QA Consulting

Introduction

- Databases!
- Database design
- Entity relationship diagrams
- Relational databases
- NoSQL

Databases

- There are many kinds of database
- Each has different pros and cons, and will be used for different situations
- We will explore several over this course, but be aware there are many more!
- Always check you are using the best tool for the job

Codd’s (12…) rules

0. The Foundation rule – any RDBMS must be able to manage databases entirely through its relational capabilities.
1. The Information rule – all information in an RDBMS must be a value of some table cell.
2. The Guaranteed Access rule – every single value has to be accessible logically with a combination of table name, primary key (row value), and attribute name (column). No other means, such as pointers, can be used to access data.
3. Systematic Treatment of Null Values – NULL values must be given systematic and uniform treatment, independent of data type. Consider that NULL values could be due to missing data, unknown data, or inapplicable data.
4. Dynamic Online Catalog Based on the Relational Model – the database structure must be stored in an online catalog known as the data dictionary. This should be accessible by authorised users who can use the same query language from directly accessing the database.
5. The Comprehensive Data Sublanguage rule – the database can only be accessed using a language either directly or via an application. This language should support data definition, view definition, data manipulation, integrity constraints, authorization, and transaction boundaries.

Codd’s (12…) rules

6. The View Updating rule – all views of the database that could theoretically be updated must be updateable by the system
7. High-Level Insert, Update, and Delete – the database should support the ability perform operations such as insert, update, and delete to more than a single row, such as union operations.
8. Physical Data Independence – if changes are made in storage or access methods, the application programs and console activity should remain logically the same
9. Logical Data Independence – user applications must not be affected by logical data changes e.g. two tables are merged, the user should see no impact
10. Integrity Independence – integrity constraints must be defined in the relational data sublanguage and storable in the catalog, not in the application programs
11. Distribution Independence – the user must not be able to tell that data is distributed over various locations, it should appear as though it is stored at one site only
12. Non-Subversion rule – single record access must not be used to subvert or bypass integrity rules and constraints expressed in the higher-level language

Database design

- Before we start creating a database, we should model it first and test it against any requirements we may already have
- For this we will be using an entity relationship diagram

- Consider:
  - What data it will be storing
  - Who will be using it
  - What the data will be used for

- Do you have any documentation to work with already?

Database design

- Entity
    - Could be an object, person, event, or an abstraction
    - Must be relevant
    - Can be referred to as an ‘instance’
- Entity type
    - Class of objects
    - Same characteristics
- E.g. ‘Person’ entity type has the entities ‘James’ and ‘Jo’

Database design

- Attributes
- Facts about the entity occurrence
- Instance level – fact about entity occurrence
- Abstract level – class of facts about instances of an entity type
- Use key attributes to identify instances
    - If there is not a single attribute then you must be able to determine unique fields by a combination of other values – composite keys

Database design

- Relationships
- Associations between entity types
- Describes how they are related – should exist both ways
- Cardinality – restricts how often an entity can occur

Database design

- Think in terms of the end user – your database needs to contain all the information that your end user needs, and it should be logical
- Ensure all information is valid – check by using constraints
- Be wary of including too much unnecessary information!

Objects First Vs. Tables First

Objects First

- With Objects first you start by creating the Java objects and then creating the database tables later.
- Using a Persistence Provider library allows us to reverse engineer the database from our code.
- The advantage with this method is that it is very easy to change what data we are persisting.

Tables First

- With Tables first you start by architecting the database and then create the Java objects later.
- It is cheaper is a database already exists especially if that database is servicing multiple systems.
- The advantage is that it forces us to plan early on and avoids any late changes to the data we are persisting

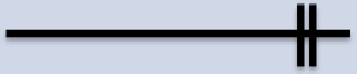
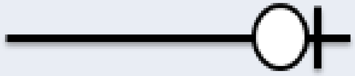
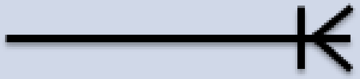
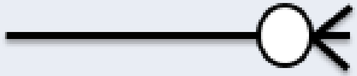
Entity Relationship Diagrams

Regardless of our approach we will need to create an ERD for our design.

- ERD are used to show the design of a database schema with names, columns and relationships.
- The ERD is about the relationships between DATA not Objects.
- With a complete set of user stories creating the ERD should be simple.
- MySQL Workbench contains a good tool for creating ERD diagrams.

Cardinality

The relationships between data

Symbol	Meaning
	One and Only One
	Zero or One
	One to Many
	Zero to Many

ERD Creation Steps

It’s as easy as 1 2 3!

1. Identify the Entity
2. Identify the Attributes
3. Identify the Primary Keys
4. Identify the relationships
5. Identify the cardinality
6. Draw a Draft
7. Map the Attributes
8. Refine the ERD

Take your user stories.

Pick one.

Follow these steps using the information in the user story.

When you’ve finished step 8, Pick another one.

Repeat till there are no more user stories.

Most user stories will need to be looked at more than once.

Don’t add anything that isn’t in a user story.

Step 1: Identify The Entity

“As a customer I want to find books by a specific author to find other works by that author I may be interested in.”

Here we can identify two entities: Book and Author.

Because this is an example we will look at both of them.

You should look at each entity separately.

Step 2: Identify The Attributes

“As a customer I want to find books by a specific author to find other works by that author I may be interested in.”

Here we can identify that for book we have an attribute of author.

Because we are looking at both entities we can also see that author has an implied attribute of name.

We don’t have to identify an attribute in every user story, this will be one of several stories so we aren’t expecting to find everything.

Step 3: Identify The Primary Keys

"As a customer I want to find books by a specific author to find other works by that author I may be interested in."

Here we can identify that for book we don't actually have a primary key that we can identify in this story.

We could make the assumption here that author's identified attribute of name could be a primary key.

We don't have to identify a primary key in every user story, this will be one of several stories so we aren't expecting to find everything.

Step 4: Identify The Relationships

“As a customer I want to find books by a specific author to find other works by that author I may be interested in.”

Here we can identify that there is a relationship between book and author.

Again, we aren’t expecting to find everything in one story.

We can create a Relationship Matrix to help us see the relationships.

	Book	Author
Book		Writes
Author	Is Written by	

Step 5: Identify The Cardinality

“As a customer I want to find books by a specific author to find other works by that author I may be interested in.”

A Book is written by **One and Only One** Author.
An Author writes **One Or More** Books.

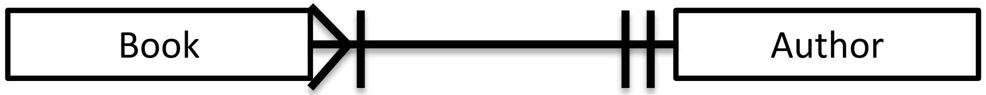
Here we can identify that an author can write many books.

Because we can say that an author isn’t an author until they have written a book we can say that the relationship is One to Many.

We can write these out so that they are easy to see.

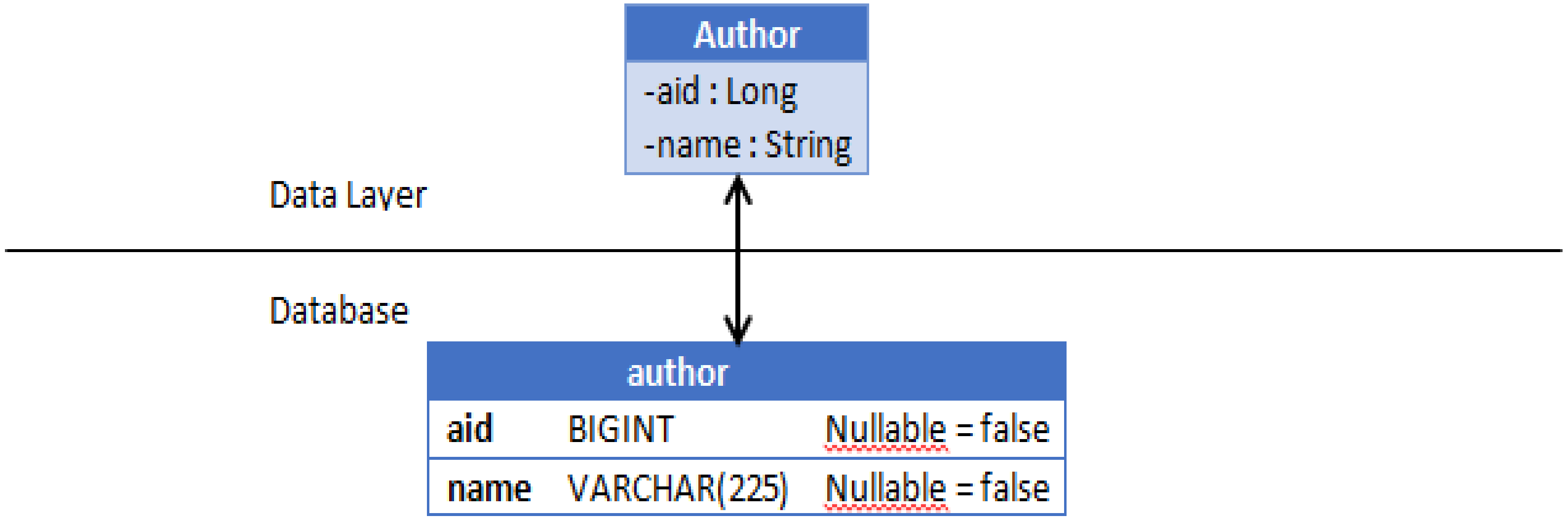
Step 6: Drawing the Draft

All we want here is the name with the relationship and cardinality



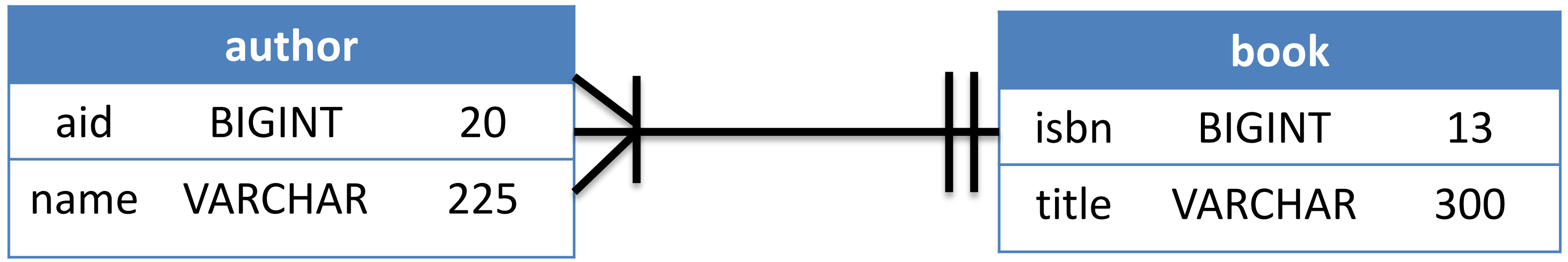
Step 7: Mapping Attributes

Because data types aren't always the same


Author
-aid : Long
-name : String
Data Layer
Database
author
aid BIGINT Nullable = false
name VARCHAR(225) Nullable = false

Step 8: Refining the ERD

You can use Workbench for this step



Keys

- Primary key – uniquely identifies a row within a table, can be a combined key consisting of more than one column
- Foreign key – include a copy of another table's primary key in a table to refer to it, reduces duplicated data
- These are also used in joins which we will go into later

Normalization

- Should eliminate non-atomic values that could negatively affect performance or require complex code
- Should eliminate redundant data
- Removes modification anomalies

- The aim is to organise the columns and tables of a relational database to minimise data redundancy

Normalization forms

- Many forms of normalization
- Applies to table
- The normal form of a database is the lowest normal form of all its tables
- Tables should generally be normalized to the third normal form – there are differing opinions on the importance of higher normal forms
- Can normalize before tables are created, when we are still creating our ERD

First Normal Form

- Table must have a key that all attributes depend on
- Every column stores atomic (not composite) values – consider carefully
- Imagine we need to record multiple telephone numbers for some customers

Customer ID	First Name	Surname	Telephone number
123	Robert	Ingram	01276 712 8282
456	Jane	Wright	03719 711 8422
789	Maria	Fernandez	09164 124 7291

First Normal Form

- Imagine we need to record multiple telephone numbers for some customers
- We could just allow Telephone Number to contain more than one value
- This wouldn’t conform with 1NF though!

Customer ID	First Name	Surname	Telephone number
123	Robert	Ingram	01276 712 8282
456	Jane	Wright	03719 711 8422 0781551627
789	Maria	Fernandez	09164 124 7291

First Normal Form

- Alternatively we could do the following…
- However two customers could have the same name, and therefore does not meet the requirements of some higher forms

Customer ID	First Name	Surname	Telephone number
123	Robert	Ingram	01276 712 8282
456	Jane	Wright	03719 711 8422
456	Jane	Wright	0781551627
789	Maria	Fernandez	09164 124 7291

First Normal Form

- This design would meet 1NF and some higher forms!

Customer Telephone Number

Customer ID	<u>Telephone number</u>
123	01276 712 8282
456	03719 711 8422
456	0781551627
789	09164 124 7291

Customer Name

<u>Customer ID</u>	First Name	Surname
123	Robert	Ingram
456	Jane	Wright
789	Maria	Fernandez

Second Normal Form

- Must meet all requirements of First Normal Form
- Non-prime attributes must not depend on a subset of any candidate key – no partial dependency

Second Normal Form

- Even if the primary key is specified as {Model Full Name}, {Manufacturer, Model} is also a candidate key, and Manufacturer Country is dependent on a subset of it: Manufacturer

Manufacturer	Model	Model Full Name	Manufacturer Country
Forte	X-Prime	Forte X-Prime	Italy
Forte	Ultraclean	Forte Ultraclean	Italy
Dent-o-Fresh	EZbrush	Dent-o-Fresh EZbrush	USA
Kobayashi	ST-60	Kobayashi ST-60	Japan
Hoch	Toothmaster	Hoch Toothmaster	Germany
Hoch	X-Prime	Hoch X-Prime	Germany

Second Normal Form

- The following conforms to 2NF

Toothbrush Manufacturers

Manufacturer	Manufacturer Country
Forte	Italy
Dent-o-Fresh	USA
Kobayashi	Japan
Hoch	Germany

Toothbrush Models

Manufacturer	Model	Model Full Name
Forte	X-Prime	Forte X-Prime
Forte	Ultraclean	Forte Ultraclean
Dent-o-Fresh	EZbrush	Dent-o-Fresh EZbrush
Kobayashi	ST-60	Kobayashi ST-60
Hoch	Toothmaster	Hoch Toothmaster
Hoch	X-Prime	Hoch X-Prime

Third Normal Form

- Must meet requirements from First and Second Normal Forms
- All non-prime attributes must not be transitively dependent on every key of R

- Transitive dependency
- **X determines Z** indirectly because **X determines Y** and **Y determines Z** (where Y does <u>not</u> determine X)
- X -> Z because X -> Y and Y -> Z

Example

- Each row needs to tell us who won a particular Tournament in a particular Year so we have a composite key {Tournament, Year} to uniquely identify a row, this is our candidate key

Tournament	Year	Winner	Winner D.O.B.
Indiana Invitational	1998	Al Fredrickson	21 July 1975
Cleveland Open	1999	Bob Albertson	28 September 1968
Des Moines Masters	1999	Al Fredrickson	21 July 1975
Indiana Invitational	1999	Chip Masterson	14 March 1977

Example

- Winner D.O.B. is transitively dependent on the candidate key via the non-prime attribute Winner
- Logical inconsistencies could appear with Winners appearing with different D.O.B.s

Tournament	Year	Winner	Winner D.O.B.
Indiana Invitational	1998	Al Fredrickson	21 July 1975
Cleveland Open	1999	Bob Albertson	28 September 1968
Des Moines Masters	1999	Al Fredrickson	21 July 1975
Indiana Invitational	1999	Chip Masterson	14 March 1977

Example

- The solution?

Tournament	Year	Winner
Indiana Invitational	1998	Al Fredrickson
Cleveland Open	1999	Bob Albertson
Des Moines Masters	1999	Al Fredrickson
Indiana Invitational	1999	Chip Masterson

Winner	Winner D.O.B.
Chip Masterson	14 March 1977
Al Fredrickson	21 July 1975
Bob Albertson	28 September 1968

Codd

- First Normal Form – The key,
- Second Normal Form – the whole key,
- Third Normal Form – and nothing but the key

Higher Normal Forms

- Boyce-Codd Normal Form (BCNF)
- Fourth Normal Form
- Fifth Normal Form
- Domain-Key Normal Form
- Sixth Normal Form

- Some people claim these forms are less useful in real-world situations and only an academic interest, however it is worth being aware of them, we just won’t go into detail

Denormalization

- Note that this is not ‘un-normalized’, denormalization happens *after* normalization
- Can improve read performance of a database

- If you have created many separate tables when normalizing a database, it could be that these relations are stored as separate disk files
    - Therefore, a query that takes information from many relations (such as a *join*) the operation may be very slow

NoSQL

- We’ve covered conventional relational databases
- NoSQL databases do not need the same structure and relationships that relational databases do
- Eventual consistency
- Write buffering
- Only primary keys can be indexed
- Queries must be written as programs – typically no query language

NoSQL

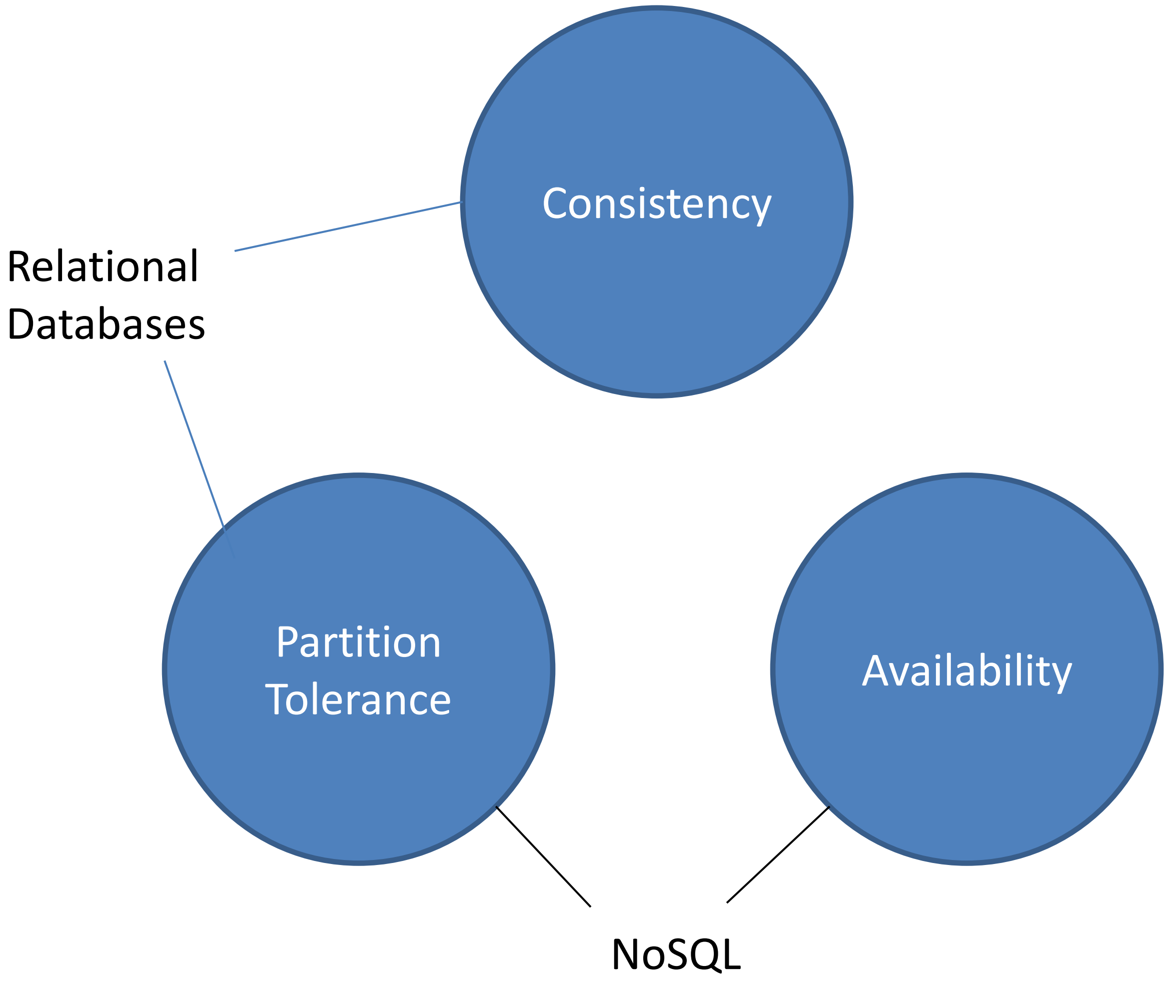
- Distributed databases
- Partitioning pattern called Sharding
- Split over numerous disks and each partition is managed on a different server – can be geographically distributed
- Fan-out queries where you submit a query to a master node then it is distributed to other nodes in the cluster
- May duplicate partitions for replication, useful for disaster recovery

NoSQL

ID: 1122	sname: Brust	fname: Andrew	address: 123 Main St.	city: New York	state: NY	zip: 10099
ID: 3214	sname: Doe	fname: John	address: 321 Else Rd.	village: Stodday	county: Lancashire	postcode: LA2 6ET

ID: 3001	customerID: 1122	amount: 500	tax: 40	processdate: 2/20/2015
ID: 3002	customerID: 1122	amount: 250	shipdate: 3/18/2015	
ID: 3003	amount: 1700	tax: 150		

CAP


Consistency
Relational Databases
Partition Tolerance
Availability
NoSQL

NoSQL

- Consider what types of data we should store in NoSQL and relational databases
- Things like inventory may *need* to be consistent – imagine if stock was depleted and only the server in London knew, that the server in Paris didn't.
    - A customer in Paris could order that item and the order would be processed even though there's no stock
- Comparatively, catalogue information may not need to be updated immediately, some customers could see the new item before others which shouldn't have a huge impact, just update it quickly

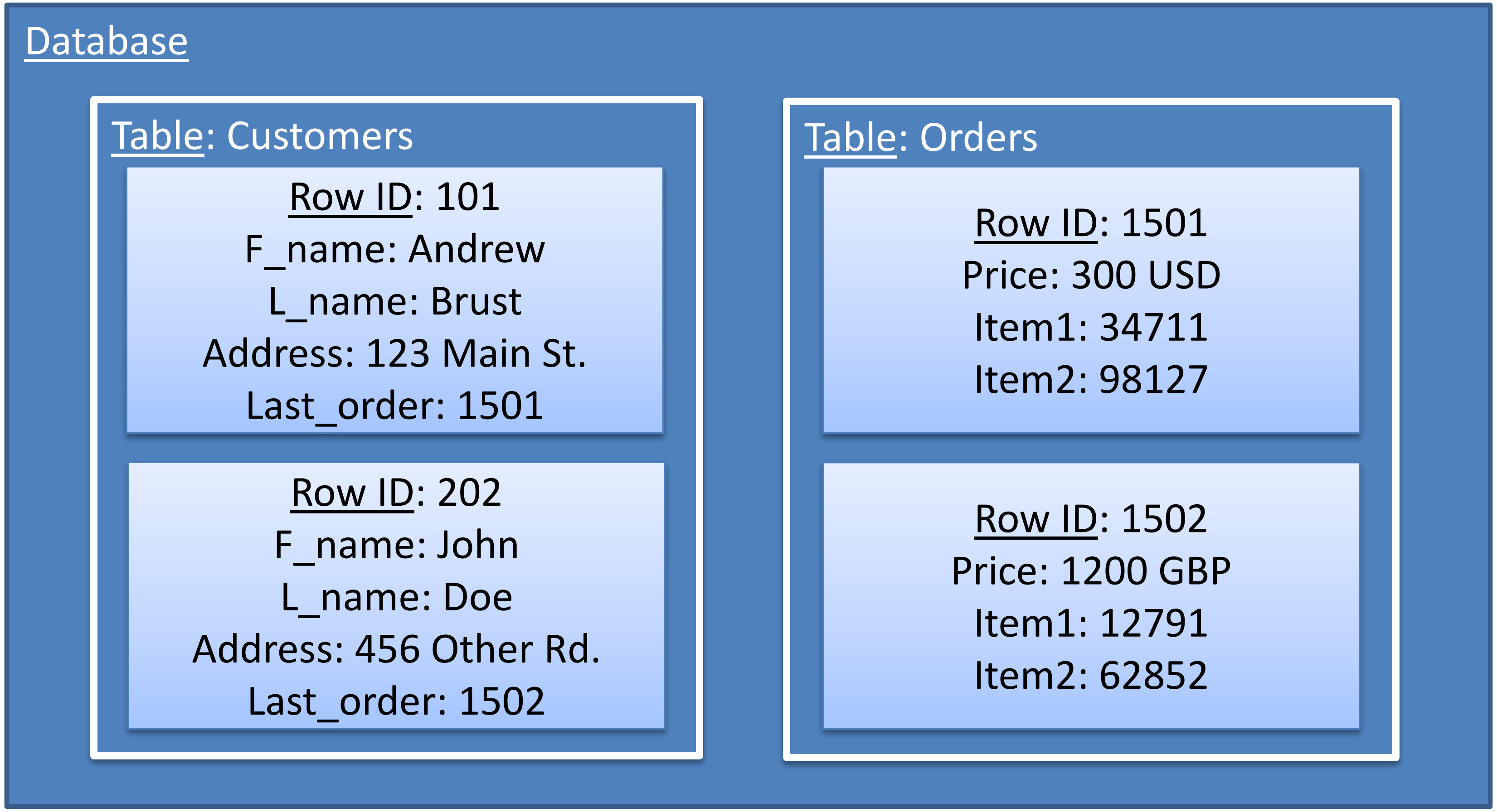
NoSQL Categories

- Key-value stores
  - There is a key that has an associated value. We’ve seen this earlier, for example Name: “Alan”
- Document stores
  - Everything in a document, documents instead of rows, JSON
- Wide Column stores
  - Know the group of columns but not the columns themselves
- Graph databases
  - Relationship focussed

Key-Value stores

- Data exists as key-value pairs
- Rows with some sort of associative array, but the schema may change from row to row
- No relationships defined between tables
- Use programmatic conventions, recognise that there are links between tables and create your own foreign keys when working with the data

Key-Value stores



Wide Column stores

- There are tables but they don’t belong to databases. In fact, there are no databases, only tables
- The tables have rows, the rows have ‘super-columns’, and the ‘super-columns’ have columns within them
- Super-columns defined when the table is defined

Wide Column stores

Table: Customers

- Row ID: 101
- Super Column: Name
  - Column: F_name: Andrew
  - Column: L_name: Brust
- Super Column: Address
  - Column: Number: 123
  - Column: Street: Main St.
- Super Column: Orders
  - Column: Last_order: 1501

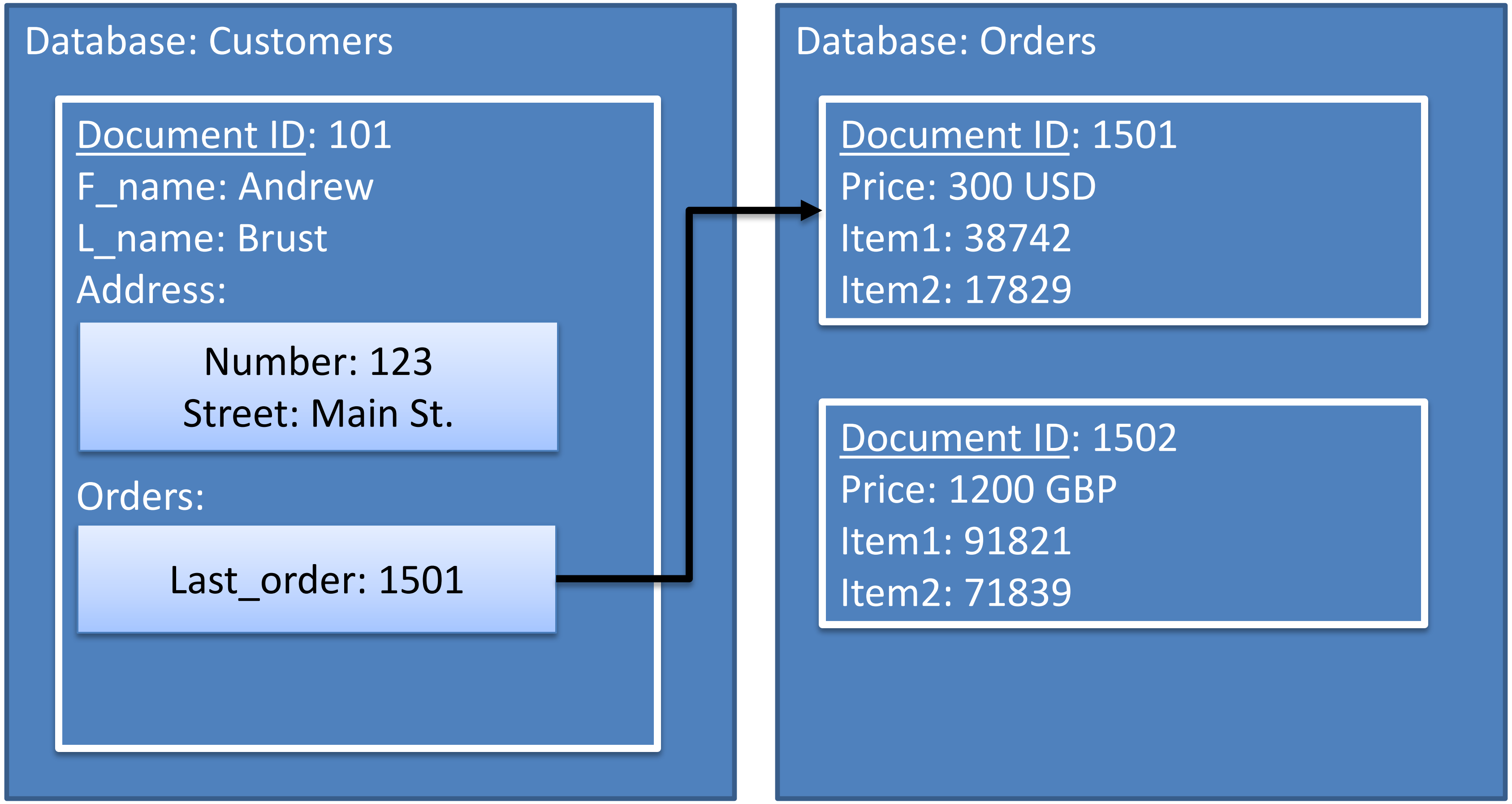
Table: Orders

- Row ID: 1501
- Super Column: Pricing
  - Column: Price: 300 USD
- Super Column: Items
  - Column: Item1: 83474
  - Column: Item2: 91831

Document stores

- Databases are back, this time they conceptually function just like tables inside of key-value or wide column stores
- Databases contain documents which function like rows
- The documents contain key-value pairs
- However the value of a key could actually be a document itself
- Can have relationships between documents even if they're in different databases
- Typically JSON objects

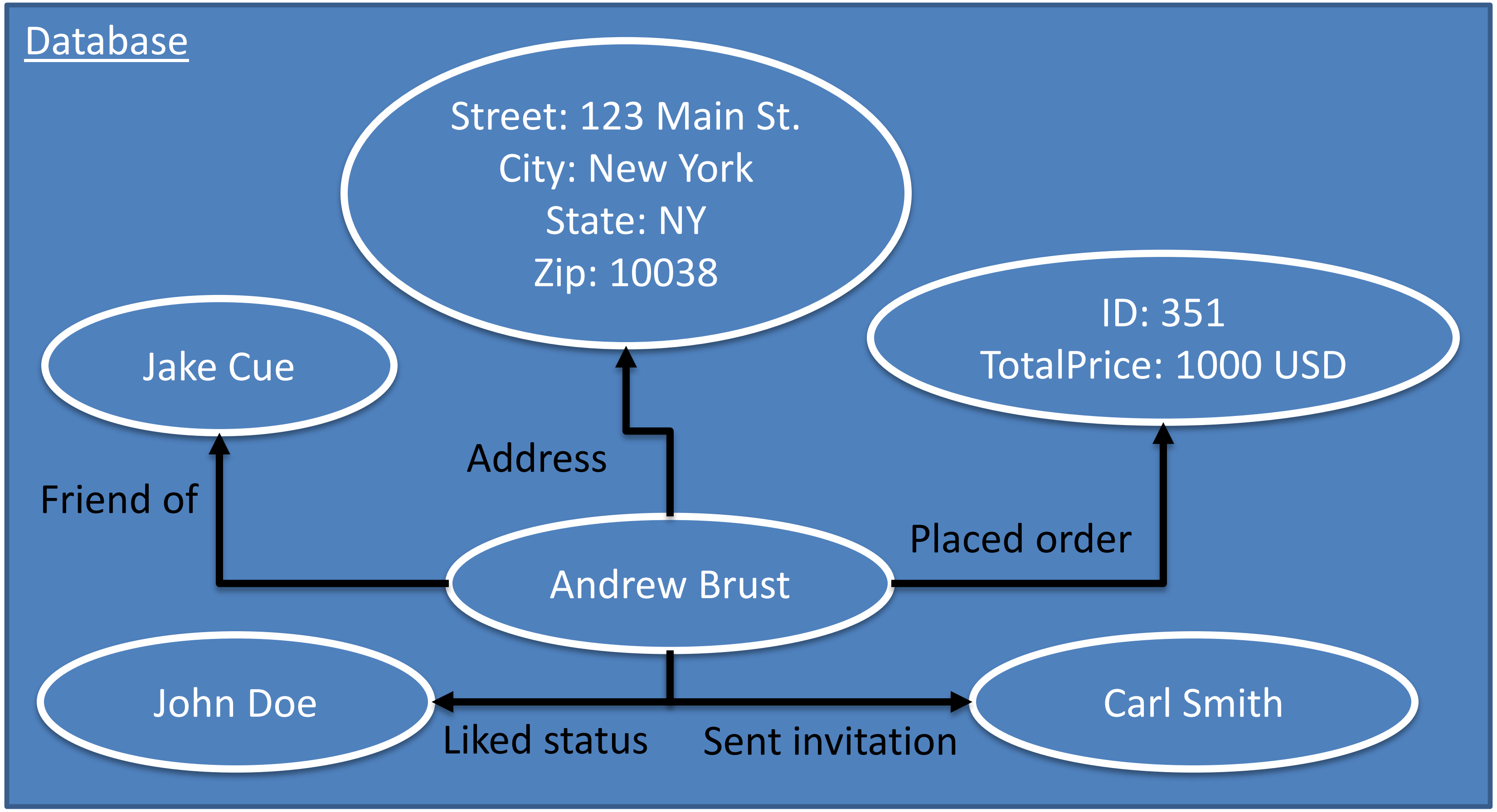
Key-Value stores



Graph databases

- Starts with a database but then gets very different
- We have nodes with names and properties, and relationships with other nodes (edges)
- Very useful for social media

Graph databases



NoSQL performance

- Best for if you have high traffic and concurrent users who only need to fetch a few bits of information
- Consider using NoSQL for things like configuration or settings for a web service, or log files and event data
- Not designed for querying and data warehousing, however wide column stores are good for Big Data
- Can also be useful with Extract, Transform, Load (ETL) operations
- Would not be directly used for Business Intelligence but can be used in the BI pipeline

NoSQL and Big Data

- No indexing isn’t a problem
- Consistency shouldn’t be an issue
- Fast reads are necessary
- Distributed file systems
- Commodity hardware
- NoSQL generally concerned over large web-scale, but Big Data is on a massive scale as well

NoSQL Databases

- MongoDB – document store
    - Follow along with the instructions and examples here https://docs.mongodb.org/getting-started/shell/
- Cassandra – wide column store
    - For now we will experiment online, later in the course we will go over installing and working with Cassandra in Big Data - http://www.planetcassandra.org/try-cassandra/
- Hbase – wide column store
    - We will use this more later in the course
- Oracle – key-value store

Querying NoSQL

- NoSQL is actually a very misleading name, it more means non-relational, many still have support for SQL queries
- Cassandra has its own – CQL
- We will access HBase with MapReduce later on
- There is also Hive that works with Hadoop that we will use
- These SQL-like languages do have many similarities

SQL

- Data Definition Language – DDL
    - CREATE, ALTER, and DROP statements
- Data Manipulation Language – DML
    - SELECT, INSERT, UPDATE, and DELETE statements
- Data Control Language – DCL
    - Security – GRANT and REVOKE statements
- Programming Interface

SQL Basics

- You now have a database filled with data that you want to query!
- A good starting point is to simply explore your data
- SELECT * FROM Customers – get all the data from the Customers table and print it to the screen
- SELECT f_name FROM Customers WHERE age > 50 – get all the first names of customers over 50

SQL Basics

- DESCRIBE Customers – shows the schema for the Customers table
- Use the DISTINCT keyword to return only unique values
- Use the ORDER BY clause to display the results ordered on a specific value either ASC or DESC

SQL Basics

- SELECT department, profit_target * 1.2 AS ‘next_target’ FROM departments – the AS word creates a ‘new’ column for the output using + - * /

- SELECT CONCAT(fname,‘ ’,lname) AS ‘fullname’ FROM employees – function is dependent on what you’re working with, but all SQL languages have some sort of concatenation. There are many other functions available e.g. SUBSTRING

SQL Basics

- SELECT fname, lname FROM employees WHERE fname IN(‘Tom’, ‘James’, ‘David’)
- SELECT fname, lname FROM employees WHERE fname NOT IN(‘Tom’, ‘James, ‘David’)
- SELECT address FROM Customers WHERE f_name LIKE ‘Jo%’ – return all address of Customers whose name begins with ‘Jo’

SQL Basics

- SELECT * FROM customers WHERE complaints IS NULL

- SELECT fname COELESCE(home_phone, mobile_phone, work_phone, ‘Not on phone’) FROM customers – coalesce returns the first non-NULL value and includes a ‘last resort’

Nested Queries

- SELECT * FROM salesperson WHERE dept_no IN
    (SELECT dept_no FROM dept
    WHERE dept_name LIKE ‘Lo%’)

- The subquery runs first, then the outer query runs on the result from the subquery

Correlated Sub-queries

-
  ```
  SELECT company_no, order_value, emp_no
  FROM sale S1
  WHERE order_value =
          (SELECT MAX(order_value)
          FROM sale S2
          WHERE S2.company_no = S1.company_no)
  ```

- Outer query runs first and the sub-query is executed once for each row in the main query

SQL Basics

- CREATE TABLE tablename (colname1 int, colname2 varchar(10)…) – creates a table called tablename with the following columns and data types
- INSERT INTO tablename (colname1, colname2…) values (20, “hello”…) – inserts the values 20 and hello into colname 1 and colname2
- Can use SELECT in an INSERT INTO statement

SQL Basics

- Referential integrity – relating tables via their content to maintain business rules
- Don’t allow people to violate these rules via manipulations
- What if we had the Sales department in a departments table and then deleted that. Would any employee in the employees table who had the department Sales also be deleted?

SQL Basics

- CREATE TABLE contact (

```
company_no      INTEGER         NOT NULL,
contact_code    CHAR(3)         NOT NULL,
name            VARCHAR(20)NULL,
job_title       VARCHAR(20)NULL,

PRIMARY KEY(company_no, contact_code),
FOREIGN KEY(company_no)
        REFERENCES company
                ON DELETE CASCADE
                ON UPDATE RESTRICT )
```

SQL Basics

- UPDATE tablename SET colname = 3742839 WHERE colname2 = ‘this’
- Update sales SET sales_target = sales_target * 1.2, notes = ‘has had 20 percent increase’ WHERE dept = 3
- Be wary of transactional statements – the COMMIT and ROLLBACK statements can help with this

SQL Basics

- DELETE FROM tablename WHERE colname1 = 20 – delete all rows where colname1 is 20, be sure to add a WHERE clause else all records will be deleted
- DROP TABLE tablename – deletes the table

SQL Joins

- SELECT SP.dept_no, lname, manager
  FROM salesperson SP JOIN dept D
  ON SP.dept_no = D.dept_no

- Joins are necessary from performing normalisation
- Effectively brings tables together
- There are different kinds of joins, the one above is an ‘inner’ join, and some databases may require the INNER keyword

SQL Joins

- Inner joins will only give results that match from the ‘main’ table
- Left outer join will return all rows from the table on the ‘left’, regardless of if there’s a match in the ‘right’
- Right outer join will return all rows from the table on the ‘right’, regardless of if there’s a match in the ‘left’
- Full outer join will return all rows from both tables

Aggregate Functions

- Use a number of rows and produce a single result
    - COUNT
    - SUM
    - MIN
    - MAX
    - AVG
- Without a WHERE clause – will work on all rows in a table
- With a WHERE clause – will work on all rows that match the condition

Grouping

- GROUP BY often used in conjunction with aggregate functions

```
SELECT region, state,
COUNT(id) AS num
FROM stores
GROUP BY region, state;
```

stores table

id	city	state	region
A	Albany	NY	EAST
B	Boston	MA	EAST
C	Chicago	IL	NORTH
D	Detroit	MI	NORTH
E	Elgin	IL	NORTH

Query Result

region	state	num
EAST	MA	1
EAST	NY	1
NORTH	IL	2
NORTH	MI	1

Inner Join

- SELECT c.cust_id, name, total FROM customers c JOIN orders o ON (c.cust.id = o.cust_id)

customers table

Cust_id	Name	Country
A	Aidan	US
B	Brian	CA
C	Cathy	MX
D	Daisy	DE

orders table

Order_id	Cust_id	Total
1	A	1539
2	C	1871
3	A	6352
4	B	1456
5	Z	2137

Query Result

Cust_id	Name	Total
A	Aidan	1539
A	Aidan	6352
B	Brian	1456
C	Cathy	1871

- Example: Left Outer Join

```
SELECT c.cust_id, name, total
FROM customers c
LEFT OUTER JOIN orders o
ON (c.cust_id = o.cust_id);
```

customers table

Cust_id	Name	Country
A	Aidan	US
B	Brian	CA
C	Cathy	MX
D	Daisy	DE

orders table

Order_id	Cust_id	Total
1	A	1539
2	C	1871
3	A	6352
4	B	1456
5	Z	2137

Query Result

Cust_id	Name	Total
A	Aidan	1539
A	Aidan	6352
B	Brian	1456
C	Cathy	1871
D	Daisy	NULL

- Example: Right Outer Join

```
SELECT c.cust_id, name, total
FROM customers c
RIGHT OUTER JOIN orders o
ON (c.cust_id = o.cust_id);
```

customers table

Cust_id	Name	Country
A	Aidan	US
B	Brian	CA
C	Cathy	MX
D	Daisy	DE

orders table

Order_id	Cust_id	Total
1	A	1539
2	C	1871
3	A	6352
4	B	1456
5	Z	2137

Query Result

Cust_id	Name	Total
A	Aidan	1539
A	Aidan	6352
B	Brian	1456
C	Cathy	1871
NULL	NULL	2137

- Example: Full Outer Join

```
SELECT c.cust_id, name, total
FROM customers c
FULL OUTER JOIN orders o
ON (c.cust_id = o.cust_id);
```

Query Result

Cust_id	Name	Total
A	Aidan	1539
A	Aidan	6352
B	Brian	1456
C	Cathy	1871
D	Daisy	NULL
NULL	NULL	2137

customers table

Cust_id	Name	Country
A	Aidan	US
B	Brian	CA
C	Cathy	MX
D	Daisy	DE

orders table

Order_id	Cust_id	Total
1	A	1539
2	C	1871
3	A	6352
4	B	1456
5	Z	2137

Finding Unmatched Entries

```
SELECT c.cust_id, name, total
FROM customers c
FULL OUTER JOIN orders o
ON (c.cust_id = o.cust_id)
WHERE c.cust_id IS NULL
OR o.total IS NULL;
```

customers table

Cust_id	Name	Country
A	Aidan	US
B	Brian	CA
C	Cathy	MX
D	Daisy	DE

orders table

Order_id	Cust_id	Total
1	A	1539
2	C	1871
3	A	6352
4	B	1456
5	Z	2137

Query Result

Cust_id	Name	Total
D	Daisy	NULL
NULL	NULL	2137

Cross Join

```
SELECT * FROM disks
CROSS JOIN sizes;
```

disks table

Name
Internal hard disk
External hard disk

sizes table

Size
1.0 terabytes
2.0 terabytes
3.0 terabytes
4.0 terabytes

Query Result

Name	Size
Internal hard disk	1.0 terabytes
Internal hard disk	2.0 terabytes
Internal hard disk	3.0 terabytes
Internal hard disk	4.0 terabytes
External hard disk	1.0 terabytes
External hard disk	2.0 terabytes
External hard disk	3.0 terabytes
External hard disk	4.0 terabytes

SQL UNION

- Use UNION [ALL] to join output from SELECT commands into a result set – use ALL to keep duplicates
- They must have the same column structure

```
SELECT emp_id, fname, lname, salary FROM employees WHERE
state=‘NY’ AND salary > 100000
UNION ALL
SELECT emp_id, fname, lname, salary FROM employees WHERE
state !=‘NY’ AND salary > 30000;
```

SQL INTERSECT

- Only returns values that appear in both SELECT statements

SELECT job_title, salary FROM employees WHERE state=‘NY’ AND salary > 100000

INTERSECT

SELECT job_title, salary FROM employees WHERE state !=‘NY’ AND salary > 30000;

SQL EXCEPT

- Returns values that only appear in one of the result sets

```
SELECT job_title, salary FROM employees WHERE
state=‘NY’ AND salary > 100000
EXCEPT
SELECT job_title, salary FROM employees WHERE state !=‘NY’ AND
salary > 30000;
```

Nested Queries

- SELECT * FROM salesperson WHERE dept_no IN

  ```
  ( SELECT dept_no FROM dept
  WHERE dept_name LIKE ‘Lo%’ )
  ```

- The subquery runs first, then the outer query runs on the result from the subquery

SQL Views

- Sometimes thought of as a pseudo-table – definition of how to create the view from existing tables/views
- Is essentially a SELECT statement that we can easily reuse without having to type out the whole thing

SQL Views

- CREATE VIEW viewname(col1, alt_col2, calc_col) AS
  SELECT col1, col2, (col3 / col4 * 100) AS calc_col
  FROM tablename | altview

- DROP viewname

- There are a few restrictions on using views with INSERT and UPDATE functions…

SQL View Restrictions

- GROUP BY cannot be joined to other tables or have an additional GROUP BY clause
- Cannot use INSERT or UPDATE if it contains DISTINCT or a sub clause such as GROUP BY or HAVING
- Can’t use INSERT if it omits any NOT NULL columns without defaults or contains calculated or aggregate columns

SQL Administration

- GRANT privilege ON tablename TO list [WITH GRANT OPTION]
- GRANT SELECT ON dept TO james WITH GRANT OPTION – james can now use SELECT on the table dept and can grant that permission to other users
- REVOKE privilege ON tablename FROM list [CASCADE] – CASCADE removes additional permissions given due to WITH GRANT OPTION

SQL and Programming

- Many programming languages have some sort of database interface
- You can connect to databases and execute queries to use the results in a program
- Provides additional support beyond SQL

Stored Procedures

- Microsoft SQL Server has T-SQL
- Oracle has PL/SQL
- Most stored procedures are coded in
- MySQL does allow for stored procedures to be defined

-
  ```
  CREATE PROCEDURE updatesalary(percentage DOUBLE)
  BEGIN
          UPDATE employee SET salary = (salary * percentage);
  END
  ```
- CALL updatesalary(0.10);

Thank you

#TrainerJames

Angharad Pitty

Academy Trainer

~~Angharad.Pitty@[illegible].com~~