DATABASE DESIGN & IMPLEMENTATION

ICT Skills

Objectives

- Column Family
- Graph
- Distribution Models
- Consistency
- Polygot Persistence
- Beyond NoSQL

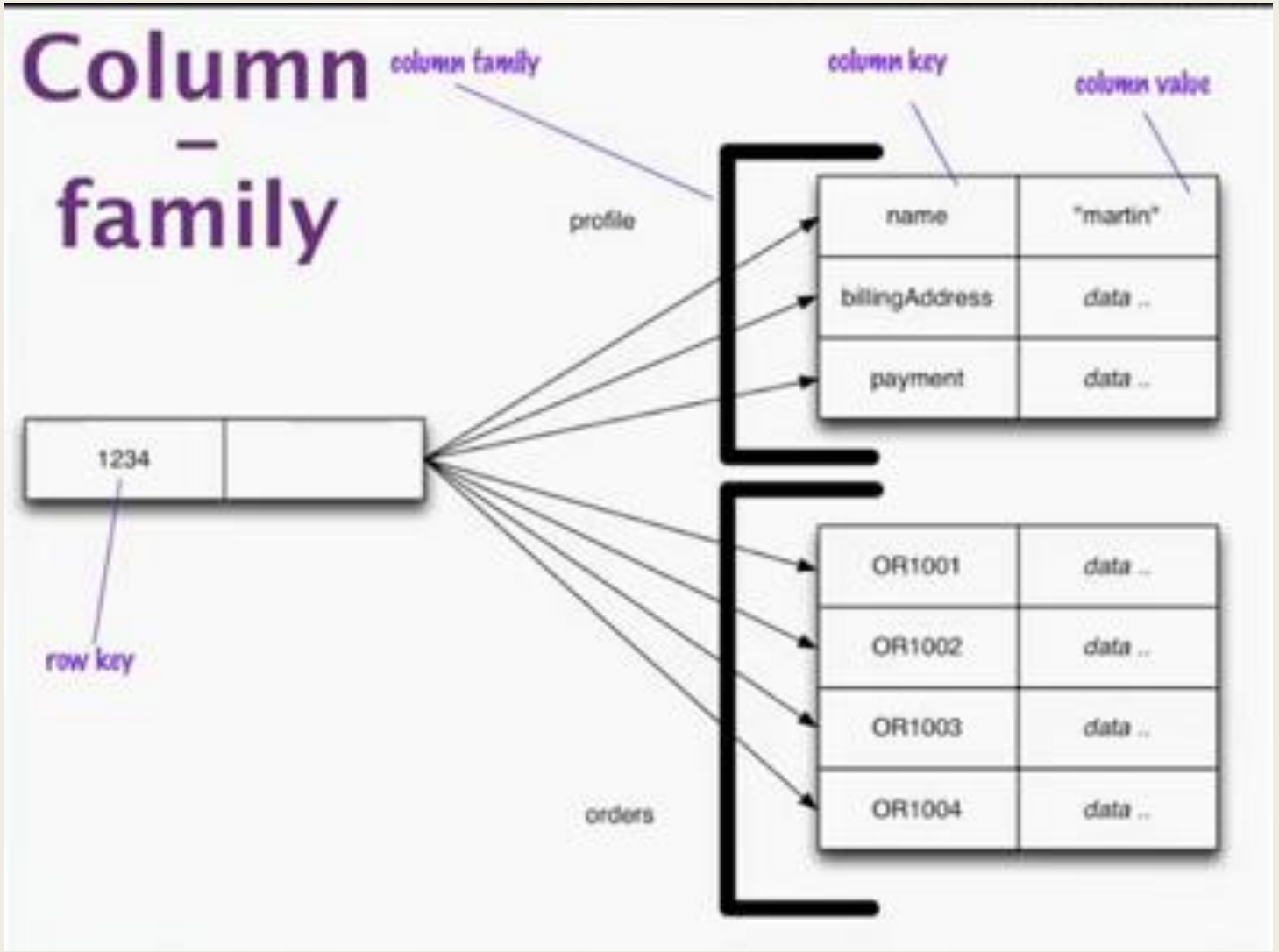
Column-Family Stores

- One of the early influencers of NoSQL databases was Google's Big Table.
- Databases with a BigTable type data model are often referred to as column stores, but that name has been around for long time. Most databases have a row as a unit of storage. However, there are many scenarios where writes are rare, but you often need to read a few columns for many rows at once. In this situation it is better to store groups of columns for all rows as the basic storage unit – which is why these databases are called column stores.
- BigTable and its relatives follow this notion of storing groups of columns together and abandoning the relational model and SQL.

Column-Family Stores

- Column family databases organise their columns into column families. Each column has to be part of a single column family, and the column acts as unit for access, with the assumption that data for a particular column family will be usually accessed together.

- Data is stored with keys that are linked to groups of columns for example a column family that stores customer details, another that stores orders for that customer, etc.
- Everything about one order is stored together in this one group of columns.



Column-Family Stores

- Column family is like a two level aggregate structure. The first key is a map of more detailed values. These second level values are referred to as columns.
- As well as accessing the row as a whole, operations also allow picking out a particular column, so to get a particular customer's name from the previous example you could do something like get('1234','name')
- Column-family databases organise their columns into column families. Each column has to be part of a single column family, and the column acts as unit for access, with the assumption that data for particular column family will be usually accessed together.
- Everything about one order is stored together in this one group of columns

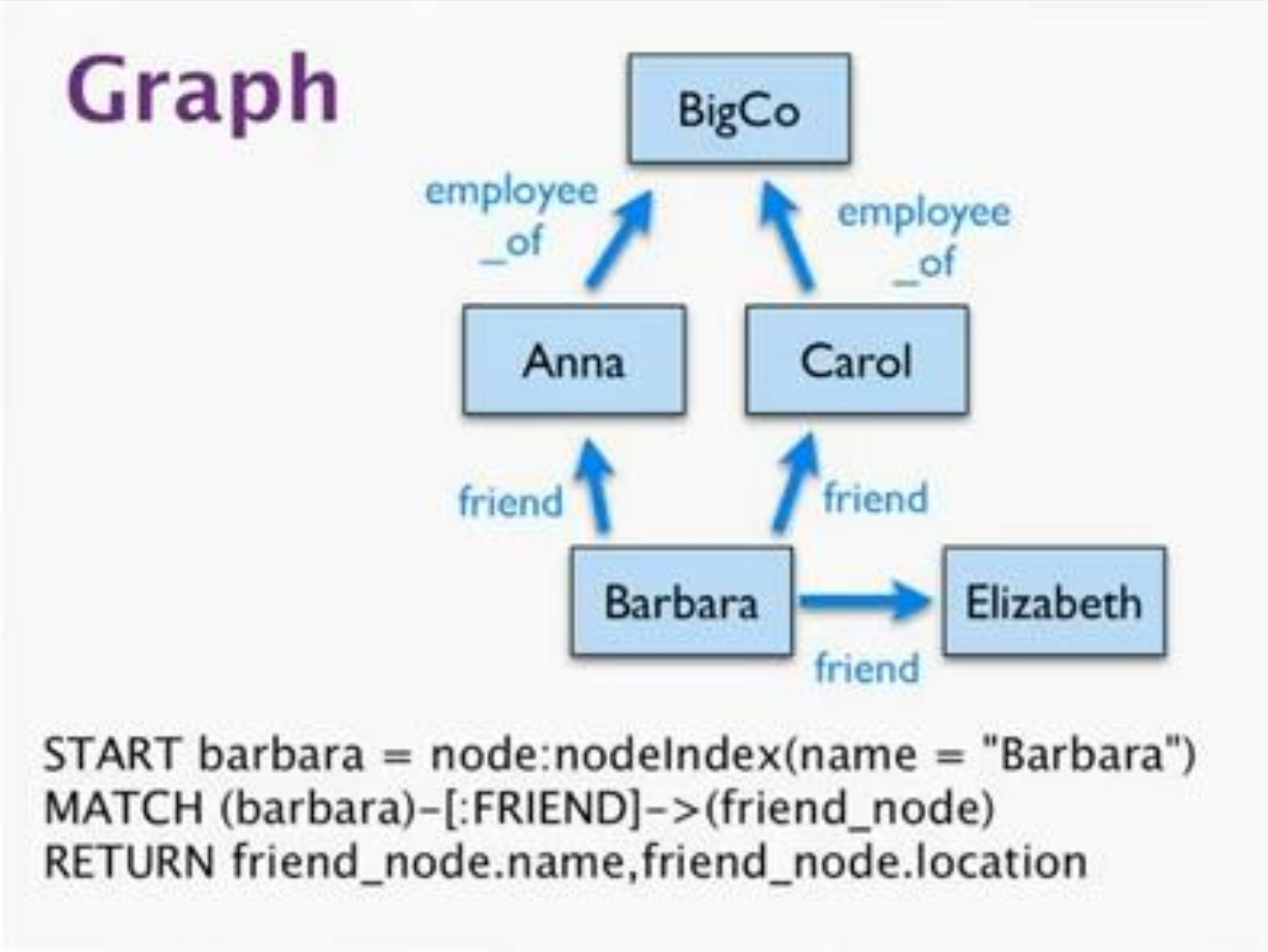
Aggregate – Oriented databases

- All aggregate oriented data models share the notion of an aggregate indexed by a key that you can use for lookup. This aggregate is central to running on a cluster, as the database will ensure that all the data for an aggregate is stored together on one node.
- The aggregate also acts as the atomic unit for updates, providing a useful, if limited, amount of transactional control.
- Key-Value treats the key as an opaque whole, you can only do key lookup for the whole aggregate, you cannot run a query nor retrieve a part of the aggregate.
- The document model makes the aggregate transparent to the database allowing you to do queries and partial retrievals. However, since the document has no schema, the database cannot act much on the structure of the document to optimise the storage and retrieval of parts of the aggregate.
- Column-family models divide the aggregate into column families, allowing the database to treat them as units of data within the row aggregate.

Graph Databases

- Graph databases allow you to store entities and relationships between these entities. Entities are known as nodes, which have properties.
- Think of a node as an instance of an object in the application. Relations are known as edges that can have properties.
- Edges have directional significance; nodes are organised by relationships which allow you to find interesting patterns between the nodes.
- The organisation of the graph lets the data to be stored once and then interpreted in different ways based on relationships.
- They work best for data that has complex relationship structures.

Graph Databases




```
START barbara = node:nodeIndex(name = "Barbara")
MATCH (barbara)-[:FRIEND]->(friend_node)
RETURN friend_node.name,friend_node.location
```

Graph Databases

- Once you have a graph of noes and edges created, you can query the graph in many ways, such as get all nodes employed by Big Co that Like NoSQL Distilled.
- A query on the graph is also known as traversing the graph.
- An advantage of a graph database is that we can change the traversing requirements without having to changes the nodes or edges. We can traverse the graph any way we like.
- Traversing the joins is very fast.

Distribution Models

- A primary driver of interest in NoSQL has been its ability to run databases on a large cluster. As data volumes increase, it becomes more difficult and expensive to scale up, a more appealing option is to scale out – run the database on a cluster of servers.
- Aggregate databases fit well with scaling out because the aggregate is a natural unit to use for distribution.
- Running over a cluster introduces more complexity.
- Replication and Sharding are two paths to data distribution.

Distribution Models

- Single Server: simplest of distribution models, use one server (no distribution at all), graph databases are an obvious type as they work best in a single server config.
- Sharding: Horizontal scalability means putting different parts of the data onto different servers. In an ideal case different users all talking to different server nodes. Each user only has to talk to one server. Load is balanced nicely. This is rare, so we have to ensure that data that's accessed together is placed on the same server node.
- Aggregated databases are designed to combine the data that's commonly accessed together.
- Many NoSQL databases offer auto-sharding, where the database takes on the responsibility of allocating data to shards and making sure data goes to the right shard.

Distribution Models

- Sharding is particularly valuable for performance because it can improve both read and write performance.
- Sharding means data is on different nodes, a node failure makes that shard's data unavailable but only for the users of that shard, other users won't suffer from the failure.
- Some databases are intended from the beginning to use sharding, in which case it is wise to run them on a cluster. Other databases use sharding a s deliberate scale up feature, in which case it is best to start single-server only use sharding once your load projections indicate that you are running out of headroom.

Distribution Models

- Master-Slave replication: you replicate across nodes. One node is designed as the master, or primary. This is the authoritative source for the data and is usually responsible for processing any updates to that data. The other nodes are slaves or secondary's. A replication process synchronises the slaves with the master.
- Replication is a good solution if you have read intensive data. You can scale horizontally to handle more read requests by adding more slave nodes and ensuring that all read requests are routed to the slaves.
- You are still limited by eh ability of the master to process updates and tis ability to pass those updates on. Consequently it isn’t such a good scheme for datasets with heavy write traffic.

Distribution Models

- Replication is good for read resilience. Should the master fail, the slaves can still handle read requests. Failure of the master does eliminate the ability to handle writes until the master is restored or a new master is appointed. However, having slaves as replicates of the master does speed up recovery since a slave can be appointed a new master very quickly.
- Replication can mean inconsistency. You have the danger that different clients reading different slaves, will see different values because the changes haven't all propagated to the salves.

Distribution Models

- Peer-to-peer replication: in replication the master is still the bottleneck and a single point of failure. Peer-to-peer addresses these problems by not having a master. All replicated nodes are equal, they can all accept writes. Biggest complication is consistency. When you can write to two different places you run the risk that two people will attempt to write to the same place at the same time.
- Combining sharding and replication: there can be multiple masters but each data item has one master.

consistency

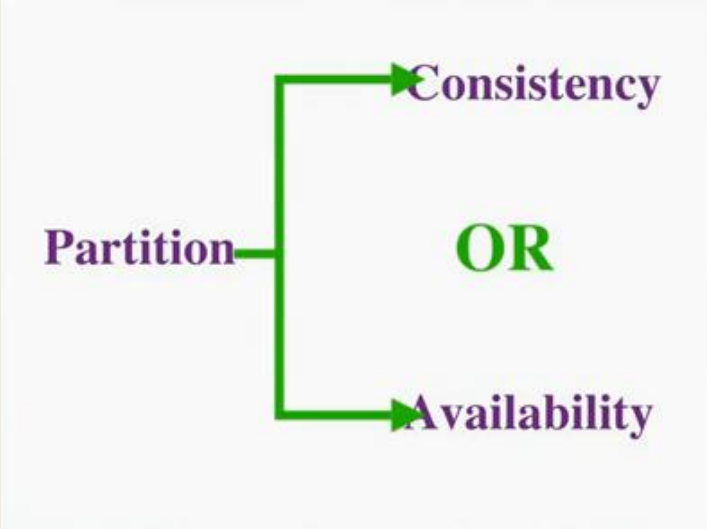
- One of the biggest changes from a centralised relational database to a cluster oriented NoSQL database is in how you think about consistency. Relational database try to exhibit strong consistency. When you start looking at the NoSQL world, phrases such as “CAP theorem” and “eventual consistency” appear.
- There are various approaches for maintaining consistency. A pessimistic approach prevents conflicts from occurring, an optimistic approach lets conflicts occur but detects them and takes action to sort them out.

CAP theorem

- CAP theorem: Originally posted by Eric Brewer in 2000.
- The basic statement of the theorem is that, given the three properties of Consistency, Availability, and Partition tolerance, you can only get two.
- Consistency is the consistent uptodate version of the data, Availability means that if you can talk to a node in the cluster, it can read and write data. Partition tolerance means that the cluster can survive communication breakages that separate the cluster into multiple partitions unable to communicate.
- Single server is an example of a consistent available system, it is not partition tolerant.
- A cluster that may suffer partitions as distributed systems do, you have to trade off consistency versus availability.

CAP theorem

- For example Martin and Pramod are both trying to book the last hotel room on a system that uses peer-to-peer distribution with two noes (Lond for Martin and Mumbai for Pramod). If we want to ensure consistency, then when Martin tries to book his room on the London nodes, that node communicates with Mumbai node before confirming the booking. Essentially both nodes must agree on the serialisation of their requests (who goes first). This gives consistency, but should the network link break then both nodes must not allow any booking, thus sacrificing availability.



CAP Theorem

- One way to improve availability is to designate one node as the master for a particular hotel and ensure all bookings are processed by that master. That master can then continue to be available even if the connection goes down. The slave node would not be allowed to book while the connection is down thus ensuring consistency. It improves the availability but the slave node users still couldn't book a room for that hotel. We cold allow slaves to book rooms even with the connection down thus allowing inconsistent writes to occur and thus a double booking. In this context this is acceptable but in others it may not be.

Polyglot

- Different databases are designed to solve different problems.
- Many enterprises use the same database engine to store business transactions, session management data, reporting, BI, data warehousing or logging information.
- Neal Ford 2006 coined the term Polyglot programming to express the idea that applications should be written in a mix of languages to take advantage of the fact that different languages are suitable for tackling different problems.
- Similarly when working on an e-commerce business problem, using a data store for the shopping cart which is highly available and can scale is important, bu thte same data store cannot help you find products bought by the customers' friends – which is a totally different question. We use the term polyglot persistence to define this hybrid approach to persistence.

Beyond NoSQL

- The appearance of NoSQL databases has done a great deal to shake up and open up the world of databases. There are other systems that don’t fit the NoSQL bucket.
- File systems: similar to key value stores, google’s file system and Hadoop.
- Event sourcing: persisting all the changes to a persistent state (location of ships)
- XML: similar to document databases, data model
- Object databases: manage in memory data structures onto disk. Didn’t take off.

Reasons to use NoSQL

The two main reasons to use NoSQL technology are:

1. To improve programmer productivity by using a database that better matches an application's needs.
   - *e.g. removing impedance mismatch by storing objects together in aggregates rather than splitting them up into relational tables*

2. To improve data access performance via some combination of handling larger data volumes, reducing latency, and improving throughput.
   - *When a database is large enough to be split over several database servers, NoSQL may be a good option*

Reasons to stick with Relational DBs

- They are well-known, therefore it is easier to find people with experience of using them
- The technology is more mature and less likely to encounter problems
- Many other tools are built on relational technology

"A DBA walks into a NOSQL bar, but turns and leaves because he couldn't find a **table**"