Web22 - Database Mandatory #2

Peter Frankild

The ACID principles

Atomicity

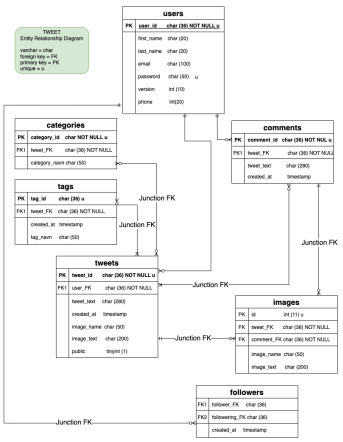
Transactions are seen when a database consists of several tables that interact with each other. This is seen in social media where a post can have multiple comment, if the tables do not have a transaction setup in MyAdminDB there will not be the proper atomicity in the database, you should not be able to delete a post without the comments for that unique post being deleted with it. This is where transaction provides security by either declaring that the operation was successful as an unique unit event or if there is an error in the two operations the event will be stopped. Another example is seen when moving money from one account to another, there are two operations to be performed, an atomic transaction that ensures the database is always consistent.

Consistency
Consistency ensure that the database only can move from one consistance state to another. This is done by ensuring that all data that is inserted to the Database is following all the defined rules, this could be constraings, cascades, triggers. This prevent corrupted data and ensure that the relationsship between forigin key and primary key is in place.

Isolation
Isolation is used when a transaction is executed on multiple tables at the same time, so that the database is kept in the same state as if the transaction were executed sequentially. It ensures that the execution is hidden until it has taken place.

Durability
Durability ensures that the transaction is preserved as a completed transaction even if the system is deleted or fails for some reason. Durability is done by making a non-volatile record of the transaction.

users						
varchar (36)	varchar (20)	varchar (20)	varchar (100)	varchar (255) un	int	varchar (20)
id PK unique	name	last_name	email	password	version	phone
2c85e7fa-55c2-11ed-bdc3-0242ac120002	A	AA	A@	passA	1	phone1
3539a5a8-55c2-11ed-bdc3-0242ac120002	B	BB	B@	passB	2	
tweets						
varchar (36)	varchar (36)	varchar (280)	timestamp	varchar (50)	varchar (200)	tinyint
id PK unique	user_fk	tweet_text	created_at	image_name	image_text	public
3444a5a8-55c2-11ed-bdc3-0242ac120003	2c85e7fa-55c2-11ed-bdc3-0242ac120002	hi one	1866852739			
35555a8-55c2-11ed-bdc3-0242ac120004	2c85e7fa-55c2-11ed-bdc3-0242ac120002	hi two	1866852739			
comments						
varchar (36)	varchar (36)	varchar (36)	varchar (280)	timestamp		
id PK	user_fk	tweet_fk	comment_text	created_at		
stats						
varchar (36)	varchar (36)	varchar (36)	varchar (50)			
stats_id	tweet_fk	user_fk	stats_name			
tags						
varchar (36)	varchar (36)	varchar (50)				
tags_id	tweet_fk	tags_name				
categories						
varchar (36)	varchar (36)	varchar (50)				
category_id	tweet_fk	category_name				
followers						
varchar (36)	varchar (36)	timestamp				
follower_id	followering_id	created_at				
images						
varchar (36)	varchar (36)	varchar (200)				
id	image_name	image_text				



Document database

A document database is a non-relational database eg. MongoDB, ArangoDB, Firebase, Firestore, SurrealDB. The document database has this structure, Database name, one or more collections, the table in an relational database. The Collections is a collection of documents. Documents contain data like an object in a relational db. Then there are several query operators that can be used in schema validating, compare and reference data.

With aggregation you can group sort, calc an analyze data, and it is possible to use more than one stage in the query, each stage run upon the results from a previous stage.
This is how we use aggregate:

```
db.posts.aggregate([
  // Stage 1: Only find documents that have more than 1 like
  {
    $match: { likes: { $gt: 1 } }//$gt(greater than)
  },
  // Stage 2: Group documents by category and sum each categories likes
  {
    $group: { _id: "$category", totalLikes: { $sum: "$likes" } }
  }
])
```

There are name restrictions when creating document databases like there can't be empty spaces and the capitalization has to be consistent throughout the use, and you can not rely on the capitalisation alone, ex Datatype and dataType won't work as unique db. There are also special characters that won't work like "$*<>:|? on a MongoDB windows installation and "$ on Unix, Linux. There are restrictions for the field name _id that always is a unique primary key in the collection.

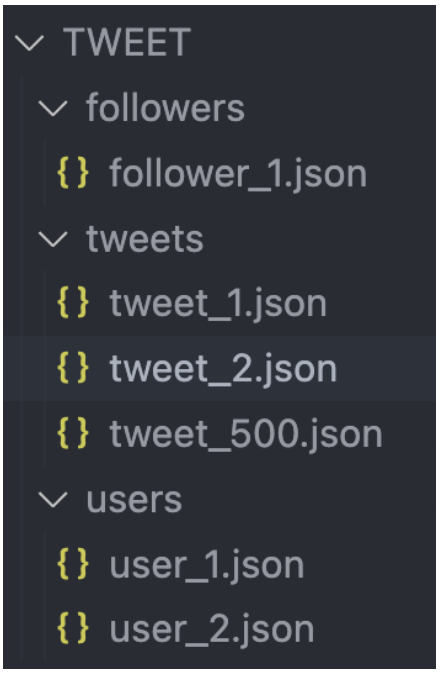
The validation rules are created using a schema that allows e.g. data types and value ranges, this will only work on newly inserted documents. If the validation fails the document will not be written to the collection, but it is possible to allow invalid documents and to have a log warning made.

```
// Using schema validation rules $jsonSchema, this is how it will look like
db.createCollection("xxx", {
validator: {
$jsonSchema: {
bsonType: "object",
title: "Student Object Validation",
required: [ "address", "major", "name", "year" ],
properties: {
name: {
bsonType: "string",
description: "'name' must be a string and is required"
},
year: {
bsonType: "int",
minimum: 2017,
maximum: 3017,
```

```
// And it's rules can be changed using
db.runCommand( {collMod:"xxx",...})
```

Modeling a document database

This is a model from the relational database TWEET in the Database - Mandatory #1 assignment.



4-htdocs > TWEET > users > {} user_1.json > ...

```
{
  "id":"user_1",
  "version":1,
  "name":"A",
  "last_name":"AA",
  "email":"A@",
  "password":"passA",
  "created_at": 431214140,
  "phone":"phone1",
  "tweets":[
    {
      "id":"tweet_1",
      "tweet_text":"hi one",
      "created_at": 431214141,
      "comments":[
        {
          "id":"comment_1",
          "user_id":"user_1",
          "tweet_id":"tweet_1",
          "comment_text":"Hallo world!",
          "created_at": 431214143
        },
        {
          "id":"comment_2",
          "user_id":"user_2",
          "tweet_id":"tweet_1",
          "comment_text":"Hallo world!",
          "created_at": 431214147
        }
      ]
    },
    {
      "id":"tweet_2",
      "tweet_text":"hi two",
      "created_at": 431214142,
      "comments":"null"
    }
  ],
  "comments":"null",
  "follower": "null",
  "followed":[
    {
      "id":"follower_1",
      "created_at": 431214144,
      "follower_id": "user_2"
    }
  ],
  "followers_count": 1,
  "profile_image": "null"
}
```

4-htdocs > TWEET > users > {} user_2.json > ...

```
{
  "id":"user_2",
  "version":1,
  "name":"B",
  "last_name":"BB",
  "email":"B@",
  "password":"passB",
  "created_at": 431214145,
  "phones":"null",
  "tweets":"null",
  "comments":[
    {
      "id":"comment_2",
      "tweet_id":"tweet_1",
      "comment_text":"Hallo world!",
      "created_at": 431214147
    }
  ],
  "follower":[
    {
      "id":"follower_1",
      "created_at": 431214148,
      "followed_id": "user_1"
    }
  ],
  "followed":"null",
  "followers_count": "null",
  "profile_image": "null"
}
```

4-htdocs > TWEET > followers > {} follower_1.json > ...

```
{
  "id": "follower_1",
  "version": 1,
  "created_at": 431214153,
  "follower_id": "user_2",
  "followed_id": "user_1"
}
```

Primary keys" and "foreign keys

Non-relational databases do not support primary and foreign keys. MongoDB stores data records as documents in a collection.

There are different terms used when describing the relations in a document db. As we know from the relations db we can refer to the relations as primary and foreign keys, in a non-relational database the _id is the default primary key. The entity attributes are defined data types such as string, boolean, arrays.
Entity relationships can be One to One, One to Many or Many-Many relationships depending on the relationship between the documents.

One to Many (1:n) occurs when one row of table A may be referenced to any number of rows in table B and where Table B only can be referenced to one row in table A.

Many-Many (n:m) occurs when each reference to many rows in table A can refer to many rows in table B. The n:m relation between tables is done by making a referring table that contains a row with an array of the two foreign keys that have the many-to-many relation.

For example, for long text descriptions, you can create an additional text description collection where the name field is given a unique id for fast processing and to avoid duplicates. An additional query then retrieves the text description. If you want to delete or rename this id you have to edit each row in the referring collection.

If you have a relationship between two documents in a non-relational db you can either use embedding or make a reference to the related document with a _id (primary key).

In a one to one you will make the relation between the two related entities embedding unique id’s in the same document.

If you have a one to many relationship you have to decide if you use a referencing or embedding If the number of relations is low the data needs to be accessed together as an embedded entity in a single document. Is the data infrequent or varying with time referencing is preferred.
In a many to many relationship, can either be a child reference or the referenced is a static entity. Parent References are used when the entity is growing. New child documents add a reference to the parent document's primary key.

The JSON structure of the documents allows defining relationships using either embedding (nested structure) or creating references of related documents using the primary keys, and is great for fast development, with a simple structure and large amount of data.

$unwind operator separates each value from an array in a json like structure.

If a document contains too much data

I will store the main data in a folder with a coded backend limit, if the limit is reached the backend will delete the oldest data in the new_data_folder and place them in the old_data_folder. in the id the data belongs to, there is an array with id and data (info), and FK (foreign key) from the new_data_folder in the database. From this database the backend will be able to find the fast new_data_folder and can retrieve data from the old_data_folder. This works when the data in the folders has an FK (foreign key) that belongs to the id they are associated with.

In-memory db

If you have a database that has to be able to handle many in a short time responses to an id. The in-memory also known as Key value setup stores the latest id and info in the ram. From the in-memory in the server it is much faster to deploy the promise of data to the client end.

Graph database

Neo4j, ArangoDB, SurrealDB on a localhost installation, you run the server and client from two different terminals.

Edge describes the relation between all the connected vertex. The Vertex also known as Node is a enterti in the database like persons, items, emails eg. the attribute is a piece of information which determines the properties of a field or tag in a database or a string of characters in a display.
In every database there is attributes that determints the property of a field like datatype, index and constraints.

```
Edge      Verb       Edge
Node      Relation   Node
Noun                 Noun
RELATE user:a -> likes -> item:adidas
```

Nodes describe entities, and can have zero or more labels.The relationship is a connection between a source node and a target node, must have a type, and it has only one direction. Nodes and relationships can have properties key, value

CRUD queries in a document database

This is how we use INSERT:

```
db.users.insertOne(        //collection
  {
    name: "Peter",         // field: value{ Document
    age: 20,               // field: value{ Document
    runtime: 121,          // field: value{ Document
    education: "mmd",             // field: value{ Document
    year: 2019                    // field: value{ Document
  }
)

db.users.insertMany([
   {
    name: "Joe",
    age: 25,
    education: "mcm",
    year: 2021
  },
    {
    name: "Lis",
    age: 19,
    education: "ak",
    year: 2020
  }
])
```

This is how we use READ:

```
db.users.find(             //collection
{age: {$gt: 20} },         //query criteria
       {name:1, education:1}      //projection
).limit(2)                 //Cursor modifier

{ _id: ObjectId("637eab6d91e17482769d872f"),
  name: 'Joe',
  education: 'mcm' }

db.users.find({})
```

This is how we use UPDATE:

```
db.users.updateOne(        //collection
 { name: "Joe" },          //update filter
{ $set: { education:"sms" }      //update action
})

{ acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
```

```
  upsertedCount: 0 }
```

This is how we use DELETE:

```
db.users.deleteOne(        //collection
{name:"joer"}              //delete filter
)

{ acknowledged: true, deletedCount: 0 }

db.users.find({})
{ _id: ObjectId("637ea8d591e17482769d872c"),
  name: 'Peter',
  age: 20,
  runtime: 121,
  education: 'mmd',
  year: 2019 }
{ _id: ObjectId("637ea8e891e17482769d872d"),
  name: 'Joe',
  age: 25,
  education: 'sms',
  year: 2021 }
{ _id: ObjectId("637ea8e891e17482769d872e"),
  name: 'Lis',
  age: 19,
  education: 'ak',
  year: 2020 }
db.users.deleteMany({})
{ acknowledged: true, deletedCount: 3 }
```

2 graph database commands

```
//select all the likes in all items that are liked from user a.
SELECT *, ->likes->item.* AS likes FROM user:a

// select who user c some final friended med and at the same time buy item x.
SELECT <-friended.*.in->bought[WHERE out=item:x] AS final FROM user:c
```

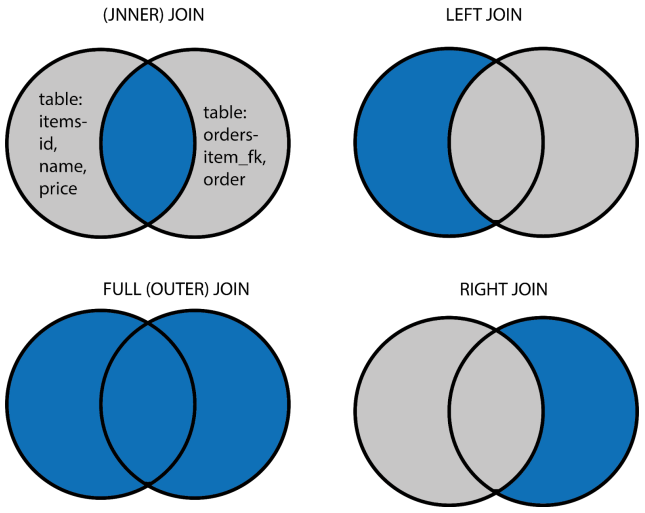
1. In a relational/document database what is Full Text Search? Make sure you apply this concept to your own database. You will be asked to show a practical example of it

Full Text Search

Full-text search is an index that is created and that will remove special characters markeds in letters. It will filter important fillers words out like - and, a or it, this is based on the language. Then the index will link words to keywords if they are in different tenses and whether they are in plural singular. Finally, everything is indexed in either upper or lower case.

Other types of JOIN

In a relational SQL db you can use the JOIN clause that combines rows from two or more tables, if there is one primary and one foreing key that relates the column between them. There are four or more different types e.g. JOIN, FULL OUTER, RIGHT, LEFT.



In the company_one database on phpMyAdmin we can get the records from the table items id and table orders item_fk where primary and foreign key match with an = operator

```
SELECT users.name, items.item_name, items.price
FROM users
JOIN items
ON users.id=items.id
```

name	item_name	price
b	Acics	55.99
d	Nice	81.99
z	Acics	99.99

```
SELECT orders.order, items.name, items.price
FROM orders
JOIN items
ON items.id=orders.item_fk
```

order	name	price
1234	Acics	99.99
1235	Acics	99.99
1237	North	55.99

```
SELECT orders.order, items.name, items.price
FROM orders
LEFT JOIN items
ON items.id=orders.user_fk
```

order	name	price
1234	Acics	99.99
1235	Acics	99.99
1237	North	55.99
1236	*NULL*	*NULL*
1240	*NULL*	*NULL*
1245	*NULL*	*NULL*

```
SELECT orders.order, items.name, items.price
FROM orders
RIGHT JOIN items
ON items.id=orders.user_fk
```

order	name	price
1234	Acics	99.99
1235	Acics	99.99
NULL	Adidas	79.00
NULL	Puma	99.99
NULL	Nice	81.99
1237	North	55.99

then there is outer join that is all just not inner join, right and left join work as a selection without the other.

```
//The FULL JOIN has issues in phpMyAdmin/ MySQL, but will work with
PostgreSQL, and SQL Server
SELECT orders.order, items.name, items.price
FROM orders
FULL JOIN items
ON items.id=orders.user_fk

LEFT / RIGHT JOINT can be unioned with UNION in phpMyAdmin/ MySQL
SELECT orders.order, items.name, items.price
FROM items
LEFT JOIN orders ON items.id = orders.user_fk
UNION
SELECT orders.order, items.name, items.price
FROM orders
RIGHT JOIN items ON orders.user_fk = items.id
```

order	name	price
1234	Acics	99.99
1235	Acics	99.99
12	Acics	99.99
NULL	Adidas	79.00
NULL	Puma	99.99
NULL	Nice	81.99
1237	Acics	55.99

2 triggers created in a relational db

1.

```
increase_total_phones --Trigger name
phones --Table
after --Time
insert --Event
```

```
UPDATE users --Definition
SET total_phones=total_phones + 1
WHERE users.id = NEW.user_fk
root@localhost --Definer
```

2.

```
increase_total_orders --Trigger name
orders --Table
after --Time
insert --Event
UPDATE users --Definition
SET total_orders=total_orders + 1
WHERE users.id = NEW.user_fk
root@localhost --Definer
```

2 stored procedures in a relational db

With Stored procedure you can prepare a SQL query code with a name and save it. It is used when you are writing the same query often, and can be EXEC with parameter values.
To save a Stored procedure In phpmyadmin go to database then routines and make a new one, it can also easily be made by a SQL query like this.CREATE PROCEDURE is the command in the query that starts the procedure.

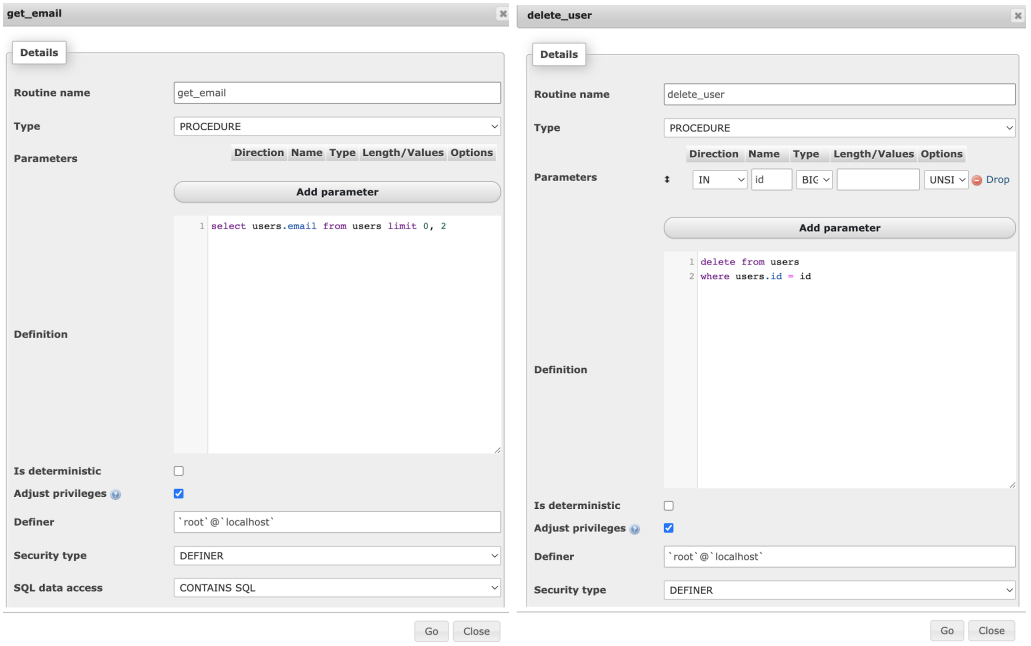
1.

```
-- Select all users
CREATE PROCEDURE SelectAllUsers
AS
SELECT * FROM users LIMIT 1, 2
GO;
-- call the procedure
EXEC SelectAllUsers;

-- get all mails from users
```

2.

```
CREATE PROCEDURE SelectAllWithEmail
AS
SELECT * FROM users
where users.email = email
GO
-- call the procedure
EXEC SelectAllWithEmail;
```



2 views in a relational

In SQL VIEW is a virtual table made from the result of a SQL statement. A view is like a real table but the field is made of one up to several real tables, but presents the data like it were coming from a single table. To create a view you have to start the query with CREATE VIEW

1.
```
CREATE VIEW [users and phones] AS
SELECT users.*, phones.phone
FROM users
JOIN phones  on users.id = phones.user_fk
```

2.
```
CREATE VIEW [average price] AS
SELECT name, price
FROM items
WHERE price < (SELECT AVG(price) FROM items);

SELECT * FROM [average price];
DROP VIEW average price;
```

SQL queries "UNION", "HAVING", "GROUP BY"

Union operator is used when you combine two or more tables. The rule is that it has to have same numbers of columns, of the same data type, and that the columns in the SELECT statement also has the same order

```
SELECT name FROM users
UNION
SELECT name FROM items
ORDER BY name
```

```
--  ALL allows duplicates
SELECT name, id FROM users
WHERE id='1'
UNION ALL
SELECT name, id FROM items
WHERE id='4'
ORDER BY name
```

The HAVING clause is used instead of using an aggregate function

```
SELECT COUNT(id), name
FROM items
GROUP BY name
HAVING COUNT(id) > 0
```

```
SELECT COUNT(id), name
FROM items
GROUP BY name
HAVING COUNT(id) > 0
ORDER BY COUNT(id) DESC
```

The GROUP BY statement make it possible to group rows with the same value into a overview, find in columns and display e.g. COUNT(), MAX(), MIN(), SUM(), AVG() in groups of names

```
SELECT COUNT(id), name
FROM items
GROUP BY name
```

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
ORDER BY COUNT(CustomerID) DESC;
```

Any topic that you find relevant

The SELECT * FROM user statement is also seen in document db as db.user.find({}) and in graph SQL LIKE Surreal DB SELECT * FROM user the same as in SQL, but when SQL uses JOIN to cross refer between tables, document embed or use referencing between documents, Graph use -> <- WHERE and FROM and can perform complex multi-depth fetches.