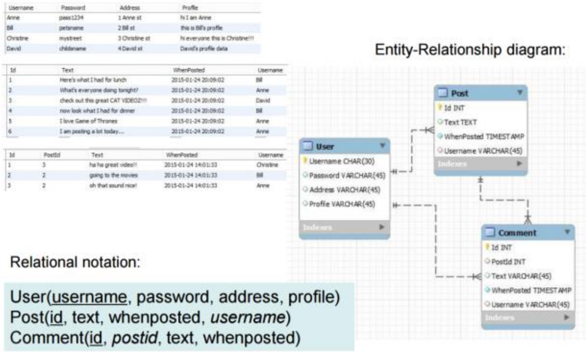
Database Systems and Information Modelling

Week 1

Design a database:

Maintaining contents of a database

- SELECT: read data from the table
- INSERT: new rows into the table
- DELETE: existing rows from the table
- UPDATE: existing row from the table



3 types of database:

Table form

Entity Relationship Diagram

- Use Data Definition Language (**DDL**) to manipulate the structure of the tables
- CREATE, DROP (delete a table), ALTER (add column), RENAME

Relational Notation

Database lifecycle

- Design the database
    - Data modelling, E-R diagrams
- Implement the database
    - Data definition language (**DDL**)
        - Create
        - Drop
        - Alter
        - Rename
- Data access / programming
    - Data manipulation language (**DML**) - CRUD
        - **C**reate (Insert)
        - **R**ead (Select)
        - **U**pdate
        - **D**elete
- Database administration
    - Data control language (**DCL**)
        - Grant
        - Revoke

Noun-Verb analysis:

- **Nouns**: Tables (rows)
- **Verbs:** Describe the entity (relationship between nouns)
    - One employee to one department OR multiple departments
- **Adjectives**

Introduction to MySQL

Statements:

```
show databases;
create database testing;
use testing;
show tables;
create table test1 (column1 int, column2 varchar(30) );
select * from test1;
```

'create' and 'use' only if you are on your own PC (not lab PC)

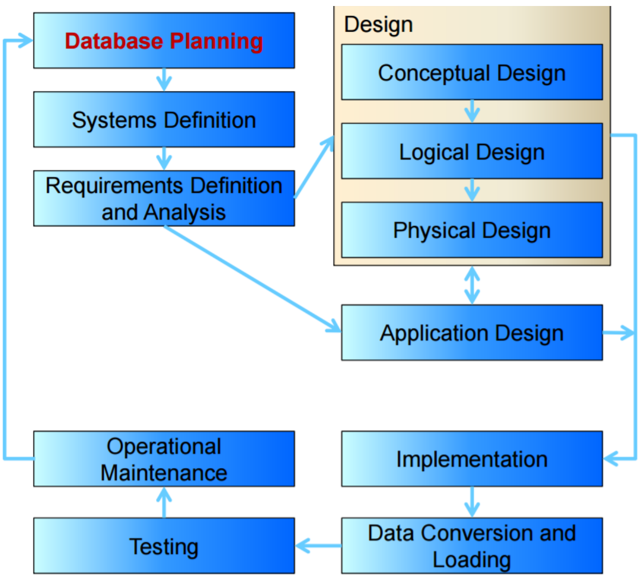
```
select * from test1;
insert into test1 values (1,'my first row');
insert into test1 values (2,'my second row');
select * from test1;
update test1 set column2 = 'my second row, changed' where column1 = 2;
select * from test1;
delete from test1 where column1 = 2;
select * from test1;
```

To clean up what you did:

```
drop database testing;
show databases;
```

Week 2

Database Development Lifecycle



1. **Database Planning**
   - **Outside** scope of the course
   - How does the enterprise work, enterprise data model? PLAN how to do the project

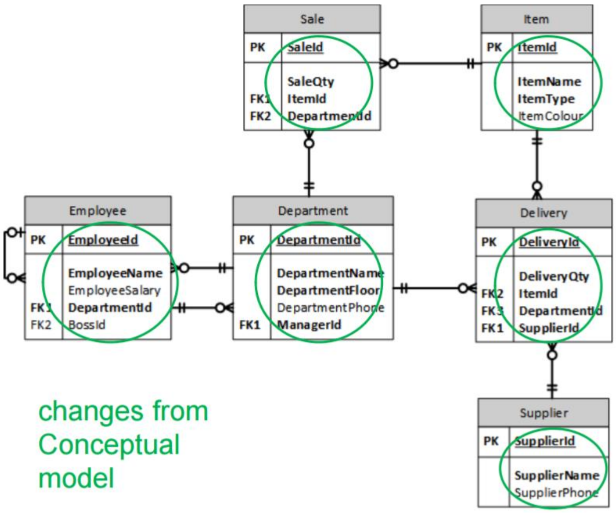
2. **System Definition**
   - Specify scope and boundaries
   - How does it interact with other systems?
   - SLIGHTLY **outside** of the course

3. **Requirements Analysis**
   - Collection and analysis of requirements for new system

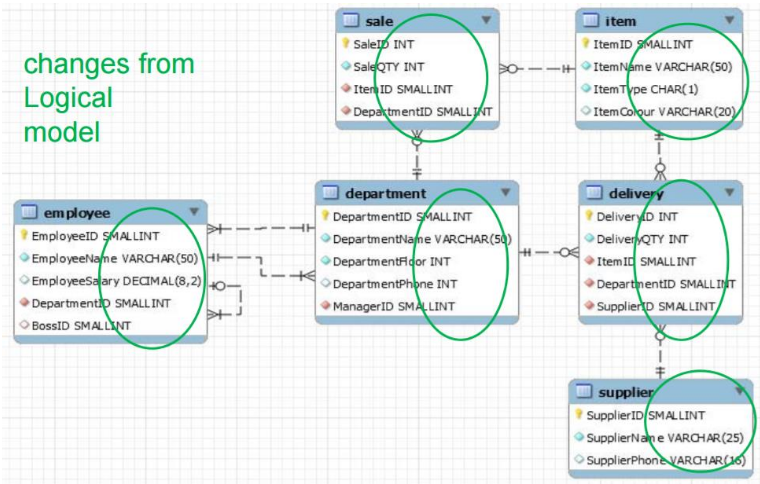
Design

4. **Conceptual Design**
   - High-level, first-pass model of entities and their connections
   - **Omits** attributes
   - Can include many-to-many relationships, repeating groups, composite attributes
     - Could be used in a non-relational database

5. **Logical Design**
- Builds on the conceptual design
- Now for RELATIONAL database
- Includes **columns and keys**
- Independent of a specific vendor and other physical considerations



6. **Physical Design**
- Implements the logical design for a specific DBMS (Database management system)
- Describes
    - Base tables
    - Datatypes
    - Indexes
    - Integrity constraints
    - File organisation
    - Security measures
- Cover some aspects of physical deign



7. **Application Design**
- Done in conjunction with design
- Design of the interface and application programs that use and process the database

8. **Implementation**
- Implementation of the design as a working database

9. **Data conversion and loading**
- Transfer existing data into the database
- Conversion from old systems

10. **Testing**

11. **Operational Maintenance**
- Monitoring and maintaining the database
- Monitoring and improving performance
- Handling changes to requirements

Example:

This database is the central component of an information system used to manage a large city department store. The store has several departments; about each we must record its name and id, phone number, and which floor it is on. Each department has several employees working for it, and one boss (an employee) who manages it.

About each employee we record their name and id, their salary, which department they work for and which other employee is their boss.

The items that the store sells each have a name and id, a type and a colour. Whenever a department sells an item, we record which item was sold, how many were sold, which department sold it – and we give each sale an integer id.

Items are delivered to the store by suppliers, about each of whom we record a name and id, and a phone number. When a delivery is made, we record how many of which item was delivered by which supplier to which department – and deliveries get an integer id too.

Questions to ask:

- What are the entities that need to be tracked?
- What information will be recorded about each entity (attributes)?
- What are the relationships between entities (many to many, one to many, one to one)?
- What are the cardinalities (mandatory or optional) of relationships?

Ans: Examples used above

<u>Implementing and Using Databases</u>

Create Tables:

```
CREATE TABLE Item (
        ItemID          SMALLINT,
        ItemName        VARCHAR (50) NOT NULL,
        …
        PRIMARY KEY (ItemID),
        FOREIGN KEY (…) references Employee(EmployeeID),
);
```

Insert Data:

```
INSERT INTO Department VALUES
        (1, 'Management', 5, 34, 1),
        (8, 'Books', 1, 81, 4),
        …
        (11, 'Marketing', 5, 38, 2);
```

SQL Select statements

```
select * from employee;    /* table dump */
select employeename from employee; /* only one column */
/* only some rows */
select * from employee where employeesalary > 50000;
/* join Employee and Department tables */
select * from employee natural join department order by employeeid;
/* depts on second floor */
select * from department where departmentfloor = 2;
/* sales made by departments on second floor */
select * from sale natural join department
where departmentfloor = 2;
/* items sold by departments on second floor */
select distinct (itemid) from sale natural join department
where departmentfloor = 2;
/* show each employee's boss */
select emp.employeename as Employee, boss.employeename as Boss
from employee emp inner join employee boss
on emp.bossid = boss.employeeid;
```

SQL Update, Insert, Delete statements

```
/* Total salary for employees of department 11 */
Select sum(EmployeeSalary) from Employee where departmentid = 11;
/* award everyone in department 11 a 15% pay rise */
Update Employee
Set EmployeeSalary = EmployeeSalary * 1.15
Where departmentid = 11;
/* add new item */
Select * from item;
Insert into Item values
(21, 'iphone 6', 'P', 'White');
/* check insert */
Select * from Item where ItemType = 'P';
/* delete items of type P */
Delete from Item where ItemType = 'P';
```

Week 3
Data Modelling

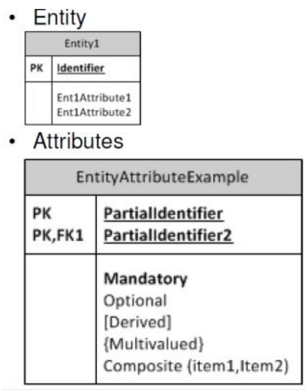
Entity Relationship (ER) Model

Database can be thought as:
- Collection of entity sets
- Relationships between entities

Entity (tables): Object or abstract concept or event which can be distinguished from other entities
- Product, order, sale, person, movie, tweet
- Have **attributes** (your columns) that describe the entity and distinguish it from other entities in the same entity set
    - EmployeeName, Address

Sets: Union, intersection, Cartesian product (when you create every possible pairs between 2 sets)

An entity
- Have many instances in the database
- Several attributes
- Necessary for the system to work
- DO NOT USUALLY INCLUDE
    - An output of the system
    - System itself
    - Company that owns the system



Entity set: Corresponds to a **table** in the database (singular nouns: employee, customer)

Entity Instance: Corresponds to a **row** in a table

Attribute: Corresponds to a **column** in a table (usually a noun: itemColour, quantitySold)

Relationship set: Link between entity sets (verbs or verb phases: has, wants, manages)

Relationship instance: Link between entity instances
- Foreign Key value = primary key value

Key (or identifier): Fully identifies an instance

Partial Key: Partially identifies an instance

Attributes can be:

- Mandatory
- Optional
- Derived
    - Information derived from other attributes. E.g. YearsEmployed -> Smarter to record the date employees joined the company, otherwise need to constantly update
- Multivalued
    - E.g. {Skill} -> many skills or 0, use {} to indicate multivalued
- Composite
    - Name (First, Middle, Last)

KEYS:

Primary Key

- (set of) columns, the value in which **uniquely identify** each instance
- No column can be removed from the key without losing identification
    - E.g. StudentID (PK), StudentName, StudentAddress. The PK helps identify all other attribute. Studentname could be repeated thus does not uniquely identify
- E.g. every primary key to uniquely identify the rows, if the PK is removed and the rest can still do it, it is not a PK.

Candidate Key

- Set of possible primary keys (typically there is only one)
- Select the primary key from this

Composite Key

- Key made up of more than one attribute
- E.g. for the entity airline flight, use composite key FlightNumber + FlightDate
    - Flight numbers fly multiple times; date doesn't uniquely identify it either.

Foreign Key

- Key used to link to a primary key in another table

NEVER

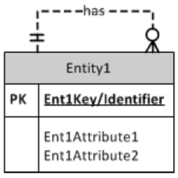
- Unique
- Never null
- DO NOT CHANGE THEIR VALUES

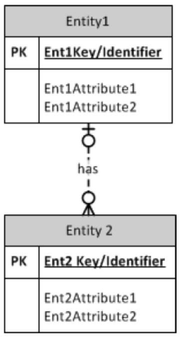
However, selecting PK, entities and attributes depend on **business rules**

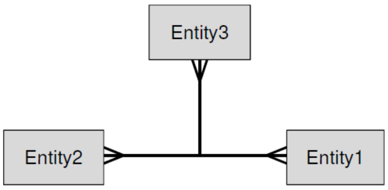
- Business rules are assertions that constrain entities which needs to be captured in your data modelling
    - Business rules depends on individual businesses
- Can impact structure and behaviour of the database
- E.g. "A customer sets up at least one account." (Assertion)
    - Customer (term) & Account (term)
    - Entities identified by the terms (nouns) that the business uses in its literature

Relationship Degree

Unary Binary Ternary

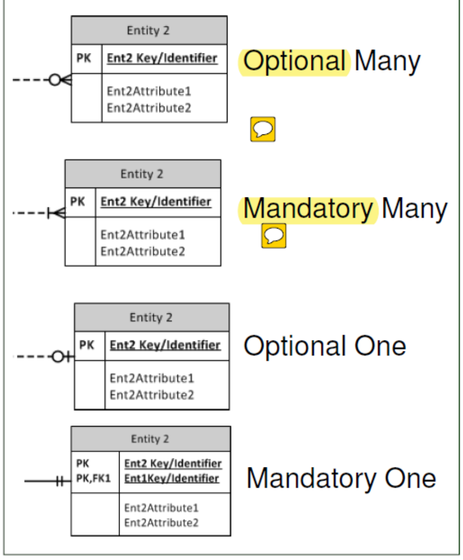






Relationship Cardinality (Ignore dotted lines)

- One to One
  - Each entity in one set is related to 0 or 1 in the other.
- One to Many
  - Each entity in one set is related to many in the other.
- Many to Many
  - Each entity in either set can be related to many in the other set
  - These require an extra step to implement in a relational database.



Optional Many: Can have a customer that has not yet placed an order.

- Some read as 0 to many or optional to many
- Could have several or no orders

Mandatory Many: Does not have any customers **UNLESS** there is a purchase

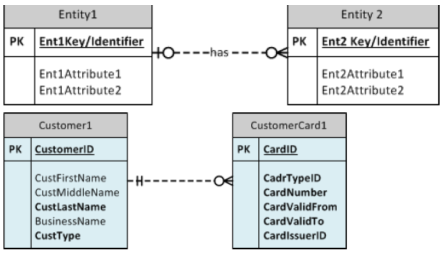
- This depends the BUSINESS RULE: Do people count as customers if they haven't made a purchase?

Strong/Weak entities

- Weak: PK of entity 2 is also PK of entity 1 (THUS replies on each other)

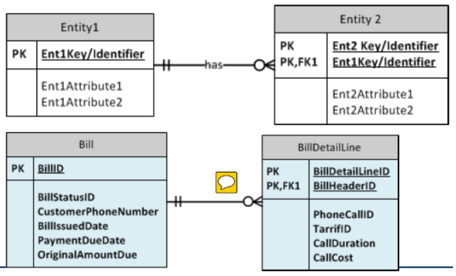
Strong Entity

- Entity 2's identity is independent of the identity of other entities



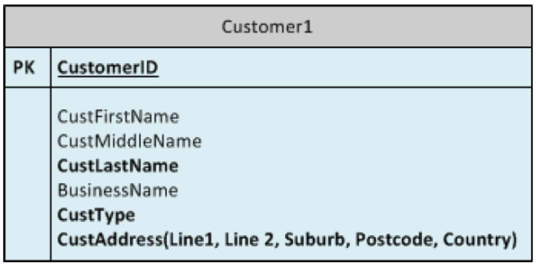
Weak Entity

Entity 2's identity depends on (includes) the identity of Entity 1



Modelling a single entity

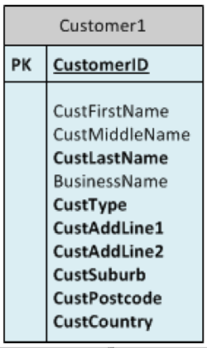
- Searching for nouns in the case, we can identify entities (e.g. customer)
- What things we need to record about the customer (becomes attributes)
- How can we identify individual customers?
    - Name?
    - Address?



Underline = PK
Bold = Not Null
() = composite attribute

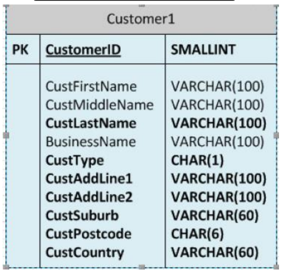
Concert to *logical* design

- Composite attributes become individual attributes
- Multi-valued attributes become a new table
- Resolve many-many relationships via a new table (have a table between the 2 many-many -> becomes one to many with new table)
- **Add FKs at crow foot end of relationships**



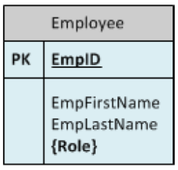
Convert to *physical* design

- Determine data types for each attribute

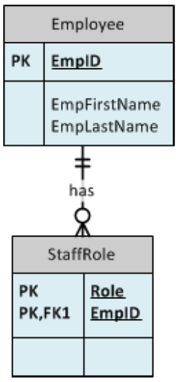


Dealing with multi-valued attributes ({})

Conceptual Design



Logical Design



StaffRole is an example of a *weak entity*

Two entities with 1-M relationship

- Having a FK naturally creates a one to many relationship

Referential Integrity: If you say X is a FK, it won't let you enter something into X UNLESS it is in the other table

- Server prevents you from having bad data

<u>Structured Query Language (SQL)</u>

SQL is a language used to create, access and maintain **relational databases**

- Based on relational algebra and relational calculus
- Supports **CRUD** (Create, Read, Update, Delete)

How do we use it?

- During IMPLEMENTATION of the database

- Implement tables from physical design using DDL Create table

- During PRODUCTION
    - Use Select commands to read the data from the tables
    - Use DML Select, Insert, Delete, Update commands to update data
        - To manipulate and read data in tables
    - Use DDL Alter, Drop commands to update the database structure
    - DCL to control access to the database

The SELECT statement in detail

```
SELECT [ALL | DISTINCT] select_expr [, select_expr ...]
    List the columns (and expressions) that are returned from the query
[FROM table_references
    Indicate the table(s) or view(s) from where the data is obtained
[WHERE where_condition]
    Indicate the conditions on whether a particular row will be in the result
[GROUP BY {col_name | expr } [ASC | DESC], ...]
    Indicate categorisation of results
[HAVING where_condition]
    Indicate the conditions under which a particular category (group) is included
    in the result
[ORDER BY {col_name | expr | position} [ASC | DESC], ...]
    Sort the result based on the criteria
[LIMIT {[offset,] row_count | row_count OFFSET offset}]
    Limit which rows are returned by their return order (ie 5 rows, 5 rows from
    row 2)
]
```

*Group rows, only 1 row is shown. E.g. if we group by Department ID: No longer repeated department ID's

Examples:

SELECT DepartmentID, count (*)
FROM Employee
GROUP BY departmentID
HAVING count (*) = 2;
/*Count (*): counts them all*/

SELECT * FROM Customer;
/*Selects entire content of table*/

SELECT CustLastName, CustFirstname
FROM Customer;
/* Selects specific columns*/

SELECT CustLastName FROM Customer
WHERE CustLastName = "Smith";

SELECT CustLastName FROM Customer
WHERE CustLastName LIKE "Sm%";
/*Needs to have Sm at the start, rest are random. IF %Sm%, basically just needs SM in the name*/

SELECT CustLastName, CustType FROM Customer
ORDER BY CustLastName DESC
LIMIT 5
OFFSET 3;
/*LIMIT: Limits top 5, OFFSET: Ignores first 3 before limiting 5.

SELECT CustType, Count(CustomerID) AS Count
FROM Customer
GROUP BY CustType;
/*This will count CustomerID in the groups of CustType. The Count(CustomerID) will come out as Count on the column name*/

SELECT Custtype, COUNT(CustomerID) FROM Customer
WHERE CustLastName LIKE “Sm%”
GROUP BY CustType
HAVING COUNT(CustomerId) = 3;
/*HAVING works on groups, WHERE works on individual rows*/

INNER JOIN vs NATURAL JOIN

- **Natural Join**: Used more. If 5 years later someone makes a change in the data model, e.g. column name, Natural join would not give error message
  - Requires PK and FK columns to have the same name
  - Do not need to specify where exactly you need to join
  - SELECT * FROM Customer NATURAL JOIN Account;
- **Inner Join**: Join rows where FK value = PK value
  - SELECT * FROM Customer INNER JOIN Account
    ON Customer.CustomerID = Account.CustomerID;

Outer Join:

- Can be left or right
- Includes records from left/right table that don’t have a matching row

```
SELECT * FROM Customer LEFT OUTER JOIN Account
    ON Customer.CustomerID = Account.CustomerID;
```

CustomerID	CustFirstName	CustMiddleNa	CustLastName	BusinessName	CustType	AccountID	AccountName	OutstandingBalance	CustomerID
1	Peter	NULL	Smith	NULL	Personal	1	Peter Smith	245.25	1
2	James	NULL	Jones	JJ Enterprises	Company	2	JJ ENt.	552.39	2
2	James	NULL	Jones	JJ Enterprises	Company	3	JJ ENt. Mgr	10.25	2
3	Akin	NULL	Smithies	Bay Wart	Company	NULL	NULL	NULL	NULL

```
SELECT * FROM Customer RIGHT OUTER JOIN Account
    ON Customer.CustomerID = Account.CustomerID;
```

CustomerID	CustFirstName	CustMiddleName	CustLastName	BusinessName	CustType	AccountID	AccountName	OutstandingBalance	CustomerID
1	Peter	NULL	Smith	NULL	Personal	1	Peter Smith	245.25	1
2	James	NULL	Jones	JJ Enterprises	Company	2	JJ ENt.	552.39	2
2	James	NULL	Jones	JJ Enterprises	Company	3	JJ ENt. Mgr	10.25	2

LEFT: Inner Join would not mention customer 3

- Customer 3 is not included in Account.CustomerID thus would not have been mentioned originally

LEFT takes all the CusomterID from Customer (the LHS of the equation) whereas RIGHT takes all the CustomerID from Account (the RHS of the equation)

- That is why for RIGHT OUTER JOIN, Customer 3 is not included, as it is not included in Account

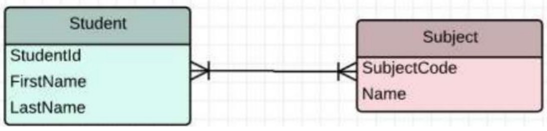
SELECT * FROM Customer, Account;
/*If there are NO JOIN CONDITIONS -> Cartesian product: Every row in Customer is combined with every record in Account) */

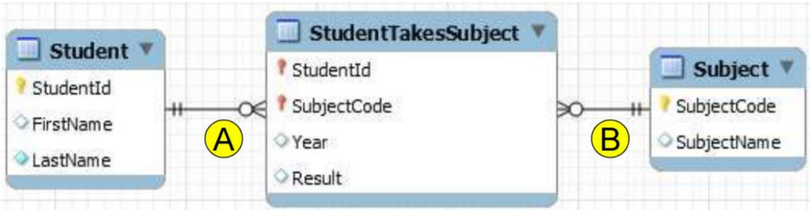
WEEK 4
New Kinds of Relationships:
Domain Integrity
- Valid values and domain (set of values columns can have)
    - Selection of data type is the initial constraint on the data (e.g. int or char)
- Default Value
    - Takes this value if no explicit value is given on Insert
- Null Value Control
    - Allows or prohibits empty fields
- Check constraint
    - Limits range of allowable values (not available in MySQL)

Entity Integrity Constraints
- Primary Key cannot be null
- No component of a composite key can be null
- Primary key must be unique

Each non-null FK value MUST match a PK value
- Rules for update and delete
    - RESTRICT
        - Don't allow deletes or updates of the parent table if related rows exist in the child table
    - CASCADE
        - Automatically delete/update the child table if related rows are delated/updated in the parent table
    - SET NULL
        - Set the FK to NULL in the child table if deleting/updating the key in parent table

Many to Many relationship
Relational database does not directly support M-M relationship
- Number of things would be repeated. From the example below, where do we record who took what subject and their result?
- Create an **associative entity** between the other 2 entities (when converting Conceptual to Logical model) -> HELPS when we need to add additional information that doesn't fit
    - Associative Entity: An entity type that associates the instances of one or more entity types and contains attributes that are peculiar to the relationship between these entity instances
- Each of these 2 relationships is like any 1-M relationship





- Each of these 2 relationships is like any 1-M relationship
    - We can add attributes to the associative entity to record when the student took the subject and the result they god

Associative Entities

WHEN TO CREATE

- Conceptual to Logical
- Implement a many-to-many relationship
- Implement a ternary relationship (3 entities)

The associative entity:

- Independent meaning
- Unique identifier, combination of FKs (FK combining other tables)
- May participate in other relationships

CREATE STATEMENTS (M-M)

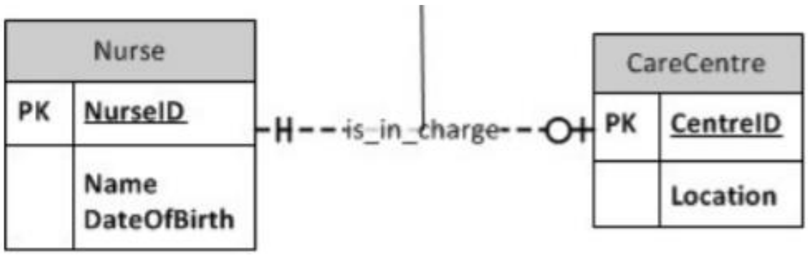
Order of creation + deletion is important

- Create tables WITHOUT FKS first
- Delete tables WITHOUT FKS first
- In the example above, Student and Subject first, StudentTakesSubject last (the middle table needs Student and Subject to form its 2 FKs)
- Insert into the JOIN TABBLE LAST

SQL Code for 3 table join:

SELECT * FROM Student NATURAL JOIN StudentTakesSubject NATURAL JOIN Subject;

Binary One-One relationship

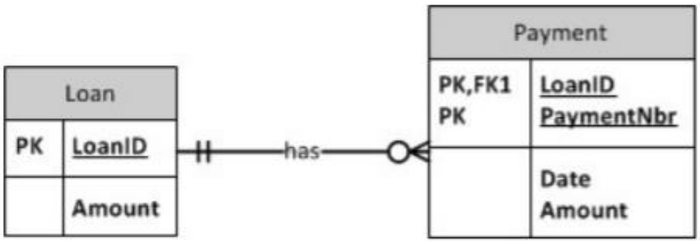


- In a case like this, need to decide where to put the FK
    - Where would the least NULL values be?
    - **RULE**: OPTIONAL side of the relationship gets the FK

1-M Special Case – Identifying Relationship

How to deal with an **identifying relationship?**

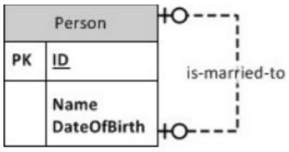
- Relationship between weak child and strong parent tables (Solid line)
- FK defines the relationship at the crows foot end
- The difference = FK becomes part of the PK



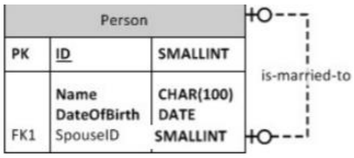
Unary relationships

When the table has a relationship with itself (e.g. an Employee table which includes the boss of each employee. If you are a boss, you are also an employee)

- One-to-One
    - Put the FK in the relation
- One-to-Many
    - Put the FK in the relation
- Many-to-Many
    - Create an extra table – Associative Entity
    - Put 2 FK
        - Need different names for the 2 FKs
        - FKs become the combined PK of the associative entity

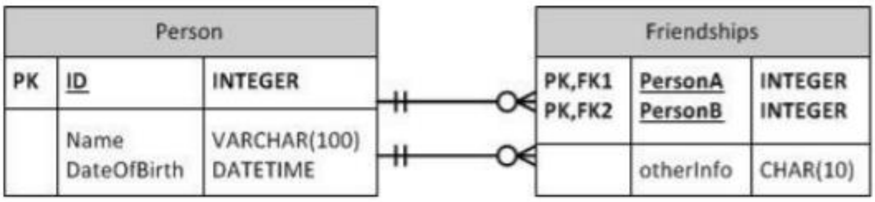


Logical Design



Physical Design

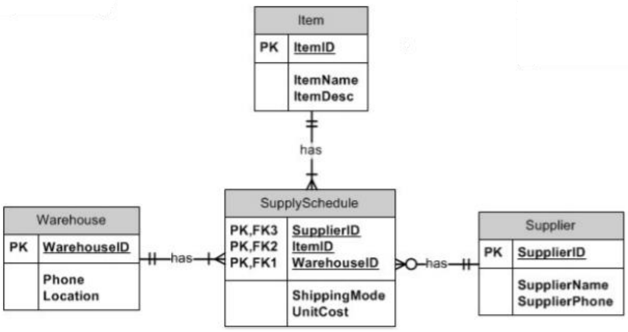
IF M-M:



Ternary Relationships

Relationships between 3 entities

- Generate an associative entity
- Set up 3 1-M relationships



MySQL

Better output by using aliases:

SELECT A.ID, A.Name AS Artefact, …, S.SellerID, S.Name as Seller, S.Phone AS SellerPhone
FROM Artefact A
INNER JOIN Offer O ON A.ID = O.ArtefactID
INNER JOIN Seller S ON S.SellerID = O.SellerID;

THIS allows you to shorten the names of entities

Subqueries / Nested Queries

Select allows you to do nest sub-queries inside the main query

- Nested query is another select query you write to produce a table of data
  - Helps to perform tests
  - Put subquery inside brackets

SELECT DISTINCT ItemID FROM Sale
WHERE DepartmentID IN
(SELECT DISTINCT ….)

Common Operators for Subqueries

- **IN/ NOT IN**
  - Is the value a member of the set returned by the subquery?
- **ALL**
  - True if all values returned meet the condition7
- **WHERE [NOT] EXISTS**
  - True if the subquery yields any results

- Which Artefacts don't have offers made on them

```
SELECT * FROM Artefact
    WHERE ID NOT IN
        (SELECT ArtefactID FROM Offer);
```

ID	Name	Description
3	Pot	Old Pot

- Which Buyers haven't made a bid for Artefact 3

```
SELECT * FROM Buyer
    WHERE BuyerID NOT IN
        (SELECT BuyerID FROM Offer
            WHERE ArtefactID = 3);
```

BuyerID	Name	Phone
1	Maggie	0333333333
2	Nicole	0444444444
3	Oleg	0555555555

- Which Buyers haven't made a bid for the "Pot" Artefact

```
SELECT * FROM Buyer
    WHERE BuyerID NOT IN
        (SELECT BuyerID FROM Offer
            WHERE ArtefactID IN
                (SELECT ID FROM Artefact
                    WHERE Name = "Pot"));
```

BuyerID	Name	Phone
1	Maggie	0333333333
2	Nicole	0444444444
3	Oleg	0555555555

Aggregate functions:

- AVG ()
- COUNT ()
- MIN ()
- MAX ()
- SUM ()

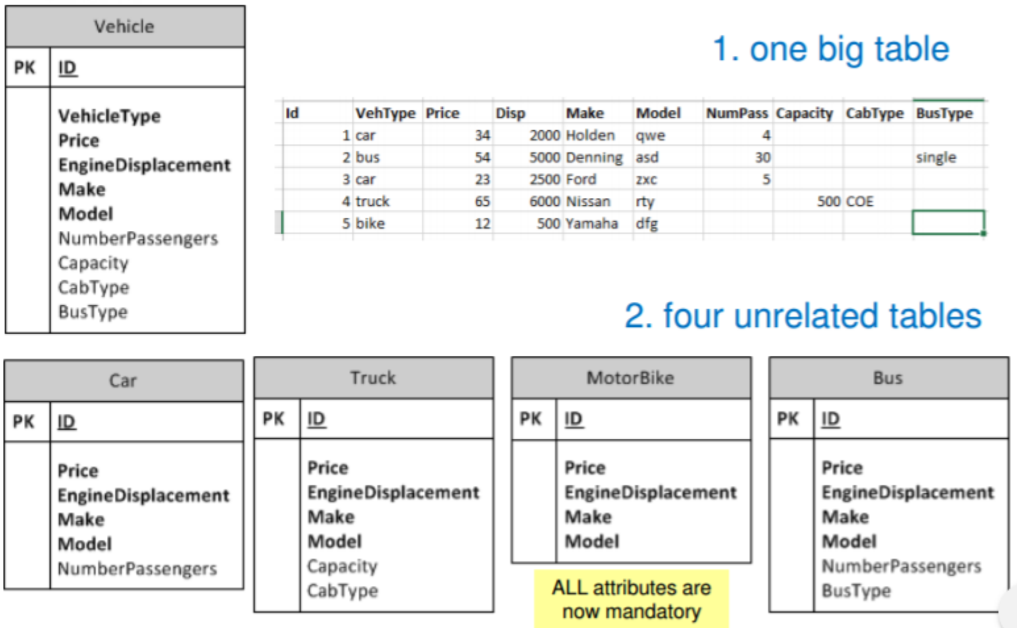
ALL except COUNT () **ignores null values** and return null if all values are null

- COUNT () counts the rows not the values and thus even if the value is NULL it is still counted

Week 5:

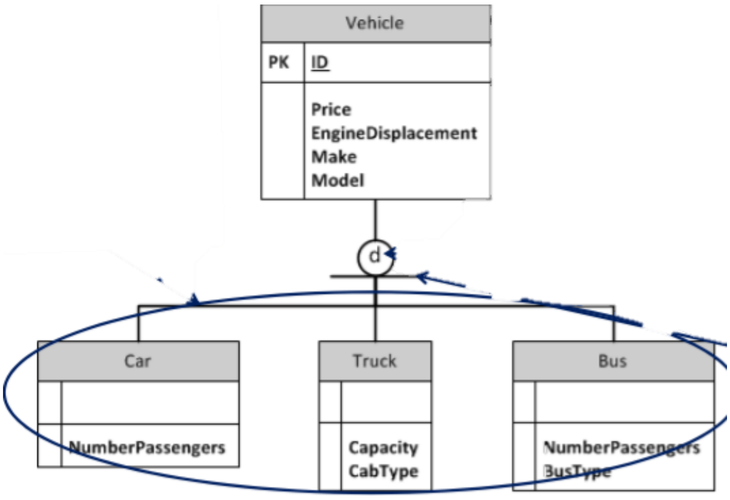
Subtypes and supertypes

Without subtyping, often will get a lot of repeated columns.



Id	VehType	Price	Disp	Make	Model	NumPass	Capacity	CabType	BusType
1	car	34	2000	Holden	qwe	4			
2	bus	54	5000	Denning	asd	30			single
3	car	23	2500	Ford	zxc	5			
4	truck	65	6000	Nissan	rty		500	COE	
5	bike	12	500	Yamaha	dfg				

With subtyping:

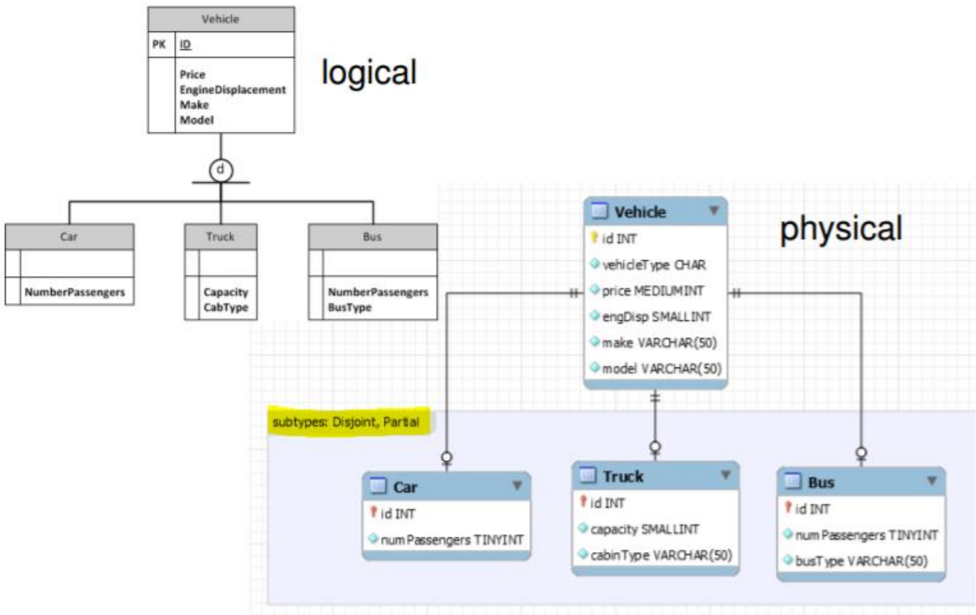


- Each of the subtypes inherits all of the attributes of the supertype (Vehicle)
- Motorbike disappears as a Vehicle could either be a Car, Truck, Bus or none of them
- 'd' means disjoint: ONLY be one of these
- Single line under the 'd' says needn't be any

SUBTYPE types:

- Disjointness Constraints: 'd' or an 'o'
  - 'd' = disjoint (can only be one of those)
  - 'o' = overlapping (can be more than one of these)
- Completeness Constraints – specifies whether an instance of a supertype must also be an instance of a subtype (OR partial)
  - Double line: entity of type subtype1 MUST also be a subtype
  - Single line: Doesn't need to be one of the subtypes

An entity CAN have relationship with 1 of the subtypes



SELECTING from entities with subtypes

- List the ID, Name, Hire Date, Employee Type and Pay of all employees

```
SELECT  Employee.ID, Employee.Name, DateHired,
        EmployeeType, HourlyRate AS Pay
        FROM Employee INNER JOIN Hourly
        ON  Employee.ID = Hourly.ID
UNION
SELECT  Employee.ID, Employee.Name, DateHired,
        EmployeeType, AnnualSalary  AS Pay
        FROM Employee INNER JOIN Salaried
        ON  Employee.ID = Salaried.ID
UNION
SELECT  Employee.ID, Employee.Name, DateHired,
        EmployeeType, BillingRate  AS Pay
        FROM Employee INNER JOIN Consultant
        ON    Employee.ID = Consultant.ID;
```

ID	Name	DateHired	EmployeeType	Pay
3	Alice	2012-12-02	H	23.43
4	Alan	2010-01-22	H	29.43
1	Sean	2012-02-02	S	92000.00
2	Linda	2011-06-12	S	92300.00
5	Peter	2010-09-07	C	210.00
6	Rich	2012-05-19	C	420.00

Combining all subtypes into 1 table

```
SELECT e.ID, e.Name, e.Address, DateHired, DateLeft,
       EmployeeType, ContractNumber, BillingRate,
       AnnualSalary, StockOption, HourlyRate
   FROM Employee e
   LEFT OUTER JOIN Hourly h ON e.ID = h.ID
   LEFT OUTER JOIN Salaried s ON e.ID = s.ID
   LEFT OUTER JOIN Consultant c ON e.ID = c.ID;
```

(requires Outer Join, but you get all the data with one query)

ID	Name	Address	DateHired	DateLeft	EmployeeType	ContractNumber	BillingRate	AnnualSalary	StockOption	HourlyRate
1	Sean	Sean's Address	2012-02-02	NULL	S	NULL	NULL	92000.00	N	NULL
2	Linda	Linda's Address	2011-06-12	NULL	S	NULL	NULL	92300.00	Y	NULL
3	Alice	Alice's Address	2012-12-02	NULL	H	NULL	NULL	NULL	NULL	23.43
4	Alan	Alan's Address	2010-01-22	NULL	H	NULL	NULL	NULL	NULL	29.43
5	Peter	Peter's Address	2010-09-07	NULL	C	19223	210.00	NULL	NULL	NULL
6	Rich	Rich's Address	2012-05-19	NULL	C	19220	420.00	NULL	NULL	NULL

SQL Wrap-up

Things to remember:

- SQUL keywords are case insensitive
  - Traditional convention is to CAPITALISE them for clarity
- Field names are case insensitive
  - ACCOUNTID == AcountID == AcCoUnTID

Comparison and Logic Operators:

- =, <, >, <=, >=, <> OR != (not equal to)
- LOGIC
  - AND, NOT, OR

SELECT * FROM Furniture
    WHERE ((Type = "Chair" AND Colour = "Black")
    OR (Type = "Lamp" AND Colour = "White"))

Set Operations

UNION: Show all rows returned from the queries

[UNION] ALL: If you want duplicate rows shown in the results you need to use the ALL keyword… e.g. UNION ALL

FORMAT ()

- Changes format of output of Select
- E.g. FORMAT (N, D)
  - N: A number which may be an integer, decimal or a double
  - D: How many decimals the output contains (N)

CAST ()

- Changes data type of output
- CAST (Expression AS Type)
  - E.g. CAST ("1234.55" AS UNSIGNED) gives 1234

HOWEVER: FORMAT them in terms of strings, so e.g. if you had 12.23, 13.12, 14.55, 100.323 and after using format, you wish to order by DESC, 100.323 (after formatted) would be at the very bottom.

- THEREFORE: Use CAST (Expression AS DECIMAL (x, y))
  - X = number of space (includes.)
  - Y = number of decimal place after.

```sql
SELECT Department.DepartmentID, FORMAT(SUM(EmployeeSalary*Bonus),2) AS TotSalary
FROM Department INNER JOIN Employee ON Department.DepartmentID = Employee.DepartmentID
GROUP BY Department.DepartmentID
ORDER BY TotSalary DESC;
```

DepartmentID	TotSalary
9	99,000.00
1	67,500.00
2	60,000.00
10	35,000.00
3	32,640.00
4	27,040.00
7	16,500.00
8	15,150.00
6	15,000.00
5	15,000.00
11	101,200.00

wrong

```sql
SELECT Department.DepartmentID, CAST(SUM(EmployeeSalary*Bonus) AS DECIMAL(9,2)) AS TotSalary
FROM Department INNER JOIN Employee ON Department.DepartmentID = Employee.DepartmentID
GROUP BY Department.DepartmentID
ORDER BY TotSalary DESC;
```

DepartmentID	TotSalary
11	101200.00
9	99000.00
1	67500.00
2	60000.00
10	35000.00
3	32640.00
4	27040.00
7	16500.00
8	15150.00
6	15000.00
5	15000.00

These are numbers, so ordering works again

IFNULL ()

- Can convert a null to a 0 (can be useful in calculations)
- SELECT 1 + IFNULL (wagevalue, 0)
- Gives 1 + 0 for null fields and 1 + wage value for non-null fields
- Failure to do this results in a null answer for values where wage value is NULL

UPPER () / LOWER ()

- Change string to upper / lower case

LEFT ()

- Returns the leftmost X characters from the string
- SELECT LEFT (“This is a test”, 6)
- Gives “This I”

RIGHT ()

- SELECT RIGHT (“This is a test”, 6)
- Gives “a test”

More on INSERT:

- Insert records from another table
    - INSERT INTO NewEmployee SELECT * FROM Employee;
        - Emplyoee must already exist
- Insert multiple rows:
    - INSERT INTO EMPLOYEE VALUES
      (DEFAULT, “A”, “A’s Addr”, “2012-02-02”, NULL, “S”),
      (…);
    - INSERT INTO Employee
      (Name, Address, Datehired, EmployeeType)
      VALUES
      (“D”, “D’s Addr”, “2012-02-02”, “C”),
      (…);

More on UPDATE:

```
UPDATE Hourly
	SET HourlyRate = HourlyRate * 1.10;
```

- Increase salaries greater than $100K by 10% and rest 5%

```
UPDATE Salaried
	SET AnnualSalary = AnnualSalary * 1.05
	WHERE AnnualSalary <= 100000;
UPDATE Salaried
	SET AnnualSalary = AnnualSalary * 1.10
	WHERE AnnualSalary > 100000;
```

BUT: This method is very slow as we are rerunning and testing every row twice. Change to:

```
UPDATE Salaried
	SET AnnualSalary =
		CASE
			WHERE AnnualSalary <= 100000
			THEN AnnualSalary * 1.05
			ELSE AnnualSalary * 1.10
		END;
```

REPLACE

- REPLACE works identically as INSERT
    - Except if an old row in a table has a key value the same as the new row, then it is overwritten

DELETE

- DELETE FROM Employee; (deletes all rows from employee)
- DELETE FROM Employee WHERE Name = "Grace"
    - If you delete a row that has rows in other tables dependent on it
        - The dependent rows are deleted too, or
        - The dependent rows get 'null' or a default, or
        - Your attempt to delete is blocked

VIEWS:

A View is a select statement that persists, and can be treated as though it were a table by other SQL statements

USED TO:

- Hide the complexity of queries from users
- Hide structure of data from users
- Hide data from users
    - Different users use different views (someone to access employee table but not salaries column)
    - One way to improve database security

To create a view:

- CREATE VIEW nameofview AS validSelectStatement

```
create view DepartmentSales as
select departmentid, departmentname, count(*) as numSales
from Department natural join Sale
group by departmentid;
```

```
select * from DepartmentSales;
```

departmentid	departmentname	numSales
3	Clothes	5
4	Equipment	3
5	Furniture	2
6	Navigation	5
7	Recreation	4
8	Books	7

```
select * from DepartmentSales
where numSales > 5;
```

departmentid	departmentname	numSales
8	Books	7

When can we Update or Insert a view?

- Select clause only contains attribute names
    - Not expressions, aggregates or distinct
- Any attributes not listed in the select clause can be set to null
- The query does not have a group by or having clause

MORE DDL commands (do we need to know them?)

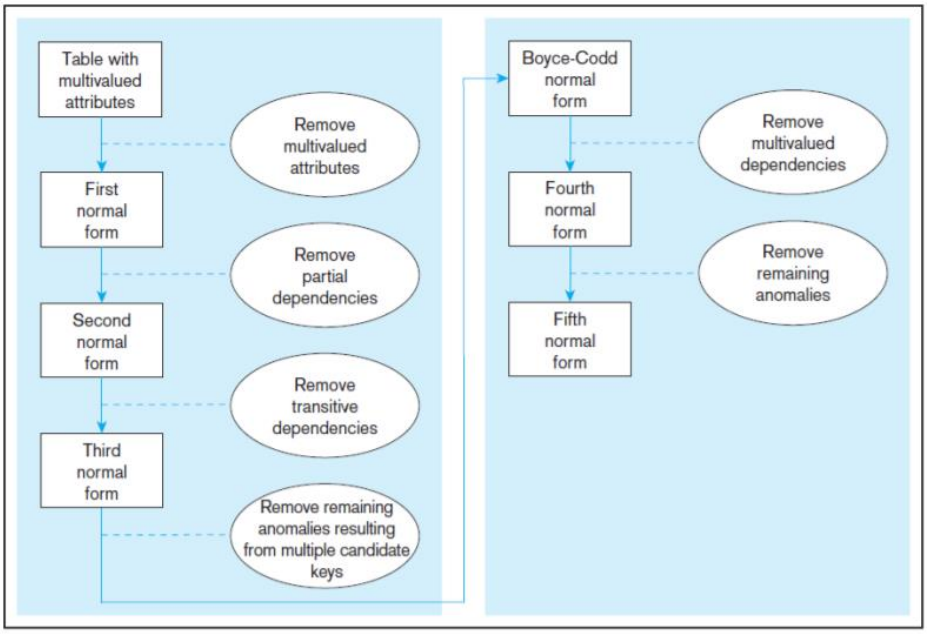
- **ALTER**: Allows us to add/remove columns from a table
    - ALTER TABLE TableName ADD AttributeName AttributeType
    - ALTER TABLE Tablename DROP AttributeName
- **RENAME:** Allows the renaming of tables
    - RENAME TABLE CurrentTableName TO NewTableName
- **TRUNCATE:** Like DELETE FROM table but does more (http://stackoverflow.com/questions/139630/whats-the-differencebetween-truncate-and-delete-in-sql)
- **DROP:** Removes the table definition and the data in the table
    - DROP TABLE TableName

<u>Week 6: Normalisation</u>

Normalisation is:

- Theoretical foundation for the relational model
- **Last phase** of logical design
    - Prevent date redundancy
    - Prevent Update anomalies

Normal form: State of a relation resulting from applying rules about the functional dependency of some attributes onto others (We go from 1st NF to Boyce-Codd NF)



Definitions:

Functional Dependency: X determines another set of attributes Y IFF each value of X is associated with ONLY one value of Y

- X -> Y (like y = f(x))
- Each x value only gives 1 y value, but y values can repeat

Determinants: Attributes on the LHS of the arrow (X in example above)

Key and Non-Key attributes: Each attribute is either part of the primary key or it is not

Partial Functional dependency: Functional dependency of one or more non-key attributes upon *part (but not all)* of the PK

Transitive dependency: Functional dependency between 2 (or more) non-key attributes

Functional dependency:
Relationship between attributes of a relation

- One or more attributes determine the value of another attribute
- PK must functionally determine ALL non-key attributes of an entity
- E.g. studentID, subjectTitle -> dateCompleted, resultObtained
    - For ANY COMBINATION of studentID and subjectTitle, only ONE dateCOmpleted and ONE resultObtained
- Only PK can be determinants (attribute(s) that determine another attribute(s))

1st NF: ANY **multivalued** attributes and repeating groups have been removed. There is a SINGLE value (possible null) at the intersection of each row and column of the table

2nd NF: ANY **partial** functional dependencies have been removed (i.e. all non-key attributes are identified by the WHOLE key)

3rd NF: Any **transitive** dependencies have been removed (i.e. non-key attributes are identified by ONLY the PK)

BCNF (NOT ASSESSED): All remaining anomalies that result from functional dependencies have been removed (i.e. because there was more than one candidate PK for the same non-keys)

EXAMPLE:
Write in relational notiation, list all attributes as one big relation. Don't include anything you can derive and use () to indicate repeating groups

- R1 (<u>InvoiceNo,</u> Date, CustomerNo, CustomerName, CustomerAddress, ClerkNo, ClerkName, (ProductNo, ProductDesc, UnitPrice, Qty))
- ProductNo. And rest can repeat

1st NF:
If an attribute is multi-valued, group of attributes is related several times for one entity, create a new table:

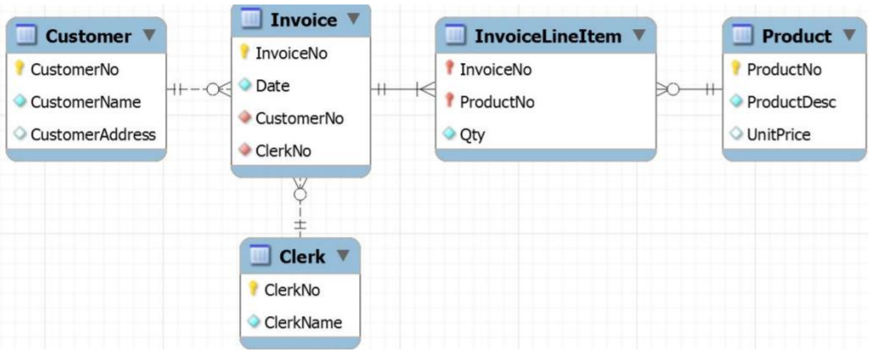
- R1 (<u>InvoiceNo</u>, Date, CustomerNo, CustomerName, CustomerAddress, ClerkNo, ClerkName)
- R2 (<u>*InvoiceNo*</u>, <u>ProductNo</u>, ProductDesc, UnitPrice, Qty)
- FK get italics

2nd NF:
Relevant only for relations whose PK is a COMPOSITE key. Remove PARTIAL FUNCTIONAL DEPENDENCIES (an attribute is dependent on only PART of the PK)

- R1 doesn't have a composite key, so already 2NF
- R2 not 2NF because ProductDesc and UnitPrice are determined by ProductNo (PART of the PK)
- R1 (<u>InvoiceNo</u>, Date, CustomerNo, CustomerName, CustomerAddress, ClerkNo, ClerkName)
- R2 (<u>*InvoiceNo*</u>, <u>*ProductNo*</u>, Qty)
- R3 (<u>ProductNo</u>, ProductDesc, UnitPrice)

3rd NF: Remove function dependency where one non-key attribute is functionally dependent on another non-key attribute

- R1 (<u>InvoiceNo</u>, Date, *CustomerNo*, *ClerkNo*)
- R2 (<u>*InvoiceNo*</u>, <u>*ProductNo*</u>, Qty)
- R3 (<u>ProductNo</u>, ProductDesc, UnitPrice)
- R4 (<u>CustomerNo</u>, CustomerName, CustomerAddress)
- R5 (<u>ClerkNo</u>, ClerkName)



IF we didn't normalise:

- Repeated data (individual facts stored many times)
- Update anomalies (change a fact, must change many times)
- Deletion anomalies (information about entity A is stored inside entity B, delete all B rows, lose record of A)
- Insertion anomalies (record a new A, must insert a B)

BCNF (not assessed): Not every determinant in the relation is a candidate key

- Check whether "every non-key attribute must provide a fact about the key, the whole key and nothing but the key"

<u>Physical Database Design</u>

Purpose – translate the logical description of data into the technical specifications for storing and retrieving data on disk

Goal – Create a design for storing data that will provide good performance and insure database integrity, recoverability and security

Inputs

- Normalised data model
- Attribute definitions
- Volume estimates
- Response time expectations
- Data security needs
- Backup/recovery needs
- Integrity expectations
- DBMS used

leads to

Decisions

- Attribute data types
- Physical record descriptions (doesn't always match logical design)
- File organisations
- Indexes
- Query optimisation

Choosing data types:
Column: Smallest unit of data in database
- Datatypes helps DBMS to store and use information efficiently
- Try to **minimise** storage space
- Types can affect performance e.g. fixed or variable length
- Must be able to represent all possible values
- Improve data integrity (quality)
- Support all required data manipulations

CHAR(M): A fixed-length string that is always right-padded with spaces to the specific length when stored. Range of M is 0 to 255. E.g. CHAR (10): 10 spaces
CHAR: Synonym for CHAR (1)
VARCHAR (M): A variable-length string. Only characters inserted are stored (no padding). Range 1 -> 65535 characters (DISADVANTAGE: Cannot add extra characters after initial storage)
BLOB, TEXT: A binary or text object with a maximum length of 65535 (2^16) bytes (blob) or characters (text)
LONGBLOB, LONGTEXT: A BLOB or TEXT column with a maximum length of 4,294,967,295 characters
ENUM (‘value1, ‘value2’…) up to 65,535 members

Integer types:
TINYINT: Signed (-128 to 127), Unsigned (0 to 255)
BIT, BOOL: Synonyms for TINYINT
SMALLINT: Signed (-32768 to 32767), unsigned (0 to 65535)
MEDIUMINT: Signed (-8388608 to 8388607), Unsigned (0 to 16777215)
INT / INTEGER: Signed (-2147483648 to 2147483647), Unsigned (0 to 4294967295)
BIGINT: Signed (-9223372036854775808 to 9223372036854775807), Unsigned (0 to 18,446,744,073,551,615)

Real Types:
Float: single-precision floating point, allowable values: -3.4E+38 to -1.17E-38, 0, to 1.17E-38 to 3.4E+38
Double / REAL: double-precision: 1.79E+308 to -2.22E-308, 0, and 2.2E-308 to 1.8E+308

M = display width, D = numbers of decimals
Float and Double often used for scientific data
DECIMAL (M, D): Fixed-point type, good for money values

DATE: 1000-01-01 to 9999-12-31
TIME: -838:59:59 to 838:59:59 (time of day or elapsed time)
DATETIME: 1000-01-01 00:00:00 to 9999-12-31 23:59:59
TIMESTAMP: 1970-01-01 00:00:00 to 2037
YEAR: 1901 to 2155

Data integrity:
- Default value: assumed value if no explicit value given
- Range control: allowable value limitations (constraints or validation rules)
- Null value control: Allowing or prohibiting empty columns
- Referential integrity: Checks values (and null value allowances) of foreign-key

Indexing columns:
Similar to an index in pages, contains pointers to table rows for fast retrieval
- Can choose which columns to index
- PKs and FKs are automatically indexed

WHEN?
- Use larger tables
- On columns which are frequently in WHERE clauses or in ORDER BY and GROUP BY commands
- DON'T USE IF: Limit the use of indexes for **volatile** databases
    - Data frequently changed, when table data changed, indexes need to be updated

De-normalisation
NORMALISATION:
- Removes data redundancy
- Solves insert, update, and delete anomalies
- Makes it easier to maintain information in a consistent state

HOWEVER
- Leads to more tables in the database
- Often need to be joined back together during selects -> expensive
- Sometimes we decide to 'de-normalize'

De-normalise if:
- Database speeds are unacceptable
- Going be very few INSERTs, UPDATEs, DELETEs
- LOTS of SELECTs

<u>Week 7 – Databases in applications</u>
Limitations of MySQL
- Cannot express all possible queries in SQL (easier to say in words but hard to program)
- Need to enforce business rules beyond domain/ref integrity
- Need procedural constructions such as loops and decisions
- Would you give end-users a query browser?
- Need a user interface that is friendly and constraining (what the customers will use)

Business logic: Check user name and password, if good, login, if bad, error
- Procedural programming can do (e.g. java) -> such as sequence, iterations, control flow
- SQL specialised for **low-level data access**

THEREFORE:
Need to combine data manipulation with the ability to handle sequence, iteration, decisions.
Embedded SQL: host language = C, Java etc.
- **SQL** embedded in code is interpreted and replaced with library calls

Dynamic SQL: Host language sends SQL to DBMS via middleware (ODBC etc.)
- Data is passed back to program as record-set
- Host language can handle business and presentation logic

Stored Procedures, Triggers
- Procedural code stored and executed in the DBMS
- Enforce business logic within the database

ADVANTAGE OF Stored procedures and triggers
- Compiled SQL statements
- Faster code execution

- Reduced network traffic
- Faster code execution
- Reduced network traffic
- Improved security and data integration
- Business logic under control of DBA
- Thinner clients

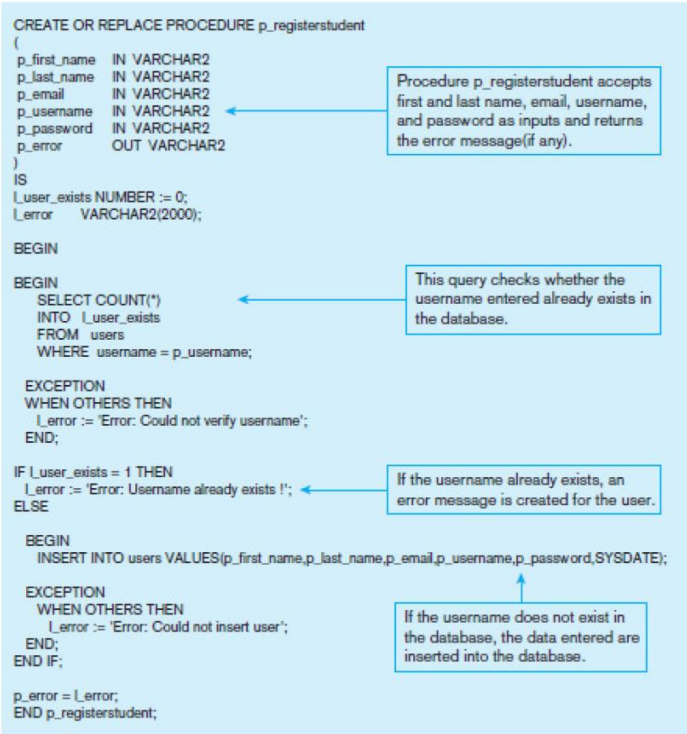
DISADVANTAGE

- Harder to write code
- Proprietary language (E.g. MySQL Stored procedures can't be used in Oracle or SQL server)

EXAMPLES of STORED PROCEDURES:

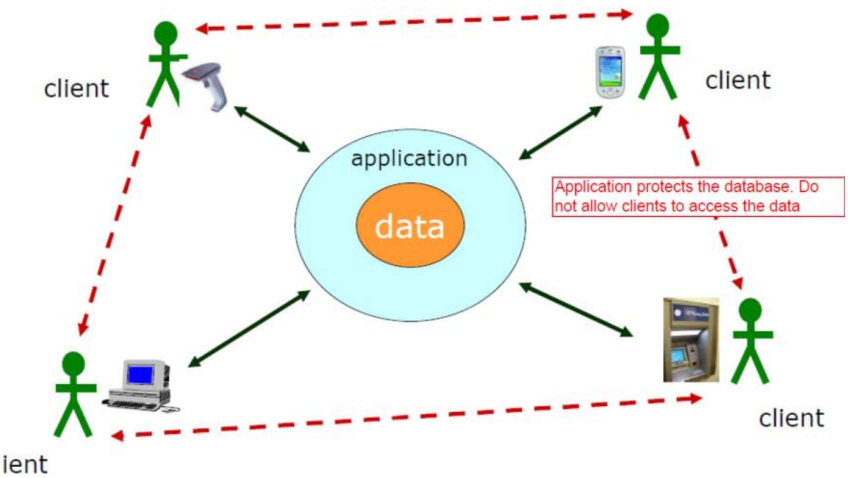
1. accept person details as inputs
2. check whether the person is already in the database
3. if yes, return error
4. if no, add to database

(source: Hoffer chapter 8)



```
CREATE OR REPLACE PROCEDURE p_registerstudent
(
 p_first_name   IN VARCHAR2
 p_last_name    IN VARCHAR2
 p_email        IN VARCHAR2
 p_username     IN VARCHAR2
 p_password     IN VARCHAR2
 p_error        OUT VARCHAR2
)
IS
l_user_exists NUMBER := 0;
l_error       VARCHAR2(2000);

BEGIN

BEGIN
    SELECT COUNT(*)
    INTO   l_user_exists
    FROM   users
    WHERE  username = p_username;

  EXCEPTION
  WHEN OTHERS THEN
    l_error := 'Error: Could not verify username';
  END;

IF l_user_exists = 1 THEN
  l_error := 'Error: Username already exists !';
ELSE

  BEGIN
    INSERT INTO users VALUES(p_first_name,p_last_name,p_email,p_username,p_password,SYSDATE);

  EXCEPTION
    WHEN OTHERS THEN
      l_error := 'Error: Could not insert user';
  END;
END IF;

p_error = l_error;
END p_registerstudent;
```


Application Architectures



System Architecture:

Presentation Logic

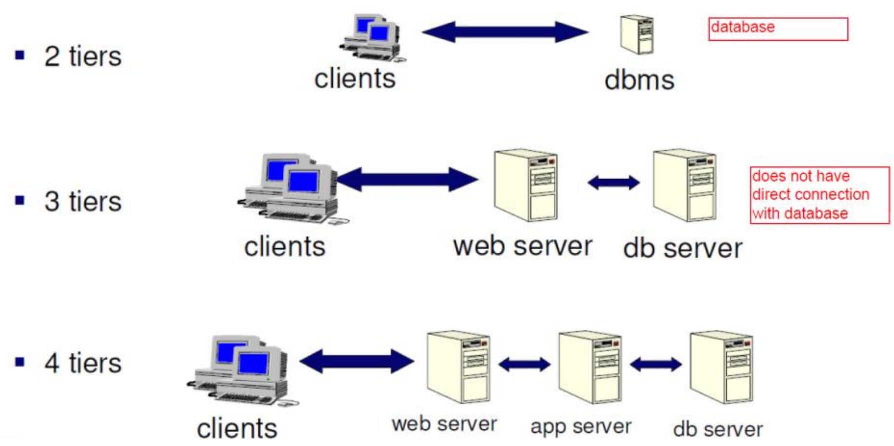
- Input (keyboard, touchscreen, voice etc.)
- Output (large screen, printer, phone, ATM etc.)

Business Logic

- Input and command handling
- Enforcement of business rules

Storage Logic

- Persistent storage of data
- Enforcement of data integrity



Evolution of application architectures:

Main frame/dumb terminal (1st tier)

- One large computer handles all logic
- Problems: Doesn't scale up
- No processing at client end
- Entire application ran on the same computer (database, business logic and user interface)
- Enabling technologies included: embedded SQL, report generators

Client-Server Architecture

- Personal computers become possible (2 and 3 tier)

2nd tier: Server is a relationship DBMS (data storage and access is done at the DBMS)

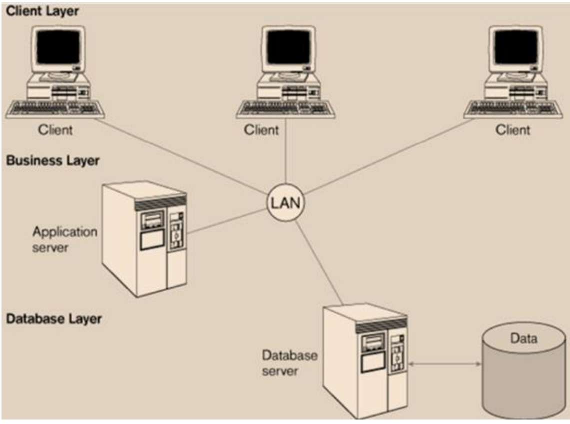
- Presentation, business logic is handled in client application
- Popular till internet. This needed to be installed on every client. Now can use LAN
- **Advantages:**
  - Client and server share processing load
  - Good data integrity since data is all processed centrally
  - Stored procedures allow some business rules to be implemented on the database server
- **Disadvantages:**
- Presentation, data model and business logic are intertwined at client
- If DB schema changes, all clients break
- Updates need to be deployed to all clients
- DB connection for every client, thus difficult to scale
- Difficult to implement beyond the organisation (to customers)

Web Architecture

- Form of 3 or 4 tier

3rd Tier:

- Client program <-> Application Server <-> Database server
- **Presentation logic**: Client handles interface
- **Business logic**: application server deals with business logic
- **Storage logic**: Database server deals with data persistence and access



Advantages:

- Scalability
- Technological flexibility (can change business logic easily)
- Swap out any single component easily
- Long-term cost reduction
- Improved security – customer machine does presentation only

Disadvantages:

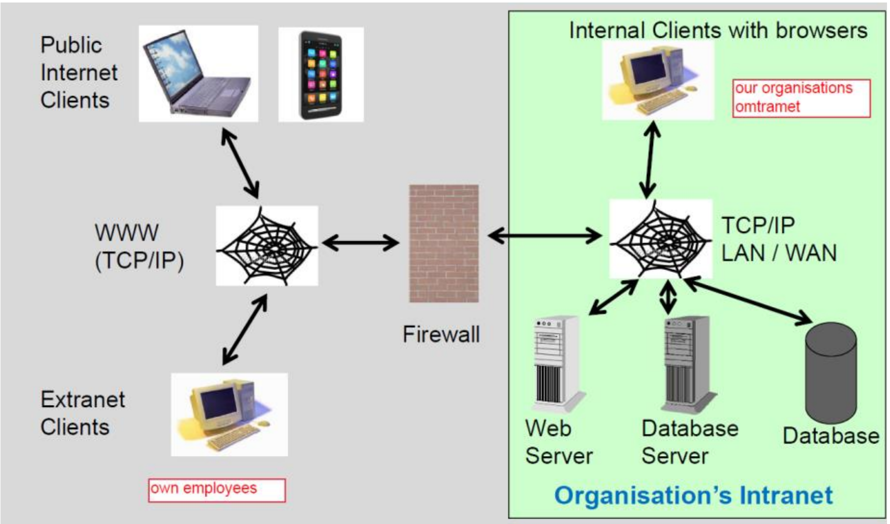
- High short-term costs
- Tools and training
- Complex to design
- Variable standards

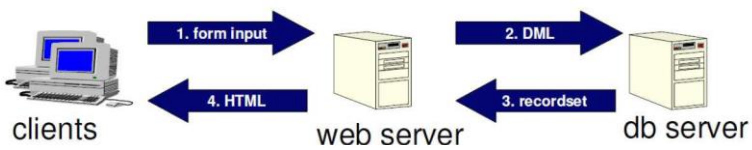
Security in multi-tier applications

Network environment creates complex security issues

- Security can be enforced at different tiers:
    - Application password security for allowing access to the application software
    - Database-level password security for determining access privileges to tables
    - Secure client/server communication via encryption

<u>WEB APPS</u>





Why create web applications:

- Ubiquitous -> anyone can use
- No need to install client software for external customers
- Simply communication protocols
- Platform and Operating System independent
- Reduction in Development time and cost
- Has enabled eGov, eBusiness, eCommerce, B2b, B2C

Web Infrastructure

- Browser: Software that retrieves and displays HTML documents
- Web Server: Software that responds to requests from browsers by transmitting HTML and other documents to browsers
- Web Pages (HTML Documents)
    - Static web pages
        - Content established at development time (manually created)
    - Dynamic web pages
        - Generated using data from database (e.g. facebook's newsfeed)
- World Wide Web (WWW) -> Total set of interlinked hypertext documents residing on web servers worldwide

Web-related languages:

HTML (define a web page), CSS (control appearanace of an HTML document), JavaScript (enable interactivity in HTML documents), Extensible Markup Language (XML – transport data between web services)

HTML:

Structured as a tree (one web page = one tree)

- Divided into a HEAD and a BODY
    - BODY: What is displayed in the browser
    - BODY divided into elements such as headings, paragraphs, tables, lists

Static vs Dynamic web pages:

STATIC:

- URL identifies a file on the server's file system
- Server fetches the file and sends it to the browser
- File contains HML
- Browser interprets the HTML for display on screen

DYNAMIC:

- URL identifies a program to be run
- Web app runs the program
- Program typically retrieves data from database
- Elements such as TABLE, LIST are populated with data

Program logs into db -> selects all rows from database table -> displays them inside an HTML table

Problems with old-style web apps

Placing "raw" SQL inside PHP/HTML files

- Mixes presentation, business logic and database
- Hard to maintain when things change
- Want separation of concerns

Lots of reinvention of wheels:

- Certain things that you need to program from scratch each time (login security, presentation template etc.)
- Each dev writes their own solution to common features

Increasing variety of clients (phones/tablets)

- Manually program for different platforms

Web services:

The WWW allows humans to access databases

- Web Services allow computers to access databases (services are interaction with each other)
- 2 major approaches: SOAP and REST
    - Simple Object Access Protocol
    - Representational State Transfer
- Structured data usually returned in XML or JSON format
- REST nouns are resources, addressed via URLs
- REST verbs correspond to DML (data manipulate language) statements
- GET (select), POST (insert), PUT (update), DELETE (delete)

<u>Week 8 – Transactions and Concurrency</u>

Database Transaction:

- Logical unit of work that must either be **entirely completed or aborted (**indivisible, atomic – succeed or fail)
- DML statements are already atomic
- RDBMS (relational database management system) also allows for *user-defined* transactions
- Successful transaction changes the database from one consistent state to another
    - One in which all data integrity constraints are satisfied

WHY do we need it?

- Users need to the ability to define a unit of work
- Concurrent access to data by >1 user or program (concurrent enables multiple people to do the same update at the same time instead of one at a time)

Also, acts as an "undo" for manual database manipulation (enables rollback)

Problem 1: Unit of Work

Single DML or DDL command:

- E.g. update 700 records, but database crashes after 200 -> you will find NO CHANGES

Multiple statements (user-defined transaction)

- START TRANSACTION ('**BEGIN'**)
    - SQL Statement
    - SQL Statement
- **COMMIT**: (commits the whole transaction) or **ROLLBACK** (undo everything – back to previous commit point)

In the case of an error:

- Any SQL statements already completed MUST BE REVERSED
- Show an error message to the user

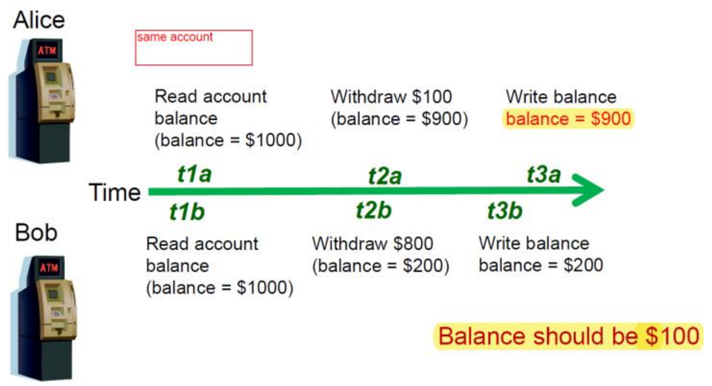
Transaction Properties (ACID):
Atomicity: Transaction is treated as a single, indivisible, logical unit of work. All operations in a transaction must be completed

Consistency: Constraints that hold before a transaction must also hold after it (multiple users accessing the same data see the same value)

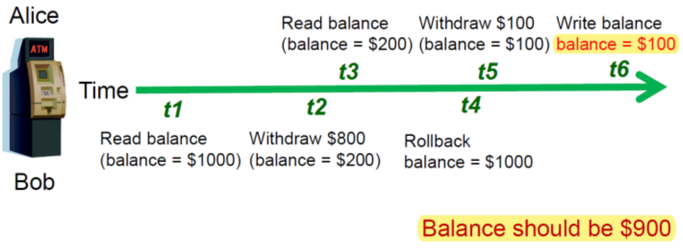
Isolation: Changes made during execution of a transaction CANNOT BE SEEN by other transaction until this one is competed

Durability: When a transaction is complete, the changes made to the database are permanent, even if the system fails

Problem 2: Concurrent Access
What happens if multiple users accessing the database at the same time…
- Lost updates
- Uncommitted data
- Inconsistent retrievals



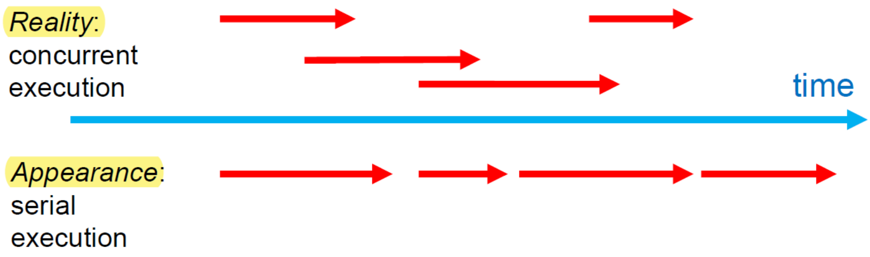
Uncommitted Data problem:
Uncommitted data occurs when two transactions execute concurrently and the first is rolled back after the second has already accessed the uncommitted data.



Inconsistent Retrieval Problem:
When one transaction calculates some aggregate functions over a set of data, while other transactions are updating the data
- Some data may be read after they are changed and some before, yielding inconsistent results

Serialisability:
Transactions ideally are “serializable”

- Multiple, concurrent transactions appear as IF they were executed one after another (shopping checkout: make everyone line up)
- Ensures that the concurrent execution of several transactions yields consistent results



But TRUE SERIAL EXECUTION (no concurrency) is very expensive

Concurrency control methods:
To achieve efficient execution of transactions, the DBMS creates a schedule of read and write operations for concurrent transactions

- Interleaves the execution of operations, based on concurrency control algorithms such as locking (main method) and time stamping

Locking:

- Guarantees exclusive use of a data item to a current transaction
    - T1 acquires a lock prior to data access; lock released when transaction is complete
    - T2 does not have access to data item currently being used by T1. Need to wait till T1 is finished
- Required to prevent another transaction from reading inconsistent data

Lock Granularity: OPTIONS:

- **Database-level lock**
    - Entire database is locked
    - Good for batch processing but unsuitable for multi-user DBMSs
- **Table-level lock**
    - Entire table is locked – as above but not quite as bad
    - T1 and T2 can access same database but different tables
    - Can cause bottlenecks, even if transactions want to access different parts of the table and would not interfere with each other
    - Not suitable for highly multi-user DBMSs
- **Page-level lock**
    - Entire disk page is locked (table can span several pages and each page can contain several rows of one or more tables (not commonly used)
- **Row-level lock**
    - Allows concurrent transactions to access different rows of the same table (even if on the same page)
    - Improves data availability but with high overhead (each row has a lock that must be read and written to)
    - MOST POPULAR
- **Field-level lock**
    - Access same row, as long as they access different attributes within that row (not commonly used)
- **Binary lock**
    - 2 stakes: Locked (1) or unlocked (0), lock not released until statement completed

    - Too restrictive to yield optimal concurrency as it locks even for two READs (no updates done)
- **Exclusive Lock**
    - Access reserved for the transaction that locked the object. Must be used when transaction intends to WRITE
    - Granted if and only if no other locks are held on the data item
- **Shared lock**
    - Other transaction also granted READ access
    - Issued when a transaction wants to READ data, and no exclusive lock held

Deadlock
Condition that occurs when two transactions wait for each other to unlock data
- **T1** has item X and wants to update data Y WHILE **T2** has item Y and wants to update data X. Both hangs and wait for each other
- Only happens with **exclusive locks**

Dealt with by:
- Prevention, detection

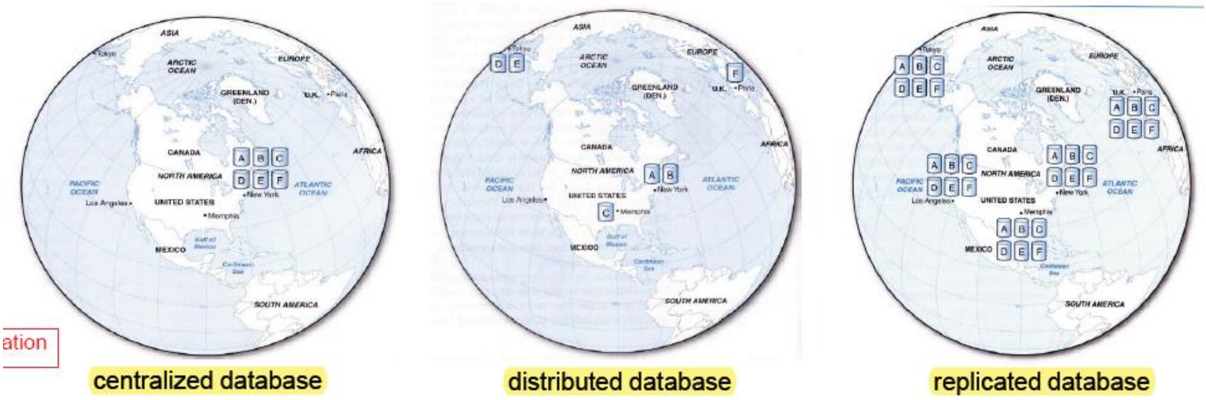
Alternative control methods:
Timestamp:
- Assigns a global unique timestamp to each transaction
- Each data item accessed by the transaction gets the timestamp
- Every data item, DBMS knows which transaction performed the last read/write on it
- When a transaction wants to read/write, the DBMS compares its timestamp with the timestamps already attached to the item and decides whether to allow access
- INCREMENTAL: Therefore, never get the same time. If used **actual** timestamp, people will still clash

Optimistic:
- Based on the assumption that the majority of database operations do not conflict
- Transaction is executed without restrictions or checking

Logging transactions:
DBMS tracks al updates to data and logs it in case of roll backs. Log contains:
- Record for the beginning of the transaction
- Each SQL statement
    - Operation being performed (update, delete, insert)
    - Objects affected by transaction
    - “before” and “after” values for updated fields
    - Pointers to previous and next transactions log entries
- End (COMMIT) of the transaction

Also, provides ability to restore a corrupted database
- If system failure occurs, DBMS will examine the log for all uncommitted or incomplete transactions and it will restore the database to a previous state

Week 8 – Distributed Databases



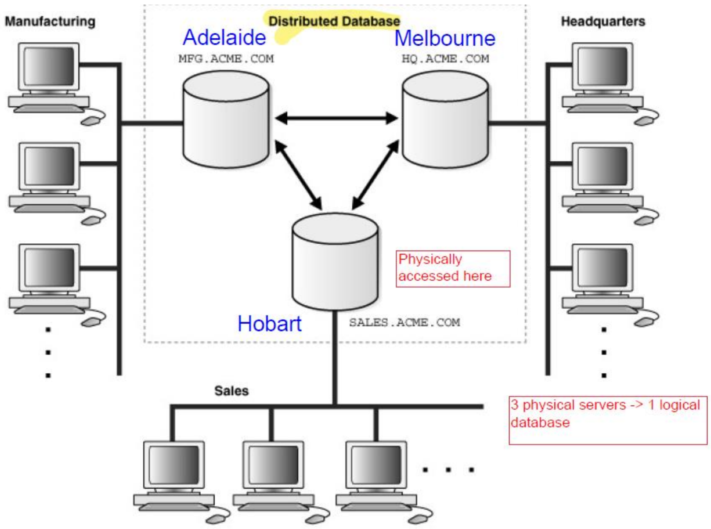
Distributed Database: Single logical database physically spread across multiple computers in multiple locations that are connected by a data communication link

- Appears to users as though it is one database

Decentralized database: Collective of independent databases which are not networked together as one logical database

- Appears to users as though many databases

We consider **distributed database:**



Advantage:

- Good fit for geographically distributed organisations / users
- Data located near site with greatest demand
- Faster data access (to local data)
- Faster data processing
- Allows modular growth
- Increased reliability and availability
    a. Less danger of single-point failure
- Supports database recovery
    a. Data replicated across multiple sites

Disadvantage:

- Complexity of management and control
    a. Database or application must stitch together data across sites
- Data integrity
    a. Additional exposure to improper updating
- Security
    a. Many server sites -> higher chance of breach
- Lack of standards

a. Different DBMS vendors use different protocols
- Increasing training costs
    a. More complex IT infrastructure
- Increased storage requirements
    a. Multiple copies of data

Objectives and trade-offs
Location transparency
- User needn't know where particular data are stored
- Requests to retrieve or update data from any site are automatically forwarded by the system to the site or sties related to the request
- All data appears as a single logical database stored at one site to the user

Local autonomy
- Node can continue to function for local users if connectivity to the networks is lost
- Being able to operate locally when connections to other databases fail

TRADE OFF
- Availability vs consistency
- Synchronous (done immediately) vs asynchronous updates

Distribution options:
Data replication – Data copied across sites
Horizontal partitioning – table rows distributed across sites (e.g. partitioning based on states, data in VIC stored in VIC)

AcctNumber	CustomerName	BranchName	Balance
200	Jones	Lakeview	1000
426	Dorman	Lakeview	796
683	McIntyre	Lakeview	1500
252	Elmore	Lakeview	330

AcctNumber	CustomerName	BranchName	Balance
324	Smith	Valley	250
153	Gray	Valley	38
500	Green	Valley	168

- Different rows of a table at different sites (partition by an attribute, department number in a department table)
- Advantages:
    a. Efficient – data stored close to where it is used
    b. Better performance – Local access optimisation
    c. Security – relevant data stored locally
    d. Ease of query – unions across partitions
- Disadvantages:
    a. Inconsistent access speed
    b. Backup vulnerability – no data replication

Vertical partitioning – table columns distributed across sites
- Different columns of a table at different sites (e.g. Netflix: multiple movies but sites somewhere else)
- Advantages and disadvantages are same as for horizontal partitioning EXCEPT:
    a. Combining data across partitions is more difficult because it requires joins (instead of unions)

Combinations of above

<u>Replication</u>
Advantages:
- High reliability due to redundant copies
- Fast access to local data
- May avoid complicated distributed integrity routines

    a. If replicated data is refreshed at scheduled intervals
    b. 2 servers dealing with real time users, 1 server looking at reports (some need a lot of processing time and will not affect real time users)
- Decouples nodes
    a. Transactions proceed even if some nodes are down
- Reduced network traffic at prime time
    a. If updates can be delayed

Disadvantage:

- Need more storage space
- Integrity: can retrieve incorrect data if updates have not arrived
- Takes time for update operations
    a. High tolerance for out-of-date data may be required
    b. Updates may cause performance problems for busy nodes
- Network communication capabilities
    a. Update place heavy demand on telecommunications
    b. E.g. before it says it has committed to a change, update to all database. However, this may create a lag

BETTER for non-volatile (read-only) data!

Synchronous updates (done immediately)

- Data continuously kept up to date
    a. Users anywhere can get the same answer
- If any copy of a data item is updated anywhere in the network, same update is immediately applied to all other copies or aborted
- Ensures data integrity and minimises the complexity of knowing where the most recent copy of data is located
- Can result in slow response time and high network usage
    a. Spends considerable time checking that an update is accurately and completely propagated across the network

Asynchronous update

- Some delay in propagating data updates to remote databases
    a. Some degree of at least temporary inconsistency is tolerated
    b. May be okay if it is temporary and well managed
- Tends to have acceptable response time
    a. Updates happen locally and data replicas are synchronised in batched and predetermined intervals
- May be more complex to plan and design
- Suits some information systems more than others (e.g. social media where real time updates are not as important

Comparing 5 configuration:

- Centralized database, distributed access
  - DB is at one location, and accessed from everywhere
- Replication with periodic snapshot update
  - Many locations, each data copy updated periodically
- Replication with near real-time synchronization of updates
  - Many locations, each data copy updated in near real time
- Partitioned, integrated, one logical database
  - Data partitioned across at many sites, within a logical database, and a single DBMS
- Partitioned, independent, nonintegrated segments
  - Data partitioned across many sites.
  - Independent, non-integrated segments
  - Multiple DBMS, multiple computers

Comparison: Centralised (place all in one location)

- POOR reliability: Highly dependent on central server
- POOR expandability: Limitations are barriers to performance (hard to add things)
- VERY HIGH Communications overhead: Traffic from all locations goes to one site
- VERY GOOD manageability
- EXCELLENT Data consistency

Comparison: Replicated with Snapshots

- GOOD reliability: redundancy and tolerated delays
- VERY GOOD Expandability: cheap to add new servers
  a. Other side sleeping so don't need to be fast
- LOW TO MEDIUM Communications overhead: Not constant, but periodic snapshots can cause bursts of network traffic
- VERY GOOD Manageability: Each copy is like every other one
- MEDIUM Data consistency: Fine as long as update delays are tolerable

Comparison: Synchronised Replication

- EXCELLENT Reliability: Minimum delays (everyone see what they get)
- VERY GOOD Expandability: Cost of additional copies may be low and synchronisation work only linearly
- MEDIUM Communications overhead: Messages are constant but some delays are tolerated
- MEDIUM Manageability: collisions add some complexity to manageability
- VERY GOOD Data consistency: Close to precise consistency

Comparison: Integrated Partitions:

- GOOD Reliability: Effective use of partitioning and redundancy
  a. Partitions by the right key (not by surname as not a good distribution). Partition by DATE: e.g. looking back in your 1st facebook post sits in the slower partition thus takes longer
- VERY GOOD Expandability: new nodes get only data they need without changes to overall database design
- LOW TO MEDIUM Communications overhead: Most queries are local but queries that require data from multiple sites can cause a temporary load

- DIFFICULT Manageability: Especially difficult for queries that need data from distributed tables, and updates must be tightly coordinated
- VERY POOR Data consistency: Considerable effort; and inconsistencies not tolerated

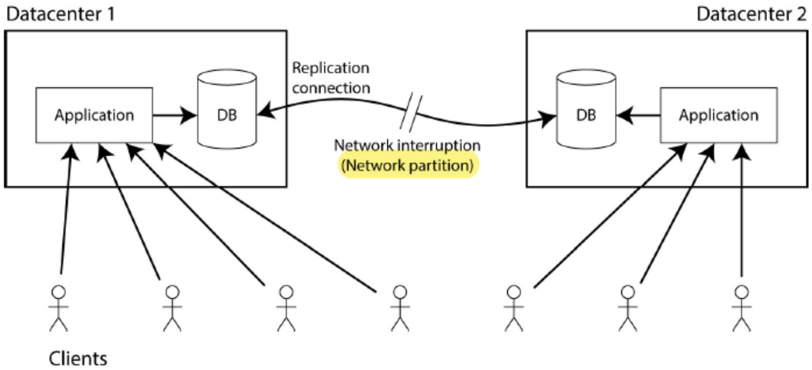
Comparison: Decentralised, Independent Partitions

- GOOD reliability: depends on only local database availability
- GOOD Expandability: new sites independent of existing ones
- LOW Communication overhead: Little if any need to pass data or queries across the network (if one exists)
- VERY GOOD Manageability: Easy for each site, until there is a need to share data across sites
- LOW Data consistency: No guarantees of consistency; in fact, sure of inconsistency

Network Partitions:

Imagine you have synchronously-updating, replicated database – common setup in industry today

- 2 nodes are interrupted (connection between the 2)



Choices:

- Shut down the system (avoid inconsistency) -> WOULD CHOOSE THIS: Could affect lives
- Keep it available to users (and accept inconsistency)

Data's 12 Commandments for Distributed Database

1. **Local site independence**: Each local site acts as an independent, autonomous, centralised DBMS. Each site responsible for security, concurrency, control, backup and recovery
2. **Central Site independence**: No site in the network relies on other sides
3. **Failure Independence**: system not affected by node failures
4. **Location Transparency**: User does not need to know location of data
5. **Fragmentation Transparency**: Data fragmentation is transparent to users, who sees only 1 logical database.
6. **Replication Transparency**: user only sees 1 logical database.
7. **Distributed Query Processing**: A distributed query may be executed at several different sites.
8. **Distributed Transaction Processing**: Transaction may update data at several different sites, and the transaction is executed transparently
9. **Hardware independence**
10. **Operating system independence**
11. **Network independence**
12. **Database independence:** Each database has their own system of how they organise

<u>WEEK 9 – Database Administration</u>

4 functions:

- **Capacity planning:** Estimate disk space and transaction load (not just physical space for database to grow but ability to handle transactions (concurrency)
- **Performance improvement:** Concepts, common approaches (keep things running)
- **Security:** Treats, web apps and SQL injection
- **Backup and recovery**

Can have many DBA located in different counties so no night shifts

Data Administrator:

- Data policies, procedures and standards
- Planning
- Data conflict resolution
- Managing info repository (data dictionary)
- Internal marketing

Database Administrator (technical role)

- Analyse and design DB
- Select DBMS / tools / vendor
- Install and upgrade DBMS
- Tune DBMS performance
- Manage security, privacy, integrity
- Backup and recover

<u>CAPACTIY PLANNING</u>

Predict when future load levels will saturate the system and determining the most cost-effective way of delaying system saturation as much as possible

- Disk space requirements
- Transaction throughput

DURING: System design + system maintenance and review

Estimating disk space requirements

- Vendors sell capacity planning solutions
- Most have same ideas at their core

Treat database size as the sum of all *table sizes*

- Table size = number of rows * average row width (columns)

Need to know storage size of different data types

Need to estimate growth of tables

- Gather estimates during system analysis
- Also need space for other files (control files, data dictionary, indexes, undo areas, redo logs etc.)

Estimating transaction load:

- Consider each application function or transaction (IF reporting application: sorting rows; Ecommerce: writing to disk every time there is a commit (buy))
- E.g. delete most expensive. Not all SQL statements have equal costs (e.g. joins)
- Important to differentiate peak vs average loads

Speed and availability requirements:

Some databases have to handle hundred or even thousands of transactions per second

- Acceptable response time? Reduce response time?

<u>PERFORMANCE IMPROVEMENT:</u>

What affects it?

- Caching data in memory (data buffers)
- Placement of datafiles across disc drives
- Database replication and server clustering
- Fast storage such as SSD
- Use of indexes to speed up searches and joins
- Good choice of data types (especially PKs) e.g. number of Pks
- Good program logic
- Good query execution plans

When to create indexes:

- Choose columns you will index (don't want too many indexes)
  - If used frequently (in Where clauses)
- Columns used for joins
- PK and FK (automatically in most DBMS)
- Unique columns
- Large tables only!!! (small tables do not need indexes)

<u>SECURITY:</u>

Threats:

- Loss of integrity
  - Keep data consistent
- Loss of availability
  - Must be available to authorised users for authorised purposes
- Loss of confidentiality
  - Must protect against unauthorised access

To project database against these types of threats, different countermeasures

Access control

- Provisions for restricting access to data
- Handled by DBA creating user accounts for those with a legitimate need to access the DB
- Keeps track of all operations on the database for all users (usage log)
- Perform audit if tampering suspected
- Based on granting and revoking privileges
- ACCOUNT LEVEL: Privileges that each user hold i.e. operations they can do
- TABLE LEVE: DBA controls a user's privilege to access tables or views

USING VIEWS:

- Can hide the database structure and some data (i.e. hid some columns in the table)
- E.g. if owner A of a table T wants another user B to be able to retrieve only some columns of T, A can create a view V of T that includes only those columns and then grant SELECT on V to B

Encryption

Particular tables or columns may be encrypted to:

- Project sensitive data (password) when they are transmitted over a network
- Encrypt data in the database (e.g. credit card numbers)

Data is encoded using an algorithm (authorized users are given keys to decipher data)

Example of security threat:

SQL Injection:

- Technique used to exploit web applications that use user inputs within database queries
- Malicious code is entered into a data entry field in such a way that it becomes part of SQL commands that are run against the database

How to prevent:

- Pass inputs as parameters to a stored procedure, rather than directly building the SQL string in the code

BACKUP AND RECOVERY:

Backup is an extra copy of your data as your data could be corrupted or deleted. Project against:

- Human error
- Hardware or software malfunction
- Malicious activity
- Natural or man-made disaster
- Government regulation

TYPES:

Physical vs Logical

- Physical
    - Raw copies of files and directories (faster than logical backup)
    - Large important databases that need fast recovery
    - Database is offline when backup occurs
    - Backup = exact copies of the database directories and files
- Logical
    - Backup completed through SQL queries
    - Output larger than physical (doesn't include log or config files)
    - Server available during backup

Online vs Offline

- ONLINE backup
    - Backup when database is "live"
- OFFLINE backup
    - When database is stopped (maximise availability to users, take backup from replication server not live server)
    - Simpler to perform and is PREFERABLE

Full vs Incremental

Backup Policy:

Backup strategy is usually a combination of full and incremental

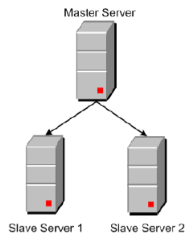
- Need to ensure you can restore from backup

OFFSITE BACKUPS

Other ways to reduce risk:

Replication

- One writer and many readers to help protect against server failure
- Multiple copies of data
- HOWEVER, replicates accidental data deletion



Clusters:

- Usually Linux / Unix only
- Automatically synchronous partition
- Multiple copies of data

Problem	Protection?
accidental drop or delete	data loss!
server failure	protected
security compromise	limited protection

RAID:

- Software or hardware RAID
- Some help protect against drive failure

Problem	Protection?
accidental drop or delete	data loss!
server failure	data may be lost!
security compromise	all data compromised!

WEEK 11 – NoSQL

Relational Model

Advantages:

- Flexible, suits any data model
- Can integrate multiple applications via shared data store
- Standard within and between organisations
- Standard interface language SQL
- Fast, reliable, concurrent and consistent

Disadvantages:

- OR impedance mismatch
- Not good with big data (too much overhead in transactions
- Not good with clustered/replicated servers

Adopted **NoSQL** because of cons of relational

Most business data are tabular

- However, some data is stored across many tables
- A lot of work to dissemble and reassemble the aggregate

NoSQL Database:

Features:

- Doesn't use relational model or SQL language
- Designed to run on distributed servers
- Most are open-source
- Built for modern web
- Schema-less (but might have an implicit schema)
    - If say all students have id and name, then suddenly input 1 student with fav colours, it can store that extra column but there is a limited value)
- 'Eventually consistent' -> NoSQL is not consistent with their data (e.g. if one person makes a change, do not guarantee everyone else can view. NOT designed for banking, more social media)

Relational DB: Has a schema that is very strict what you can input (e.g. id and name)
NoSQL: store anything you like that may not follow the column structure (especially start up where you don't need to redesign)

Goals:

- Improve programmer productivity (OR mismatch)
- Handle larger data volumes and throughput (big data)

Types: RAVEN DB, MongoDB

TYPES of NoSQL:

Key-Value store

KEY = Primary key
VALUE = **anything** (number, array, JSON) – the application is in charge of interpreting what it means

Document databases:

Like a key-value db except that the document is "examinable" by the db, so its content can be queried and parts of it updated

Column Families:

"Column family" is something like a relational tables. It contains many "rows" but each row can store a *different set of columns*

- Columns rather than rows are stored together on disk. Makes analysis by column faster
- Aggregate analysis by columns then by rows (not expecting you to analyse the whole row)

Aggregate-oriented databases

Key-value, document store and column-family are "aggregate-oriented" database
PROS:

- Entire aggregate of data is stored together
- Efficient storage on clusters / distributed databases

CONS:

- Hard to analyse across subfields of aggregates
- E.g. sum over products instead of orders

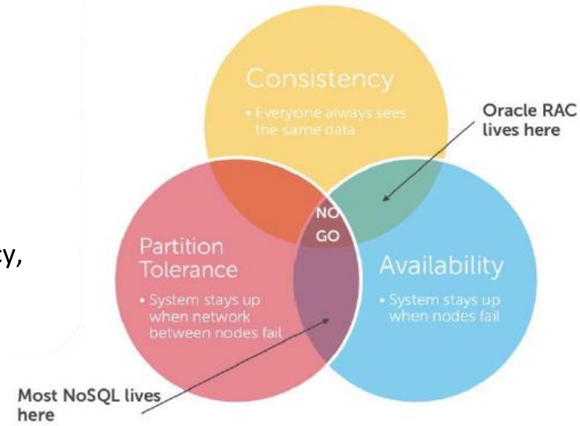
Graph databases:

A 'graph' is a node-and-arc network

- Social graphs (e.g. friendship graphs) are common examples
- Graphs are difficult to program in relational DB
- A graph DB stores entities and their relationships
- Graph queries deduce knowledge from the graph

Distributed data: the CAP theory

CAP theorem says something has to give: Consistency, Partition Tolerance, Availability (CAP)



IF you have a distributed database…

- When a PARTITION occurs, must choose CONSSITENCY or AVAILABILITY

e.g. Melb and NewYork Server:

- If we have partition and cannot update server anymore (server cannot talk)
    - Shut server down
    - Let server keep running but run the risk of doubling up
- Depends on the business => Whether they rather consistency or availability

ACID vs BASE:

ACID (Atomic, Consistent, Isolated, Durable) => SQL
Vs
BASE (Basically Available, Soft state, Eventual consistency) => NoSQL

- Basically, Available: This constraints states that the system does guarantee the availability of the data; there will be a response to any request. BUT data may be in an inconsistent or changing state
- Soft state: State of the system could change over time – even during times without input there may be changes going on due to 'eventually consistency'
- Eventual consistency: System will eventually become consistent once it stops receiving inputs.