of data in them. What's more, inserting such a JSON document into a collection doesn't mean it has to have exactly the same tree structure as another document in the same collection. The DBMS does not enforce the schema. If you add another entity with a different structure to the same collection, the DBMS won't complain.

Transactions generally cannot span more than one database operation. If you have to update data in two collections, you need two operations – therefore there generally is no possibility of a transaction when more than one collection is involved.

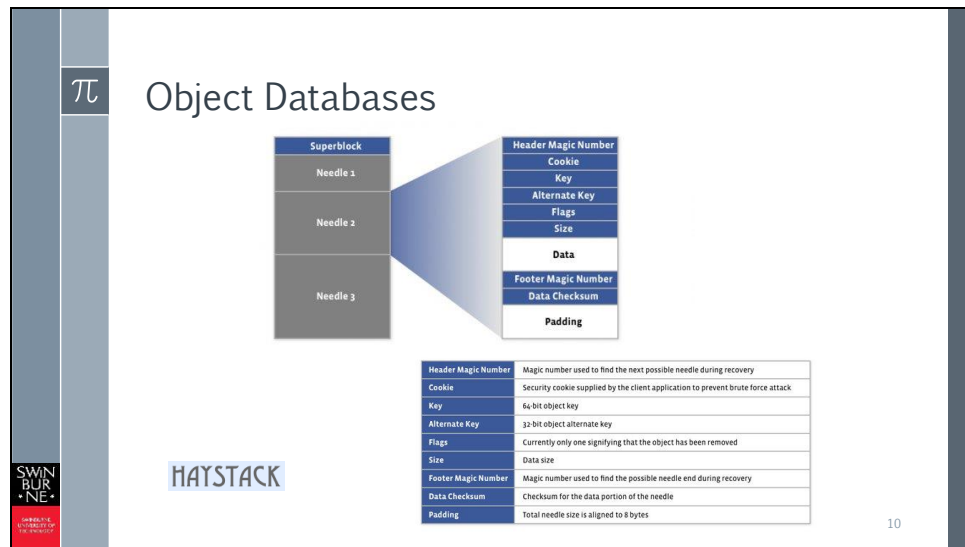
An exception to this rule is RavenDB. If you need transaction support with a document database, investigate this.



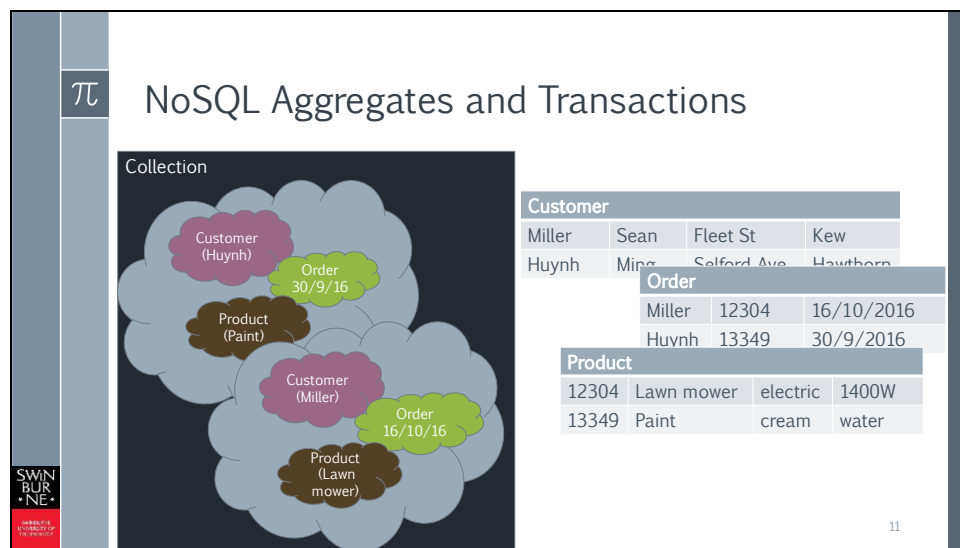


Graph databases are for storing entities and their relationships. The relationships are directed and have a meaning. They work like joins in a relational database; traversing a relationship leads to another entity. In relational DBMSs, joins are slow and to be avoided when performance is important. Graph databases are optimised for traversing relationships. In graph databases, entities can have any number of relationships, and each entity can have different relationships. Relationships are important, and properties can be added to them to describe them in more detail.

Graph databases are often used for applications that make suggestions of related products on web sites ("People who bought this product also bought.."). Because every node can link to every other node, graph databases make the use of clustered servers harder, although for example InfiniteGraph supports distributing the nodes (entities) over a cluster of servers. Neo4J supports transactions fully – we say it is ACID compliant.



Haystack is a simple object database. Like a log file, it stores all entries one after the other. The entries are called needles and occupy one of two files, the actual store file. There is also an index file that points to each entry in the store file. The index contains pointers to the actual data, information about the size of an entry and whether it has been deleted.

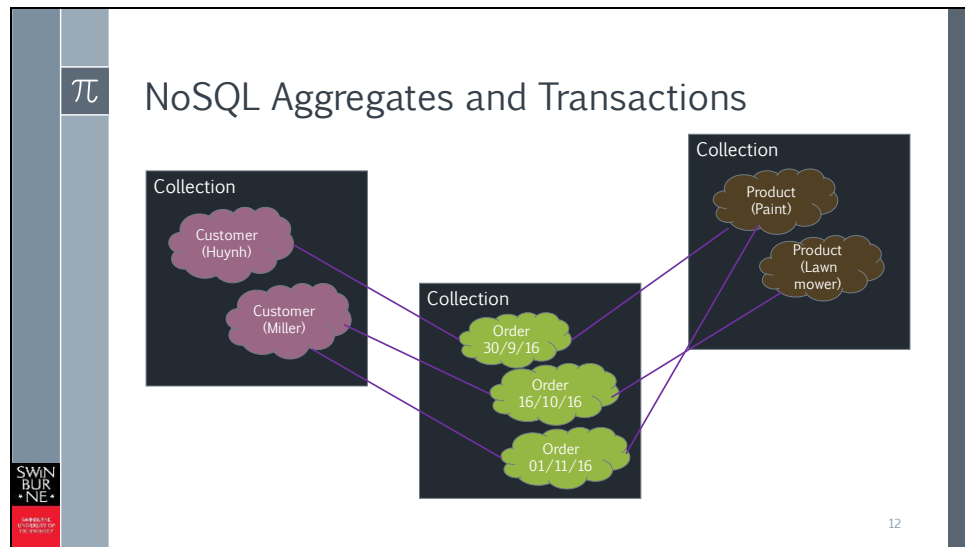


Relational databases define relations by unique identifiers, and there is little choice in defining the entities if we want to apply a proper relational model. In NoSQL we have great freedom in designing our collections. We can decide to include the customer, order and product data all within one entity, so they are combined and stored in the same location for fast retrieval. There is no need to search for the customer, order and product separately to perform a join. They are already combined in an aggregate object. This does not mean that you cannot bring the customer entity on its own without including order and product. Parts of an entity can be retrieved without problems. But you have to be aware that there are several copies of the same data. The same product has likely been sold many times. When it is stored in combination with the order, there is a new product entry for each order, even if the order is about the same product.

Updating a single entity is transactional, because the aggregate is the smallest entity we can work on.

How much data should be combined into one entity? This depends on how you use your data. If your application processes mostly orders and you don't sell the same products often, it is meaningful to combine customer, order and products. You have to be aware that some data is repeated in this design. But you do not have to store everything about an entity every time. Suppose the engine of the lawn mower is not interesting to one customer, you may choose not to mention it on that customer's order, and so you don't have to store the information either. The same product can be stored differently in different orders.



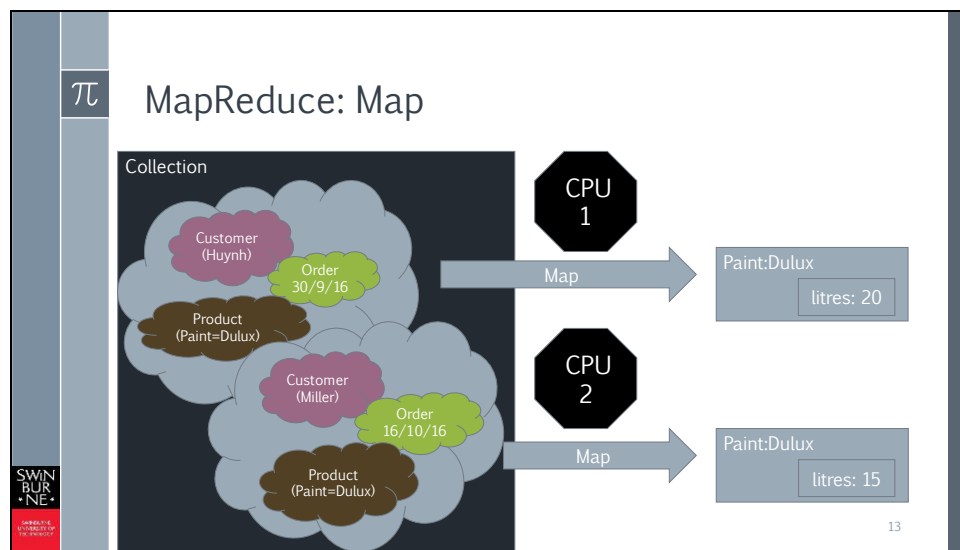


If you decide to separate the entities customer, order and product into different collections, NoSQL databases let you model relationships between these entities. But you have to be aware there is no transaction support for entities that are not stored in the same collection. There is no way of ensuring the order and product are updated in the same transaction in the model shown here.

When we have a database setup across a cluster of servers, in this model we are not guaranteed that the collections are stored on the same server. We have to make a judgement call whether we have to co-locate them.

Note that in this model, there is no duplication – we can reference the same Paint product for several orders as we do in the relational model.





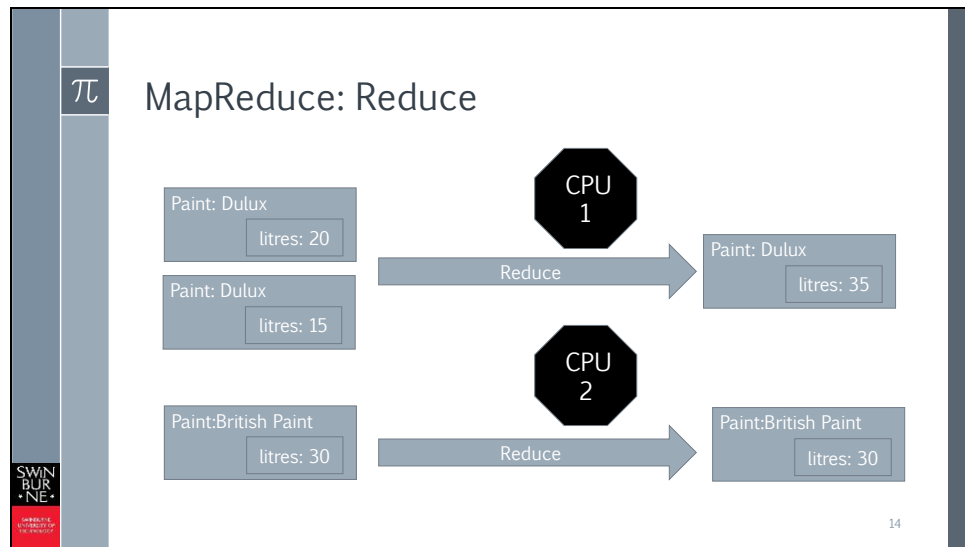
MapReduce is a way of subdividing a computation that uses many resources so that the computation can be assigned to many processors that work in parallel. Because one of the advantages of NoSQL databases is that they can easily be distributed over a cluster of servers, MapReduce is an important part of NoSQL database performance. The database entries are typically hosted on many servers, and the servers have to collaborate to retrieve the data, if the query asks for a listing of entries, a summation, or other calculation that involves several entries.

Because many NoSQL databases are aggregate-based, finding information from parts of the aggregates is time-consuming. MapReduce helps with this: It subdivides the job into subtasks and then collects the answer.

Suppose we have stored orders, order entries and customers as an aggregate in a single collection. Now we would like to find out how many litres of a certain type of paint we have sold. So we have to go through all the aggregates in the collection and extract the ones that include this particular paint.

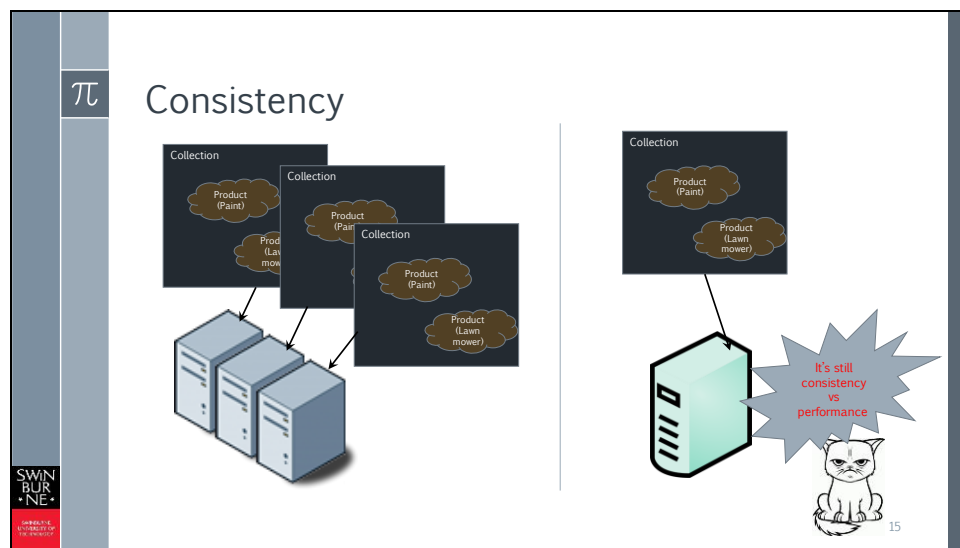
The map procedure splits the task of extracting all paint entries from the aggregates into subprocesses that can be executed in parallel. Each map subprocess accesses one order aggregate and returns the required data as a key-value pair.





When a mapping process has finished, its result is sent to the Reduce function. In the basic MapReduce implementation, only one Reduce function collects all the results. When there is only one key we are looking for (like the Dulux paint), this may be fast enough.

But if we are looking for the information for several products, the Reduce function can be partitioned. In this way, one reduce function can look after one or a few keys that it collects, and all Reduce function partitions can work in parallel.



In the NoSQL world, database instances are often replicated over several units of hardware. Because NoSQL was designed for large data sets and high performance, distributing the database to multiple nodes is a common strategy. Replication means copying the same data to several servers. This helps distribute the query load. Queries are often directed to the server that is geographically close.

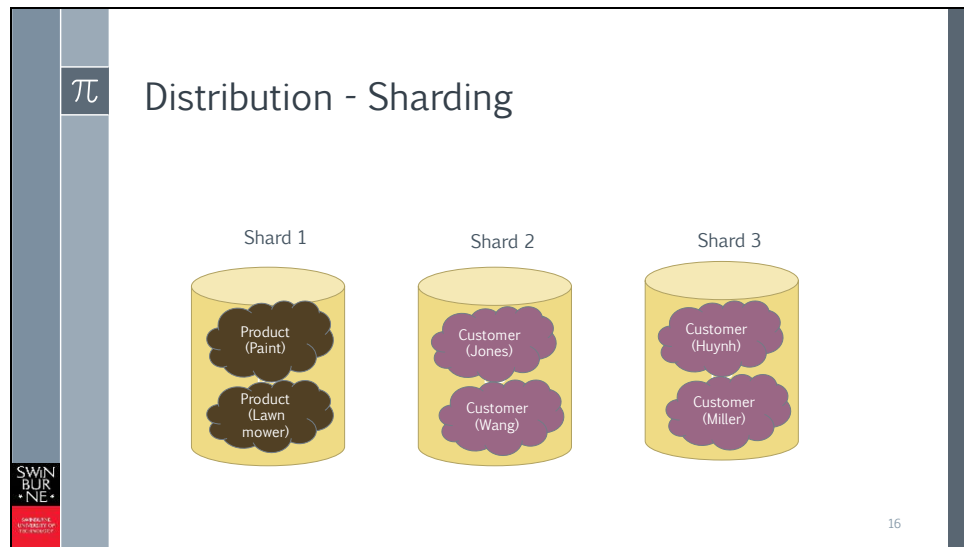
The problem this causes is write consistency – every update of a collection has to be repeated for every copy. There are several strategies with this – the DBMS can be configured to update all replicas before the update is considered complete. It can also be configured to update just one replica before the update is considered complete. A common strategy is to choose a majority of copies for update. This means that once two out of three possible replicas are updated, the process is considered complete, and the third replica is updated done ‘whenever the DBMS has time’.

Naturally, how many instances are updated immediately has an effect on durability – if a server fails and the update is lost, is there another copy that has the update?

The more copies are updated immediately, the lower the performance.

Naturally, if you only have one copy of each collection, replica consistency is not a problem. Durability then only depends on how often you back up the data.

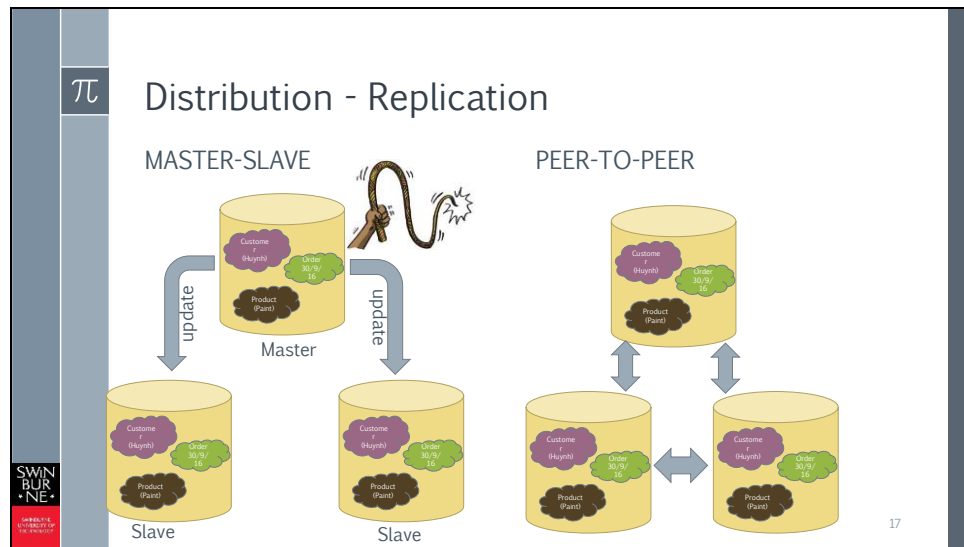




Sharding is a way of subdividing the database so that the load of the data processing tasks are distributed between servers. The challenge for the database administrator is to subdivide the data so that the load is even. Another challenge is to subdivide the data in such a way that only one shard needs to be accessed for any read or write operation. Aggregates help with this – they already bundle data that is frequently used in combination.

Sharding can be done horizontally (entries of the same collection are stored on different shards) or vertically (different collections are stored on different shards). In this example, there is vertical sharding between shard 1 and shard 2, and horizontal sharding between shard 2 and shard 3.

Sharding does not help with durability, or resilience – if one shard fails, its data is unavailable.



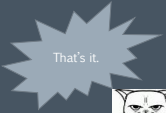


In Master-Slave replication, the Master receives all the updates. A replication process updates the slaves. The slaves are read-only. This is particularly useful when there are many reads of the database and relatively few updates.

It is also a resilient setup – many DBMSs can appoint a new Master automatically when the Master fails. This means we have a hot backup – the slaves are backup nodes that are always current and in the case of a failure, they can be appointed as new Master.

When there are many write operations, a peer-to-peer setup may be more useful. Each node can receive a write operation, and the replication process copies the data to the other peers. The only difficulty is consistency – if the same data is changed on two replicas, a conflict resolution mechanism is needed.

## Summary



- › Non-relational databases have become popular for their performance and ability to accommodate large amounts of data.
- › Polyglot persistence uses many NoSQL databases to store different parts of the application data.
- › Aggregate-based databases embrace data duplication for quick access.
- › NoSQL databases are easily set up in a distributed fashion.
- › Sharding and replication are used to enhance performance or, in the case of replication, to ensure durability.
- › MapReduce is often used to execute computations on large datasets in parallel.
- › Support for transactions is often limited and inconsistencies have to be tolerated.

18

Here are the most important points discussed in this module. You may want to stop the recording to have a read through them. When you are ready, start the quiz about this module.