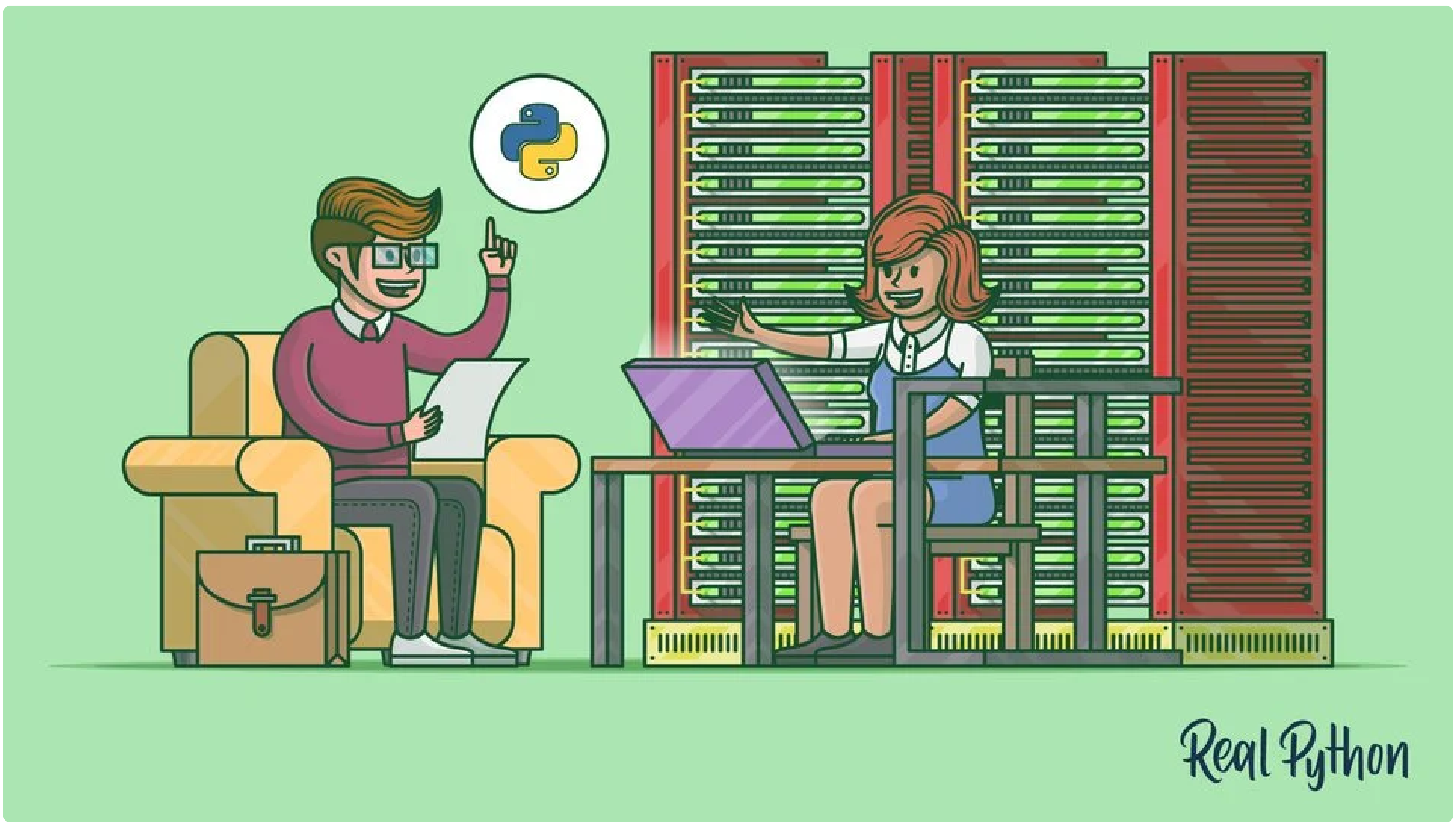


Data Engineer Interview Questions With Python

by Reuben Ferrante
Dec 11, 2019 6 Comments databases devops intermediate

Mark as Completed

Tweet Share Email

Table of Contents

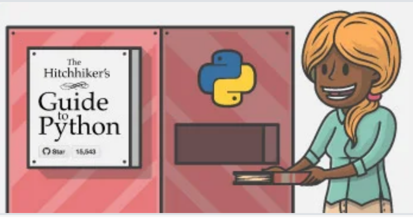
Going to interviews can be a time-consuming and tiring process, and technical interviews can be even more stressful! This tutorial is aimed to prepare you for some common questions you'll encounter during your data engineer interview. You'll learn how to answer questions about databases, Python, and SQL.

By the end of this tutorial, you'll be able to:

- Understand common data engineer interview questions
- Distinguish between relational and non-relational databases
- Set up databases using Python
- Use Python for querying data

Becoming a Data Engineer

The data engineering role can be a vast and varied one. You'll need to have a working knowledge of multiple technologies and concepts. Data engineers are flexible in their thinking. As a result, they can be proficient in multiple topics, such as databases, software development, DevOps, and big data.



What Does a Data Engineer Do?

Given its varied skill set, a data engineering role can span many different job descriptions. A data engineer can be responsible for database design, schema design, and creating multiple database solutions. This work might also involve a Database Administrator.

As a **data engineer**, you might act as a bridge between the database and the data science teams. In that case, you'll be responsible for data cleaning and preparation, as well. If big data is involved, then it's your job to come up with an efficient solution for that data. This work can overlap with the DevOps role.

You'll also need to make efficient data queries for reporting and analysis. You might need to interact with multiple databases or write Stored Procedures. For many solutions like high-traffic websites or services, there may be more than one database present. In these cases, the data engineer is responsible for setting up the databases, maintaining them, and transferring data between them.

How Can Python Help Data Engineers?

Python is known for being the swiss army knife of programming languages. It's especially useful in data science, backend systems, and server-side scripting. That's because Python has strong typing, simple syntax, and an abundance of third-party libraries to use. Pandas, SciPy, Tensorflow, SQLAlchemy, and NumPy are some of the most widely used libraries in production across different industries.

Most importantly, Python decreases development time, which means fewer expenses for companies. For a data engineer, most code execution is database-bound, not CPU-bound. Because of this, it makes sense to capitalize on Python's simplicity, even at the cost of slower performance when compared to compiled languages such as C# and Java.

Answering Data Engineer Interview Questions

Now that you know what your role might consist of, it's time to learn how to answer some data engineer interview questions! While there's a lot of ground to cover, you'll see practical Python examples throughout the tutorial to guide you along the way.

Questions on Relational Databases

Databases are one of the most crucial components in a system. Without them, there can be no state and no history. While you may not have considered database design to be a priority, know that it can have a significant impact on how quickly your page loads. In the past few years, several large corporations have introduced several new tools and techniques:

- NoSQL
- Cache databases
- Graph databases
- NoSQL support in SQL databases

These and other techniques were invented to try and increase the speed at which databases process requests. You'll likely need to talk about these concepts in your data engineer interview, so let's go over some questions!

Q1: Relational vs Non-Relational Databases

A **relational database** is one where data is stored in the form of a table. Each table has a **schema**, which is the columns and types a record is required to have. Each schema must have at least one primary key that uniquely identifies that record. In other words, there are no duplicate rows in your database. Moreover, each table can be related to other tables using foreign keys.

One important aspect of relational databases is that a change in a schema must be applied to all records. This can sometimes cause breakages and big headaches during migrations. **Non-relational databases** tackle things in a different way. They are inherently schema-less, which means that records can be saved with different schemas and with a different, nested structure. Records can still have primary keys, but a change in the schema is done on an entry-by-entry basis.

You would need to perform a speed comparison test based on the type of function being performed. You can choose `INSERT`, `UPDATE`, `DELETE`, or another function. Schema design, indices, the number of aggregations, and the number of records will also affect this analysis, so you'll need to test thoroughly. You'll learn more about how to do this later on.

Databases also differ in **scalability**. A non-relational database may be less of a headache to distribute. That's because a collection of related records can be easily stored on a particular node. On the other hand, relational databases require more thought and usually make use of a master-slave system.

A SQLite Example

Now that you've answered what relational databases are, it's time to dig into some Python! SQLite is a convenient database that you can use on your local machine. The database is a single file, which makes it ideal for prototyping purposes. First, import the required Python library and create a new database:

Python

```python
import sqlite3

db = sqlite3.connect(':memory:')  # Using an in-memory database
cur = db.cursor()
```

You're now connected to an in-memory database and have your cursor object ready to go.

Next, you'll create the following three tables:

1. **Customer:** This table will contain a primary key as well as the customer's first and last names.
2. **Items:** This table will contain a primary key, the item name, and the item price.
3. **Items Bought**: This table will contain an order number, date, and price. It will also connect to the primary keys in the Items and Customer tables.

Now that you have an idea of what your tables will look like, you can go ahead and create them:

Python

```python
cur.execute('''CREATE TABLE IF NOT EXISTS Customer (
                id integer PRIMARY KEY,
                firstname varchar(255),
                lastname varchar(255) )''')
cur.execute('''CREATE TABLE IF NOT EXISTS Item (
                id integer PRIMARY KEY,
                title varchar(255),
                price decimal )''')
cur.execute('''CREATE TABLE IF NOT EXISTS BoughtItem (
                ordernumber integer PRIMARY KEY,
                customerid integer,
                itemid integer,
                price decimal,
                CONSTRAINT customerid
                    FOREIGN KEY (customerid) REFERENCES Customer(id),
                CONSTRAINT itemid
                    FOREIGN KEY (itemid) REFERENCES Item(id) )''')
```

You've passed a query to `cur.execute()` to create your three tables.

The last step is to populate your tables with data:

Python

```python
cur.execute('''INSERT INTO Customer(firstname, lastname)
               VALUES ('Bob', 'Adams'),
                      ('Amy', 'Smith'),
                      ('Rob', 'Bennet');''')
cur.execute('''INSERT INTO Item(title, price)
               VALUES ('USB', 10.2),
                      ('Mouse', 12.23),
                      ('Monitor', 199.99);''')
cur.execute('''INSERT INTO BoughtItem(customerid, itemid, price)
               VALUES (1, 1, 10.2),
                      (1, 2, 12.23),
                      (1, 3, 199.99),
                      (2, 3, 180.00),
                      (3, 2, 11.23);''') # Discounted price
```

Now that there are a few records in each table, you can use this data to answer a few more data engineer interview questions.

Your Guide to the Python Programming Language and a Best Practices Handbook
python-guide.org
The Hitchhiker's Guide to Python

Q2: SQL Aggregation Functions

Aggregation functions are those that perform a mathematical operation on a result set. Some examples include `AVG`, `COUNT`, `MIN`, `MAX`, and `SUM`. Often, you'll need `GROUP BY` and `HAVING` clauses to complement these aggregations. One useful aggregation function is `AVG`, which you can use to compute the mean of a given result set:

Python >>>

```python
>>> cur.execute('''SELECT itemid, AVG(price) FROM BoughtItem GROUP BY itemid''')
>>> print(cur.fetchall())
[(1, 10.2), (2, 11.73), (3, 189.995)]
```

Here, you've retrieved the average price for each of the items bought in your database. You can see that the item with an `itemid` of `1` has an average price of $10.20.

To make the above output easier to understand, you can display the item name instead of the `itemid`:

Python >>>

```
>>> cur.execute('''SELECT item.title, AVG(boughtitem.price) FROM BoughtItem as boughtitem
...             INNER JOIN Item as item on (item.id = boughtitem.itemid)
...             GROUP BY boughtitem.itemid''')
...
>>> print(cur.fetchall())
[('USB', 10.2), ('Mouse', 11.73), ('Monitor', 189.995)]
```

Now, you see more easily that the item with an average price of $10.20 is the USB.

Another useful aggregation is SUM. You can use this function to display the total amount of money that each customer spent:

Python >>>

```
>>> cur.execute('''SELECT customer.firstname, SUM(boughtitem.price) FROM BoughtItem as boughtitem
...             INNER JOIN Customer as customer on (customer.id = boughtitem.customerid)
...             GROUP BY customer.firstname''')
...
>>> print(cur.fetchall())
[('Amy', 180), ('Bob', 222.42000000000002), ('Rob', 11.23)]
```

On average, the customer named Amy spent about $180, while Rob only spent $11.23!

If your interviewer likes databases, then you might want to brush up on nested queries, join types, and the steps a relational database takes to perform your query.

Q3: Speeding Up SQL Queries

Speed depends on various factors, but is mostly affected by how many of each of the following are present:

- Joins
- Aggregations
- Traversals
- Records

The greater the number of joins, the higher the complexity and the larger the number of traversals in tables. Multiple joins are quite expensive to perform on several thousands of records involving several tables because the database also needs to cache the intermediate result! At this point, you might start to think about how to increase your memory size.

Speed is also affected by whether or not there are **indices** present in the database. Indices are extremely important and allow you to quickly search through a table and find a match for some column specified in the query.

Indices sort the records at the cost of higher insert time, as well as some storage. Multiple columns can be combined to create a single index. For example, the columns `date` and `price` might be combined because your query depends on both conditions.

Q4: Debugging SQL Queries

Most databases include an `EXPLAIN QUERY PLAN` that describes the steps the database takes to execute the query. For SQLite, you can enable this functionality by adding `EXPLAIN QUERY PLAN` in front of a `SELECT` statement:

Python >>>

```
>>> cur.execute('''EXPLAIN QUERY PLAN SELECT customer.firstname, item.title,
...                item.price, boughtitem.price FROM BoughtItem as boughtitem
...                INNER JOIN Customer as customer on (customer.id = boughtitem.customerid)
...                INNER JOIN Item as item on (item.id = boughtitem.itemid)''')
...
>>> print(cur.fetchall())
[(4, 0, 0, 'SCAN TABLE BoughtItem AS boughtitem'),
(6, 0, 0, 'SEARCH TABLE Customer AS customer USING INTEGER PRIMARY KEY (rowid=?)'),
(9, 0, 0, 'SEARCH TABLE Item AS item USING INTEGER PRIMARY KEY (rowid=?)')]
```

This query tries to list the first name, item title, original price, and bought price for all the bought items.

Here's what the query plan itself looks like:

SQL

```
SCAN TABLE BoughtItem AS boughtitem
SEARCH TABLE Customer AS customer USING INTEGER PRIMARY KEY (rowid=?)
SEARCH TABLE Item AS item USING INTEGER PRIMARY KEY (rowid=?)
```

Note that fetch statement in your Python code only returns the explanation, but not the results. That's because `EXPLAIN QUERY PLAN` is not intended to be used in production.

Questions on Non-Relational Databases

In the previous section, you laid out the differences between relational and non-relational databases and used SQLite with Python. Now you're going to focus on NoSQL. Your goal is to highlight its strengths, differences, and use cases.

A MongoDB Example

You'll use the same data as before, but this time your database will be MongoDB. This NoSQL database is document-based and scales very well. First things first, you'll need to install the required Python library:

Shell

```
$ pip install pymongo
```

You also might want to install the MongoDB Compass Community. It includes a local IDE that's perfect for visualizing the database. With it, you can see the created records, create triggers, and act as visual admin for the database.

Note: To run the code in this section, you'll need a running database server. To learn more about how to set it up, check out Introduction to MongoDB and Python.

Here's how you create the database and insert some data:

Python

```
import pymongo

client = pymongo.MongoClient("mongodb://localhost:27017/")

# Note: This database is not created until it is populated by some data
db = client["example_database"]

customers = db["customers"]
items = db["items"]

customers_data = [{ "firstname": "Bob", "lastname": "Adams" },
                  { "firstname": "Amy", "lastname": "Smith" },
                  { "firstname": "Rob", "lastname": "Bennet" },]
items_data = [{ "title": "USB", "price": 10.2 },
              { "title": "Mouse", "price": 12.23 },
              { "title": "Monitor", "price": 199.99 },]

customers.insert_many(customers_data)
items.insert_many(items_data)
```

As you might have noticed, MongoDB stores data records in **collections**, which are the equivalent to a list of dictionaries in Python. In practice, MongoDB stores BSON documents.

Q5: Querying Data With MongoDB

Let’s try to replicate the `BoughtItem` table first, as you did in SQL. To do this, you must append a new field to a customer. MongoDB’s documentation specifies that the keyword operator **set** can be used to update a record without having to write all the existing fields:

Python

```
# Just add "boughtitems" to the customer where the firstname is Bob
bob = customers.update_many(
        {"firstname": "Bob"},
        {
            "$set": {
                "boughtitems": [
                    {
                        "title": "USB",
                        "price": 10.2,
                        "currency": "EUR",
                        "notes": "Customer wants it delivered via FedEx",
                        "original_item_id": 1
                    }
                ]
            },
        }
    )
```

Notice how you added additional fields to the `customer` without explicitly defining the schema beforehand. Nifty!

In fact, you can update another customer with a slightly altered schema:

Python

```
amy = customers.update_many(
        {"firstname": "Amy"},
        {
            "$set": {
                "boughtitems":[
                    {
                        "title": "Monitor",
                        "price": 199.99,
                        "original_item_id": 3,
                        "discounted": False
                    }
                ]
            } ,
        }
    )
print(type(amy))  # pymongo.results.UpdateResult
```

Similar to SQL, document-based databases also allow queries and aggregations to be executed. However, the functionality can differ both syntactically and in the underlying execution. In fact, you might have noticed that MongoDB reserves the `$` character to specify some command or aggregation on the records, such as `$group`. You can learn more about this behavior in the official docs.

You can perform queries just like you did in SQL. To start, you can create an index:

Python >>>

```
>>> customers.create_index([("name", pymongo.DESCENDING)])
```

This is optional, but it speeds up queries that require name lookups.

Then, you can retrieve the customer names sorted in ascending order:

Python >>>

```
>>> items = customers.find().sort("name", pymongo.ASCENDING)
```

You can also iterate through and print the bought items:

Python >>>

```
>>> for item in items:
...     print(item.get('boughtitems'))
...
None
[{'title': 'Monitor', 'price': 199.99, 'original_item_id': 3, 'discounted': False}]
[{'title': 'USB', 'price': 10.2, 'currency': 'EUR', 'notes': 'Customer wants it delivered via FedEx'
```

You can even retrieve a list of unique names in the database:

Python >>>

```
>>> customers.distinct("firstname")
['Bob', 'Amy', 'Rob']
```

Now that you know the names of the customers in your database, you can create a query to retrieve information about them:

Python >>>

```
>>> for i in customers.find({"$or": [{'firstname':'Bob'}, {'firstname':'Amy'}]},
...                                  {'firstname':1, 'boughtitems':1, '_id':0}):
...     print(i)
...
{'firstname': 'Bob', 'boughtitems': [{'title': 'USB', 'price': 10.2, 'currency': 'EUR', 'notes': 'Cu
{'firstname': 'Amy', 'boughtitems': [{'title': 'Monitor', 'price': 199.99, 'original_item_id': 3, 'd
```

Here's the equivalent SQL query:

SQL

```
SELECT firstname, boughtitems FROM customers WHERE firstname LIKE ('Bob', 'Amy')
```

Note that even though the syntax may differ only slightly, there's a drastic difference in the way queries are executed underneath the hood. This is to be expected because of the different query structures and use cases between SQL and NoSQL databases.

Q6: NoSQL vs SQL

If you have a constantly changing schema, such as financial regulatory information, then NoSQL can modify the records and nest related information. Imagine the number of joins you'd have to do in SQL if you had eight orders of nesting! However, this situation is more common than you would think.

Now, what if you want to run reports, extract information on that financial data, and infer conclusions? In this case, you need to run complex queries, and SQL tends to be faster in this respect.

Note: SQL databases, particularly PostgreSQL, have also released a feature that allows queryable JSON data to be inserted as part of a record. While this can combine the best of both worlds, speed may be of concern.

It's faster to query unstructured data from a NoSQL database than it is to query JSON fields from a JSON-type column in PostgreSQL. You can always do a speed comparison test for a definitive answer.

Nonetheless, this feature might reduce the need for an additional database. Sometimes, pickled or serialized objects are stored in records in the form of binary types, and then de-serialized on read.

Speed isn't the only metric, though. You'll also want to take into account things like transactions, atomicity, durability, and scalability. **Transactions** are important in financial applications, and such features take precedence.

Since there's a wide range of databases, each with its own features, it's the data engineer's job to make an informed decision on which database to use in each application. For more information, you can read up on ACID properties relating to database transactions.

You may also be asked what other databases you know of in your data engineer interview. There are several other relevant databases that are used by many companies:

- **Elastic Search** is highly efficient in text search. It leverages its document-based database to create a powerful search tool.
- **Newt DB** combines ZODB and the PostgreSQL JSONB feature to create a Python-friendly NoSQL database.
- **InfluxDB** is used in time-series applications to store events.

The list goes on, but this illustrates how a wide variety of available databases all cater to their niche industry.

Questions on Cache Databases

Cache databases hold frequently accessed data. They live alongside the main SQL and NoSQL databases. Their aim is to alleviate load and serve requests faster.

A Redis Example

You've covered SQL and NoSQL databases for long-term storage solutions, but what about faster, more immediate storage? How can a data engineer change how fast data is retrieved from a database?

Typical web-applications retrieve commonly-used data, like a user's profile or name, very often. If all of the data is contained in one database, then the number of **hits** the database server gets is going to be over the top and unnecessary. As such, a faster, more immediate storage solution is needed.

While this reduces server load, it also creates two headaches for the data engineer, backend team, and DevOps team. First, you'll now need some database that has a faster read time than your main SQL or NoSQL database. However, the contents of both databases must eventually match. (Welcome to the problem of **state consistency** between databases! Enjoy.)

The second headache is that DevOps now needs to worry about scalability, redundancy, and so on for the new cache database. In the next section, you'll dive into issues like these with the help of Redis.

Q7: How to Use Cache Databases

You may have gotten enough information from the introduction to answer this question! A **cache database** is a fast storage solution used to store short-lived, structured, or unstructured data. It can be partitioned and scaled according to your needs, but it's typically much smaller in size than your main database. Because of this, your cache database can reside in memory, allowing you to bypass the need to read from a disk.

Note: If you've ever used dictionaries in Python, then Redis follows the same structure. It's a key-value store, where you can `SET` and `GET` data just like a Python `dict`.

When a request comes in, you first check the cache database, then the main database. This way, you can prevent any unnecessary and repetitive requests from reaching the main database's server. Since a cache database has a lower read time, you also benefit from a performance increase!

You can use pip to install the required library:

Shell
```
$ pip install redis
```

Now, consider a request to get the user's name from their ID:

Python

```python
import redis
from datetime import timedelta

# In a real web application, configuration is obtained from settings or utils
r = redis.Redis()

# Assume this is a getter handling a request
def get_name(request, *args, **kwargs):
    id = request.get('id')
    if id in r:
        return r.get(id)  # Assume that we have an {id: name} store
    else:
        # Get data from the main DB here, assume we already did it
        name = 'Bob'
        # Set the value in the cache database, with an expiration time
        r.setex(id, timedelta(minutes=60), value=name)
        return name
```

This code checks if the name is in Redis using the `id` key. If not, then the name is set with an expiration time, which you use because the cache is short-lived.

Now, what if your interviewer asks you what's wrong with this code? Your response should be that there's no exception handling! Databases can have many problems, like dropped connections, so it's always a good idea to try and catch those exceptions.

Questions on Design Patterns and ETL Concepts

In large applications, you'll often use more than one type of database. In fact, it's possible to use PostgreSQL, MongoDB, and Redis all within just one application! One challenging problem is dealing with state changes between databases, which exposes the developer to issues of consistency. Consider the following scenario:

1. **A value** in Database #1 is updated.
2. **That same value** in Database #2 is kept the same (not updated).
3. **A query** is run on Database #2.

Now, you've got yourself an inconsistent and outdated result! The results returned from the second database won't reflect the updated value in the first one. This can happen with any two databases, but it's especially common when the main database is a NoSQL database, and information is transformed into SQL for query purposes.

Databases may have background workers to tackle such problems. These workers **extract** data from one database, **transform** it in some way, and **load** it into the target database. When you're converting from a NoSQL database to a SQL one, the Extract, transform, load (ETL) process takes the following steps:

1. **Extract:** There is a MongoDB trigger whenever a record is created, updated, and so on. A callback function is called asynchronously on a separate thread.
2. **Transform:** Parts of the record are extracted, normalized, and put into the correct data structure (or row) to be inserted into SQL.
3. **Load:** The SQL database is updated in batches, or as a single record for high volume writes.

This workflow is quite common in financial, gaming, and reporting applications. In these cases, the constantly-changing schema requires a NoSQL database, but reporting, analysis, and aggregations require a SQL database.

Q8: ETL Challenges

There are several challenging concepts in ETL, including the following:

- Big data

- Stateful problems
- Asynchronous workers
- Type-matching

The list goes on! However, since the steps in the ETL process are well-defined and logical, the data and backend engineers will typically worry more about performance and availability rather than implementation.

If your application is writing thousands of records per second to MongoDB, then your ETL worker needs to keep up with transforming, loading, and delivering the data to the user in the requested form. Speed and latency can become an issue, so these workers are typically written in fast languages. You can use compiled code for the transform step to speed things up, as this part is usually CPU-bound.

Note: Multi-processing and separation of workers are other solutions that you might want to consider.

If you're dealing with a lot of CPU-intensive functions, then you might want to check out Numba. This library compiles functions to make them faster on execution. Best of all, this is easily implemented in Python, though there are some limitations on what functions can be used in these compiled functions.

Q9: Design Patterns in Big Data

Imagine Amazon needs to create a **recommender system** to suggest suitable products to users. The data science team needs data and lots of it! They go to you, the data engineer, and ask you to create a separate staging database warehouse. That's where they'll clean up and transform the data.

You might be shocked to receive such a request. When you have terabytes of data, you'll need multiple machines to handle all of that information. A database aggregation function can be a very complex operation. How can you query, aggregate, and make use of relatively big data in an efficient way?

Apache had initially introduced MapReduce, which follows the **map, shuffle, reduce** workflow. The idea is to map different data on separate machines, also called clusters. Then, you can perform work on the data, grouped by a key, and finally, aggregate the data in the final stage.

This workflow is still used today, but it's been fading recently in favor of Spark. The design pattern, however, forms the basis of most big data workflows and is a highly intriguing concept. You can read more on MapReduce at IBM Analytics.

Q10: Common Aspects of the ETL Process and Big Data Workflows

You might think this a rather odd question, but it's simply a check of your computer science knowledge, as well as your overall design knowledge and experience.

Both workflows follow the **Producer-Consumer** pattern. A worker (the Producer) produces data of some kind and outputs it to a pipeline. This pipeline can take many forms, including network messages and triggers. After the Producer outputs the data, the Consumer consumes and makes use of it. These workers typically work in an asynchronous manner and are executed in separate processes.

You can liken the Producer to the extract and transform steps of the ETL process. Similarly, in big data, the **mapper** can be seen as the Producer, while the **reducer** is effectively the Consumer. This separation of concerns is extremely important and effective in the development and architecture design of applications.

Conclusion

Congratulations! You've covered a lot of ground and answered several data engineer interview questions. You now understand a bit more about the many different hats a data engineer can wear, as well as what your responsibilities are with respect to databases, design, and workflow.