Data Engineering on AWS (UDACITY)

Data Modelling

ACID Transactions

Properties of database transactions intended to guarantee validity even in the event of errors or power failures.

Atomicity: The whole transaction is processed or nothing is processed. A commonly cited example of an atomic transaction is money transactions between two bank accounts. The transaction of transferring money from one account to the other is made up of two operations. First, you have to withdraw money in one account, and second you have to save the withdrawn money to the second account. An atomic transaction, i.e., when either all operations occur or nothing occurs, keeps the database in a consistent state. This ensures that if either of those two operations (withdrawing money from the 1st account or saving the money to the 2nd account) fail, the money is neither lost nor created. Source Wikipedia for a detailed description of this example.

Consistency: Only transactions that abide by constraints and rules are written into the database, otherwise the database keeps the previous state. The data should be correct across all rows and tables. Check out additional information about consistency on Wikipedia.

Isolation: Transactions are processed independently and securely, order does not matter. A low level of isolation enables many users to access the data simultaneously, however this also increases the possibilities of concurrency effects (e.g., dirty reads or lost updates). On the other hand, a high level of isolation reduces these chances of concurrency effects, but also uses more system resources and transactions blocking each other. Source: Wikipedia

Durability: Completed transactions are saved to database even in cases of system failure. A commonly cited example includes tracking flight seat bookings. So once the flight booking records a confirmed seat booking, the seat remains booked even if a system failure occurs. Source: Wikipedia.

**When Not to Use a Relational Database**

Have large amounts of data: Relational Databases are not distributed databases and because of this they can only scale vertically by adding more storage in the machine itself. You are limited by how much you can scale and how much data you can store on one machine. You cannot add more machines like you can in NoSQL databases.

Need to be able to store different data type formats: Relational databases are not designed to handle unstructured data.

Need high throughput -- fast reads: While ACID transactions bring benefits, they also slow down the process of reading and writing data. If you need very fast reads and writes, using a relational database may not suit your needs.

Need a flexible schema: Flexible schema can allow for columns to be added that do not have to be used by every row, saving disk space.

Need high availability: The fact that relational databases are not distributed (and even when they are, they have a coordinator/worker architecture), they have a single point of failure. When that database goes down, a fail-over to a backup system occurs and takes time.

Need horizontal scalability: Horizontal scalability is the ability to add more machines or nodes to a system to increase performance and space for data.

Normalization

1. First Normal Form (1NF):

For a table to be in the First Normal Form, it should follow the following 4 rules:

It should only have single (atomic) valued attributes/columns.

Values stored in a column should be of the same domain

All the columns in a table should have unique names.

And the order in which data is stored, does not matter.

Example:

Consider a table:

|  |  |
| --- | --- |
| **Customer ID** | Purchased Items |
| 1 | Notebook, Pen, Eraser |
| 2 | Notebook, Pencil |
| 3 | Pen |

This table is not in 1NF because the 'Purchased Items' column has multiple values.

A 1NF version of this table would be:

| **Customer ID** | **Purchased Items** |
| --- | --- |
| 1 | Notebook |
| 1 | Pen |
| 1 | Eraser |
| 2 | Notebook |
| 2 | Pencil |
| 3 | Pen |

2. Second Normal Form (2NF):

A table is in the Second Normal Form if:

It is in the First Normal form.

And, it has no Partial Dependency. (A table has a partial dependency if a column depends not only on the primary key but also on another column of the table.)

Example:

Consider a table:

| **StudentID** | **Course** | **Professor** |
| --- | --- | --- |
| 1 | Math | Prof. A |
| 1 | English | Prof. B |
| 2 | Math | Prof. A |
| 3 | English | Prof. B |

This table is in 1NF but not in 2NF because 'Professor' depends on 'Course', not just 'StudentID' (the primary key).

The 2NF version of this table would be:

StudentCourses Table:

| **StudentID** | **Course** |
| --- | --- |
| 1 | Math |
| 1 | English |
| 2 | Math |
| 3 | English |

Courses Table:

| **Course** | **Professor** |
| --- | --- |
| Math | Prof. A |
| English | Prof. B |

3. Third Normal Form (3NF):

A table is in the Third Normal Form if:

It is in the Second Normal form.

And, it has no Transitive Dependency. (A transitive dependency in a table is a condition where a non-prime attribute depends on another non-prime attribute.)

Example:

Consider a table:

| **StudentID** | **Course** | **Professor** | **Professor Office** |
| --- | --- | --- | --- |
| 1 | Math | Prof. A | Office 101 |
| 1 | English | Prof. B | Office 102 |
| 2 | Math | Prof. A | Office 101 |
| 3 | English | Prof. B | Office 102 |

This table is in 2NF but not in 3NF because 'Professor Office' depends on 'Professor', not directly on 'StudentID' (the primary key).

The 3NF version of this table would be:

StudentCourses Table:

| **StudentID** | **Course** |
| --- | --- |
| 1 | Math |
| 1 | English |
| 2 | Math |
| 3 | English |

Courses Table:

| **Course** | **Professor** |
| --- | --- |
| Math | Prof. A |
| English | Prof. B |

Professors Table:

| **Professor** | **Professor Office** |
| --- | --- |
| Prof. A | Office 101 |
| Prof. B | Office 102 |

Remember that normalization, while helpful in most situations, is not always the best approach. It can lead to increased complexity and poorer performance, especially in read-heavy workloads. Therefore, it is important to consider the specific requirements and context of your database before deciding to normalize.

SQL Statement:

Now let's assume that the customer moved and we need to update the customer's address. However we do not want to add a new customer id. In other words, if there is any conflict on the customer\_id, we do not want that to change.

This would be a good candidate for using the **ON CONFLICT DO NOTHING** clause.

INSERT INTO customer\_address (customer\_id, customer\_street, customer\_city, customer\_state)

VALUES

(

432, '923 Knox Street', 'Albany', 'NY'

)

ON CONFLICT (customer\_id)

DO NOTHING;

Now, let's imagine we want to add more details in the existing address for an existing customer. This would be a good candidate for using the **ON CONFLICT DO UPDATE** clause.

INSERT INTO customer\_address (customer\_id, customer\_street)

VALUES

(

432, '923 Knox Street, Suite 1'

)

ON CONFLICT (customer\_id)

DO UPDATE

SET customer\_street = EXCLUDED.customer\_street;