Building a Transactional Database with Amazon Aurora

Prepared for

Clinton Daniel

University of South Florida - Muma College of Business

by

Safiya Joseph

Debjani Sarma

Srikar Reddy Kamatham

November 15, 2024

a. Description of the DBMS

Amazon Aurora is a cloud-based database management system (DBMS) offered by Amazon Web Services (AWS) and was built to deliver the performance of high-end commercial databases while remaining cost-effective. It is a part of the Amazon Relational Database Service (RDS) which makes it easier to manage a relational database in the cloud.

Amazon Aurora:

- organizes data in tables with columns and rows
- uses primary and foreign keys to maintain relationships between the tables
- supports Structured Query Language (SQL) for data manipulation
- has advanced transactional capabilities making it suitable for transactional applications.

Advantages of Amazon Aurora

- **Security & Integration**: AWS Identity Access Management (IAM) ensures secure access and offers encryption at rest and in transit, and Aurora integrates well with other services like CloudWatch for monitoring, RDS Proxy for connection Pooling and Lambda for serverless computing.
- **Serverless**: serverless configuration works well when there are variable workloads
- **Automated Backup**: Aurora provides automated backups to facilitate recovery
- **High performance & Scalability**: I/O operations are minimized to ensure a high transaction rate and Aurora scales database storage up to 128TB without affecting performance

Disadvantages of Amazon Aurora

- **Higher costs** than traditional MySQL and PostgreSQL
    - especially true when it comes to the advanced capabilities of Amazon Aurora.
- **Vendor Lock-In**
    - This makes it difficult and even costly to migrate to a different cloud provider.
- **Limited Control**
    - AWS IAM for Security can be complexed as it requires careful configuration for network access but can be complex for persons unfamiliar with AWS systems.

Overall, Amazon Aurora is a versatile database management system that is cost-effective and suitable for many applications offering compatibility with MySQL or PostgreSQL. It reduces administrative overhead and allows developers to focus their efforts on transaction functionality. While it is a powerful choice for transaction-oriented database management, it may not be suited for every scenario due to budget constraints and other hidden complexities.

b. Detailed description of the KDD Nuggets referenced data

Online Retail II Dataset

https://www.kaggle.com/datasets/kabilan45/online-retail-ii-dataset/data

This dataset has 8 columns and 1,067,371 rows. The dataset includes columns of data for invoices and products of an online retail store. Selling a wide variety of products. The data spans 2 years of sales data from 2009 – 2011.

Columns:

- **Invoice**: Identifies the invoice number associated with a transaction.
- **StockCode**: Unique identifier for each product in stock.
- **Description**: A description of each product.
- **Quantity**: The number of units of the product in a transaction.
- **InvoiceDate**: The date and time of each transaction.
- **Price**: The price per unit of the product.
- **Customer ID**: Unique identifier for each customer.
- **Country**: The customer's country of origin.

We renamed the columns of the dataset as follows for database design:

Old Name	New Name
Invoice	InvoiceID
StockCode	ProductID
Description	ProductName
Quantity	Quantity
InvoiceDate	InvoiceDate
Price	Price
Customer ID	CustomerID
Country	Country

c. Description of the product and what makes it transactional

The transactional database is designed for interaction between invoices and products to facilitate efficient retrieval of transactional data.

Entities:

- **Product**: A table that stores information about each of the products.
    - ProductID serves as the Primary Key.
- **Invoice**: A table that stores information about each of the invoices.
    - InvoiceID serves as the Primary Key.
- **ProductInvoice**: An associative table to establish a many to many relationship between the Invoice and Product tables. It supports the relationship by storing data on which products are associated with which invoices and in what quantities as well.
    - ProductID and InvoiceID form a composite primary key for this table.
    - ProductID is a foreign key to the Product table and InvoiceID is a foreign key to the Invoice table.

The database is considered transactional because it is structured to handle data with integrity, efficiency and support ACID properties which are needed for transactional operations.

- **Atomicity**: Every transaction either fully completes or fully fails ensuring that there are no partial transactions leading to inconsistencies.
    - Eg. Creating a new invoice either succeeds fully or fails with no partial invoice records.
- **Consistency**: The foreign key constraints maintain consistency throughout the database.
    - Eg. A product must exist in the Product table for it to be referenced in the ProductInvoice table.
- **Isolation**: The database design supports simultaneous or concurrent transactions by isolating them from each other and preventing conflicts.
    - Eg. One user is adding a product while another user is adding an invoice can be done simultaneously there is no interference.
- **Durability**: Once a transaction is completed, it remains in the database in case of system failures.
    - Eg. Finalizing an invoice ensures that the changes are saved, even in case of system failure.

The inclusion of the ProductInvoice table helps to support the transactional nature of the database by providing the flexibility necessary for an ecommerce database. It aids in data consistency and scalability.

d. Description of the data structures

Product Table

Column Name	Constraints	Data Type	Description
ProductID	Primary Key, Not Null	VARCHAR(20)	A unique identifier for each product.
ProductName	Not Null	VARCHAR(255)	The name product.
Price	Not Null	DECIMAL(10,2)	The price of the product.

ProductInvoice Table

Column Name	Constraints	Data Type	Description
ProductID	Primary Key, Foreign Key, Not Null	VARCHAR(10)	A unique identifier for each product.
InvoiceID	Primary Key, Foreign Key, Not Null	VARCHAR(10)	A unique identifier for each invoice.
Quantity	Not Null	INT	The quantity of each item included in a particular invoice.

Invoice Table

Column Name	Constraints	Data Type	Description
InvoiceID	Primary Key, Not Null	VARCHAR(10)	A unique identifier for each invoice.
InvoiceDate	Not Null	TIMESTAMP	The date and time of the invoice was created.
CustomerID	Not Null	VARCHAR(10)	A unique identifier for each customer making the purchase.
Country	Not Null	VARCHAR(100)	The country where the customer is located.

e. Detailed description of the CRUD Operations

A database is transactional if it supports ACID properties. CRUD operations must follow these principles to ensure data integrity and consistency, or in case of failures.

Transactional Check:

The range from BEGIN to COMMIT is referred to as a Transactional Boundary. If a transaction is started using 'BEGIN' and data is inserted into multiple tables, Amazon Aurora ensures that these changes are committed together using 'COMMIT'. 'ROLLBACK' can be used so if any part of the transaction fails, the entire transaction is rolled back.

The **CREATE** operation allows you to add new data into the database.

Here, being transactional means that it is guaranteed that either the related data is inserted or none of it is inserted, in case of failure.

Example: Insert a new Invoice with Products

```
BEGIN;
 -- Insert a new product (if it doesn't already exist)
INSERT INTO Product (ProductID, ProductName, Price)
        VALUES ('23020', 'GLASS SONGBIRD STORAGE JAR', 12.5);
 -- Insert a new invoice
INSERT INTO Invoice (InvoiceID, InvoiceDate, CustomerID, Country)
        VALUES ('577611', '11-24-2011 14:00', '12351', 'Germany');
 -- Insert into ProductInvoice to specify the quantities of products
INSERT INTO ProductInvoice (InvoiceID, ProductID, Quantity)
        VALUES ('577611', '23020', 1);
 -- Commit the transaction
COMMIT;
 -- ROLLBACK in case of failures
ROLLBACK;
```

The **READ** operation allows you to query the database to retrieve information.

Read Operations can be done within a transactional boundary to retrieve data.

Example: Read the invoice details for a specific customer

```
BEGIN;
SELECT i.InvoiceID, i.InvoiceDate, p.ProductName, pi.Quantity, p.Price, i.Country
FROM Invoice AS i
JOIN ProductInvoice as pi ON i.InvoiceID = pi.InvoiceID
JOIN Product as p ON pi.ProductID = p.ProductID
WHERE i.CustomerID = '12349';
-- Commit the transaction
 COMMIT;
-- ROLLBACK in case of failures
ROLLBACK;
```

The **UPDATE** operation modifies existing data in the database.

When multiple tables or related records are updated, the changes must be made as part of a single transaction. This way, if one update fails, all changes can be rolled back to maintain consistency.

Example: Update the Quantity of a Product in an Invoice

```
BEGIN;
 -- Update the quantity of product '22423' in invoice '577609'
UPDATE ProductInvoice
        SET Quantity = 2
        WHERE InvoiceID = '577609' AND ProductID = '22423';
 -- Commit the transaction
COMMIT;
-- ROLLBACK in case of failures
ROLLBACK;
```

The **DELETE** operation is used to remove records from the database.

When deleting records, foreign key constraints should be considered to avoid violating referential integrity.

Example:Delete an Entire Invoice

```
BEGIN;
-- Delete all records related to invoice '577609' from ProductInvoice
DELETE FROM ProductInvoice
    WHERE InvoiceID = '577609'
-- Delete the invoice record itself
DELETE FROM Invoice
    WHERE InvoiceID = '577609';
-- Commit the transaction
COMMIT;
-- In case of failure, roll back the changes
ROLLBACK;
```

References

PostgreSQL. (n.d.). Documentation. https://www.postgresql.org/docs/

What is Amazon Aurora? - amazon Aurora. (n.d.). https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/CHAP_AuroraOverview.html