Chapter 4

Database Internals: Transactions

Contents

1 REDO / COMBINE NEXT SECTIONS

2 Table Creation and Deletion

- Up to this point we have glossed over the details of how tables and schemas are created. There are a few reasons for doing this:

  1. In professional settings it is relatively uncommon to be creating tables. By definition they are created only once – as opposed to accessing them which can be done much more frequently.
  2. The considerations for how a table is created are very specific to the database variant. Data types are treated differently within different database.
     - For example, in Postgres, there is no "fixed length string type". Even if you define a column as a CHAR, the database treats it as a VARCHAR. Other SQL variants do not follow this same pattern and therefore the amount of space taken up by each column and its relative efficiencies are different in different variants.
     - These types of physical layer differences can mean that optimizations and best practices that work on one database variant may not work on another.
  3. On top of the the physical layer differences, variants also express different syntax in their creation statements.
     - Just like the rest of SQL, the core syntax displays is similar between variants. However, once you step outside the basics of creating a table, the syntactical differences start increasing.
  4. Last, but not least, questions regarding creating a table rarely appear in data science or data analyst interviews.

- Given the above we won't spend too much time understanding the syntax of creating a table, though we will cover the important aspects involved.
- The basic syntax for creating a table follows the following form:

```
CREATE TABLE schema.table_name
(
        column_1_name data_type,
        column_2_name data_type,
        column_3_name data_type
);
```

  It is a pretty simple syntax overall – name and datatypes are the only two required arguments.

3 Database Operations: CRUD

There are four essential operations for any database which we abbreviate **CRUD**:

- **C**reate: New data can be added to a database[1]
  - CREATE: Create a table for data to be stored.
  - INSERT: Put data into a table.

[1] Relational databases break up the process of data creation into two steps. The first is to create a container for the data (via CREATE) and then second is to populate that table, using either a COPY or INSERT command.

  - COPY: Bulk data loading operation.

- **R**ead: Retrieve previously stored data.
  - SELECT: Returns data to the client.
- **U**pdate: Previously stored data can be changed in a database.
  - UPDATE: Changes already stored data.
  - ALTER: Changes the structure of a table.
- **D**elete: Previously stored data can be removed
  - DROP: Removes a database object
  - DELETE: Removes data from a table
- Many databases, including newer versions of PostgreSQL include a command UPSERT, which is a portmanteau of UPDATE and INSERT. UPSERT will insert a new row, unless that row is already present, in which case it will update.
- This breakdown can be found in a variety of settings, including when used in RESTful APIs, which is the basic architecture used for client server communication via the internet. As can be seen in Table 4.1, each operation above maps directly to a type of HTTP request.

Operation	HTTP Method
Create	PUT
Read	GET
Update	PUT
Delete	DELETE

Table 4.1: CRUD map to HTTP Requests

In this course we aren't as interested in operations outside of SELECT. For most data analysts and data scientists, 99.999% of the queries that they will write are retrieving data.

4 Creating Tables, Constraints and Deleting tables

- To create an empty table we use the `CREATE TABLE` command and specify the table name and the columns therein.
- At the time that the table is created we specify the entire table schema and specifically the column names and the data types of those columns.
- If we were create a table which contains the names of people and the balances of their bank accounts, we would use the following command:

```
CREATE TABLE cls.balances (
        aName varchar(30)
        , aBal int
);
```

  which would create a table with two columns, the first being called aName and the second aBal. The first column is a `varchar` – or variable length character string and the second is an integer.

- This command also allows us to specify additional properties of the table and columns within the table.
- There are a few classes of properties that we can specify:
  1. **Storage information:** Specific configuration regarding how the data is stored at the physical layer (beyond the scope of this course)
  2. **Constraints:** Rules that the data within the table must abide by *before* the data is inserted into the table. We will talk a bit about this below.
  3. **Index information:** What indexing strategies should be used in the table (we'll talk about this later).
  4. **Triggers:** A "trigger" is a piece of procedural (usually non-SQL code) that is run when a particular SQL command is run. For example, you could create a trigger which would send an email or raise an alert if a particular type of SQL command is run on the table. This is beyond the scope of this course.
- Constraints and Triggers are powerful tools within a database because they allow us to verify data as it is being loaded and execute additional commands upon the data being updated or changed. The downside of both of these powerful tools is that they put additional strain at on the database and therefore in certain situations they are avoided. For example, databases with very large write-loads (e.g. they are writing a lot of data very quickly) will usually avoid adding constraints because the cost of checking those constraints can bottleneck the system.
- The two most frequently used constraints are `NOT NULL` and `UNIQUE` which enforce the obvious on a column (or set of columns). For example, we could write the following as our create table:

```
CREATE TABLE cls.balances2 (
        aName varchar(30) UNIQUE NOT NULL
        , aBal int NOT NULL
);
```

  which would prevent the database from allowing any data to be inserted with either a Null abalance or a Null or Non-unique aName.
- Uniqueness constraints can also be created at the table level on groups of columns. It is possible to encounter a situation where we wanted a pair (or more) of columns to be unique together we could also specify this.
- There are tons of other constraints, but they are not as common and their syntax is frequently SQL variant specific. Here is a fun constraint on aName:

```
CREATE TABLE cls.balances3 (
        aName varchar(30) UNIQUE NOT NULL CHECK (trim(aname) <> '' AND length(aname) > 5 )
        , aBal int NOT NULL
);
```

  This would check to make sure that aName is (1) unique, (2) not null, (3) not equal to an empty string and (4) length larger than 5.
- To delete a table from the database we use the command `DROP`. To drop the tables we have created above you could use the following commands:

```
DROP TABLES cls.balances2;
DROP TABLES cls.balances3;
```

- A useful argument for these two commands specifies behavior in the case that the table already or does not exists. The arguments `IF NOT EXISTS` and `IF EXISTS` which are used on `CREATE` and `DROP` respectively and will only create if the table does not exist and drop if it does. These are useful when loading and dropping data to avoid errors being returned. You can find examples of these commands in the sql-data repository and specifically the functions contained in the `load_data.py` file.

5 Altering Tables

- To change the structure of a table we use the command `ALTER`.
- The most common operations that we do with `ALTER` are adding, dropping and modifying columns.
- These operations should not be undertaken lightly as they can be incredibly expensive for the database. Mentally, you should equate running one of these commands with reading and rewriting the entire table.
- Here are two examples:

```
ALTER TABLE cls.balances ADD COLUMN new_col_1 varchar(10);
ALTER TABLE cls.balances DROP COLUMN new_col_1;
```

6 Inserting, Copying, Updating and Deleting

- We use the `INSERT` command to put small amounts of data into a table. For larger amounts we use the `COPY` command.
- While the `INSERT` command facilitates getting data into a table a few different ways, we'll focus on two: providing values directly and loading values generated from a query.
- Consider the following commands, which create a table and then insert values into it from the output of another query.

```
create table cls.unique_county_names (
countyname varchar(20) NOT NULL UNIQUE
);

insert into cls.unique_county_names
        (select distinct countyname from cls.cars);
```

- The other common way to use `INSERT` to put data into a table is by providing values, such as in the example below:

```
CREATE TABLE cls.balances (
        aName varchar(30)
        , aBal int
);

INSERT INTO CLS.balances VALUES ('Nick', 1000), ('Jim', 300)
        , ('Judy', 100), ('Jean', 1500);
```

- Each row is put in parenthesis in this method.
- Inserting data into a table is (unsurprisingly) costly and should not be undertaken lightly.
- If we wish to change data within a table, while maintaining the same data types we use the `UPDATE` command, as in the following example:

```
CREATE TABLE cls.balances (
        aName varchar(30)
        , aBal int
);

INSERT INTO CLS.balances VALUES ('Nick', 1000), ('Jim', 300)
        , ('Judy', 100), ('Jean', 1500);

UPDATE cls.balances set aName = 'Nicholas' where aName = 'Nick';
UPDATE cls.balances set aBal = 0;
```

- To remove data from a table we use `DELETE`:

```
DELETE FROM cls.balances where aName = 'Nicholas';
```

- If we want to drop all rows from a table we use `TRUNCATE`, which is a high performance delete when removing *all* rows.

```
TRUNCATE TABLE cls.balances;
```

7 Transactions and ACID

- Consider the following table and queries:

```
CREATE TABLE cls.balances (
        aName varchar(30)
        , aBal int
);

INSERT INTO CLS.balances VALUES ('Nick', 1000), ('Jim', 300)
        , ('Judy', 100), ('Jean', 1500);

--Assume that Nick makes a deposit of 100:
update cls.balances set aBal = aBal + 100 where aName = 'Nick';

--Assume that Jim withdraws 200:
update cls.balances set aBal = aBal - 200 where aName = 'Jim';

--Assume that Jean drops her account:
delete from cls.balances where aName = 'Jean';

--Assume that Julian creates an account with 250:
insert into cls.balances values ('Julian', 25);
```

The resulting table would have four rows:

```
select * from balances;

 aname  | abal
--------+------
 Judy   |  100
 Nick   | 1100
 Jim    |  100
 Julian |  250
 (4 rows)
```

- A **transaction** is a unit of work which is to be treated as a single entity within a database. A transaction may consist of *multiple* queries.
- Transactions are important in a few specific instances: (1) If there are multiple users/sessions accessing and manipulating the database and (2) If queries are required be executed together.
- If nothing within the transaction is changing the state of the database (think the underlying data), then transactions become much less important.
- If there is only a single user interacting with the database in a single session, then the implications of transactions aren't that important, but this is not usually the case.
- The lifecycle, or transaction process looks like the following:
    1. Begin the transaction
    2. Attempt all statements within the transactions:
    3. If they are successful then *commit* the transaction.
    4. Otherwise (at least one statement fails): then *rollback* the transaction.

5. End the transaction.

- The phrase *commit* is technical – it's when whatever changes to the database are set so that other users see them.
- If a commit fails, the database will *rollback* to its previous state.
- Consider the following example, based on the *balances* table from the previous section, where Nick gives $100 to Jim.

```
BEGIN;

UPDATE CLS.BALANCES SET ABAL = ABAL + 100 where aname = 'Jim';

UPDATE CLS.BALANCES SET ABAL = ABAL - 100 where aname = 'Nick';

COMMIT;
```

- In this example the two commands regarding Nick and Jim are treated as a *single* command. We want this to be the case – if something goes wrong, the state of the database returns to the state that it was at the start of the transaction.
- Most SQL clients have an *AUTOCOMMIT* feature, which runs every query as its own transaction. Some SQL clients will use the semi-colon as a transaction divider. For most of the SQL clients I've seen and used these parameters are configurable.

```
BEGIN;

UPDATE CLS.BALANCES SET ABAL = ABAL + 100 where aname = 'Jim';

--Query below has an error and thus the database will be returned to the
--state before the BEGIN
UPdt CLS.BALANCES SET ABAL = ABAL - 100 where aname = 'Nick';

COMMIT;
```

- Consider the queries in Figure 4.1. Each column represents a connection to the database while time increases downward. Any query not in a transaction block is its own transaction.
- Importantly, the database keeps the transaction behavior separate between the connections until those transactions are committed.
- What makes a good transaction? Transactions in Relational Databases have the following properties, generally referred to as ACID:
  - **A**tomic: Transactions are treated as a single unit. If there are 9 queries within a transaction and the 8th one fails, the seven queries that preceded it are not committed to the database.
  - **C**onsistent: A transaction is processed if and only if it does not violate any system rules. For example, if you try to put a string in an integer field then the transaction will fail.
  - **I**solation: Transactions that are executed concurrently behave *appropriately.* More on this in the next section.
  - **D**urable: Once a transaction is committed, it is committed perpetually. If the system reboots or restarts after a transaction is committed then you can expect that transaction to have gone

Figure 4.1: Transaction Demonstration

Connection #1

```
SELECT * FROM CLS.BALANCES;

aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)

BEGIN;

UPDATE CLS.BALANCES
        SET ABAL = ABAL + 100
        WHERE aname = 'Jim';

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
300
(1 row)

UPDATE CLS.BALANCES
        SET ABAL = ABAL - 100
        WHERE aname = 'Nick';

COMMIT;
```

Connection #2

```
SELECT * FROM CLS.BALANCES;
aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
200
(1 row)

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
300
(1 row)
```

through. After a restart the system will not contain the previous state of the database.

8 Isolation Levels in Relational Databases

- The Isolation aspect of ACID is complex. Specifically, Isolation refers to how the database operates in the case when multiple transactions conflict with each other. For example, if two transactions try to update the same row of data within a table then isolation describes how this conflict gets resolved.
- When I think of Isolation I don't think of it as a "single property" – like Atomicity or Durability, but more as a spectrum of behaviors that a database can demonstrate in relation to specific conflicts.
- The ANSI SQL standard specifics four different levels of transaction isolation. These isolation levels are defined by what phenomena they disallow, as can be seen in Table 4.2. Note that these Isolation levels are listed from "weakest" to "strongest". We will go over each below.[2]

Isolation Level	**Dirty Read**	**Nonrepeatable Read**	**Phantom Read**	**Serialization Anomaly**
Read uncommitted	Allowed[3]	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed[4]	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

Table 4.2: ANSI standard for Isolations

1. **Dirty Read:** A dirty read occurs when a user sees data that is not committed. Consider the following example, once again starting from the transactions in 4.2 (which are similar to those in Figure 4.1 above). A dirty read has occurred because, in connection #2 the balance shows the uncommitted 300 which is a contamination between transactions.

   We can't give a proper demonstration of a dirty read in Postgres because, even at the lowest Isolation Level ("Read Uncommitted") Postgres's Isolation system won't allow it.

2. **Nonrepeatable Read:** A nonrepeatable read occurs when the state of database has changed between two reads within a transaction. Specifically, nonrepeatable reads occur when the *value within a row* has changed between two `SELECT` statements. Consider the set of transactions in Figure 4.3 which demonstrate this occurring as the second connection returns two different values for the same query *within* a transaction.

3. **Phantom Read:** A Phantom Read occurs when the state of database has changed between two reads within a transaction and it *effects which rows are returned*. Consider the example in Figure 4.4. In this example the rows that are being returned are incorrect, in that the second transaction shouldn't see the changes from the first transaction as they have not been committed.

4. **Serialization Anomaly:** A serialization anomaly occurs when the result of concurrent transactions is different depending on the order. For example, consider the transactions in Figure 4.5. In this case, the order of the transactions, which are concurrent, determines the final table values.

[2]There are a number of people who have strong opinions about the effectiveness of these designations. *A Critique of ANSI SQL Isolation Levels* published by Microsoft Research's Advanced Technology division provides a good starting point for this discussion.

Figure 4.2: Dirty Read Example

Connection #1

```
SELECT * FROM CLS.BALANCES;

aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)

BEGIN;

UPDATE CLS.BALANCES
        SET ABAL = ABAL + 100
        WHERE aname = 'Jim';

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
300
(1 row)

UPDATE CLS.BALANCES
        SET ABAL = ABAL - 100
        WHERE aname = 'Nick';

COMMIT;
```

Connection #2

```
SELECT * FROM CLS.BALANCES;
aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
300
(1 row)

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
300
(1 row)
```

Figure 4.3: Nonrepeatable Read Example

Connection #1

```
SELECT * FROM CLS.BALANCES;

aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)

BEGIN;

UPDATE CLS.BALANCES
        SET ABAL = ABAL + 100
        WHERE aname = 'Jim';

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
300
(1 row)

UPDATE CLS.BALANCES
        SET ABAL = ABAL - 100
        WHERE aname = 'Nick';

COMMIT;
```

Connection #2

```
BEGIN;

SELECT * FROM CLS.BALANCES;
aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
200
(1 row)

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
300
(1 row)

COMMIT;
```

Figure 4.4: Phantom Read Example

Connection #1

```
SELECT * FROM CLS.BALANCES;

aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)

BEGIN;

UPDATE CLS.BALANCES
        SET ABAL = 50
        WHERE aname = 'Jim';









COMMIT;
```

Connection #2

```
BEGIN;
SELECT * FROM CLS.BALANCES;
aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)









SELECT ANAME from CLS.BALANCES
WHERE ABAL <= 100;
 aname
--------
 Julian
 Jim
(2 rows)




COMMIT;
```

Figure 4.5: Serialization Anomaly Example

Connection #1

```
SELECT * FROM CLS.BALANCES;

aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)

BEGIN;

UPDATE CLS.BALANCES
        SET ABAL = 50
        WHERE abal >= 100;

COMMIT;
```

Connection #2

```
SELECT * FROM CLS.BALANCES;

aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)

BEGIN;

UPDATE CLS.BALANCES
        SET ABAL = 25
        WHERE abal >= 100;

COMMIT;
```

- Referring back to Table 4.2, there are multiple isolation levels defined which either allow or disallow the above cases to occur. In Postgres, we can see the current isolation level as well as set the isolation level within a transaction using the commands:

```
BEGIN;

sql_class=*# show transaction isolation level;
 transaction_isolation
-----------------------
 read committed
(1 row)

set transaction isolation level Serializable;
SET

show transaction isolation level;
 transaction_isolation
-----------------------
 serializable
(1 row)

COMMIT;
```

- By switching the isolation level associated with the transaction we can change the databases behavior in these situations.
- What happens if we trigger this behavior? In PostgreSQL, the offending transaction will either "lock" or "fail" due to an error. In the case that the isolation level is set to `serializable` and two conflicting transactions occur, the database will return an error like:

```
ERROR:  could not serialize access
        due to read/write dependencies
        among transactions
DETAIL:  Cancelled on identification
         as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

- A "lock" on the other hand occurs when the database realizes that two processes are vying for a single resource and chooses to "block" one of those processes until something gets resolved. In the case of a lock occurring user intervention maybe required if certain time out thresholds aren't met.
- Why does this matter?

9 Why do we care (NoSQL)?

- THIS IS STILL UNDER CONSTRUCTION.
- Why is ACID important to think about? It seems that these properties seem "obvious", in the sense that they are things that we want in a database. While that maybe true, making sure that a database is ACID compliant is costly in a few ways.
- While there is no "official" definition of NoSQL, one way that other database systems distinguish

themselves is by not being fully ACID compliant. Amazon's Cloud Service FAQ notes:[5]

> NoSQL databases often trade some ACID properties of traditional relational database management systems (RDBMS) for a more flexible data model that scales horizontally. These characteristics make NoSQL databases an excellent choice in situations where traditional RDBMS encounter architectural challenges to overcome some combination of performance bottlenecks, scalability, operational complexity, and increasing administration and support costs.

- Is this the *only* thing that distinguishes NoSQL databases from traditional RDMS systems? Absolutely not.
- There are tons of very reasonable ways that a database can relax the rules around an RDMS and be called a NoSQL database – relaxing the transaction assumptions is just one of the ways that it can be done.
- Another common distinction between RDMS and NoSQL databases is how information within the database is stored and it's ability (or inability) to be used internally to the database, such as in joins.
- MongoDB ("Mongo") presents an interesting case study in what it means to be a NoSQL database. When Mongo was originally released, the first versions had known limitations around their durability. A (relatively) at the time, database meme was that MongoDB was "marketing pretending to be a database" or that it was the SnapChat of databases (see the image in 4.6 for something that was passed around in the early 2010's).

Figure 4.6: Something to keep in mind! Source unknown.

[5] https://aws.amazon.com/nosql/?nc1=f_cc

The original versions of MongoDB were well known to not be durable

MongoDB, for example, is not as atomic as other databases. In Mongo for example, a WHERE clause may not return a matching row if the row is in the middle of an update, even if both pre- and post-update that row would match the WHERE clause. Amazon's Redshift system also relaxes some consistency measures to make sure that data can load quickly.

10 NoSQL

Types of databases and NoSQL.

Generally NoSQL relaxes some of the constraints that can be found in relational databases.

- **Key-Value Stores**: Key-value stores are databases that are organized around two columns, a key and a value. Values in this case generally have set, fixed schemas. Examples of this type of database include HBase, Amazon's Dynamo DB and Cassandra. The common use case of this data is severing user-specific data, such as saved game files or account information. In most of these cases there is enough structure on the data in the value the database itself can use that information to it's advantage.

- **Document Stores**: This type of database structure is similar to the Key-Value store, expect that instead of the record being a value, the record is a document, which is different from a value in that it has a less fixed size schema. The reason that this is called a document store, as opposed to a key value store is to make it clear that the database isn't going to try to use information regarding the values in optimizing its data structure. Examples of this include MongoDB and common use cases include storing network information.

- **Relational databases**: Relational databases are the most common type of database. The key feature of these databases is that the information within the database is expected to be stored in multiple tables that join together. In terms of performance, the database engines expect to use joins and provide structure on optimizing tables along different dimensions to facilitate multiple joins. There are two major classes of these databases and the properties of each are sufficiently different that we cover them separately.

  - **Row-based databases**: A Row-based database, such as PostgresSQL and Postgres store data in rows on the hard drive. In other words, if you were looking at a row-column entry on the hard drive and then read the next big of data, it would contain the next column in the database, alone that same row. This means that loading whole rows, once a row has been identified, is performant. If you were working at a bank-terminal and were looking up customer information, this would make sense, since once the customer's location has been identified on the hard drive, all the ancillary information about that customer is right there. The first two databases that we are going to talk about in this class, PostgresSQL and Postgres are both row-based relational databases.

  - **Columnar databases**: Column-based ("Columnar") databases, store information in columns on the hard drive. This type of database technology is relatively recent, examples include Vertica, SAS HANA and Amazon's redshift. These databases exist to assist data analysts, as the most common operations that data analysts do are not row, but column based. For example, in a row-based database, finding the average of a column is difficult because the information will be scattered over the entire data partition. Columnar databases are optimized for this type of operation by placing the data in column order. The final database that we are going to consider in this class, Amazon Redshift, is a columnar database.

11 Transaction Implementations [TBD]