7 concepts of databases every data engineer should know

If you are preparing for Data Engineering position interview you **must** know all the main concepts of databases. And it's not a click-bait. A few month ago I had several data engineering interviews and all of them included questions about the topics I'm mentioning in this article. Even if you are not preparing to the interview, you still may want to check those concepts and refresh them in your head.

Also, it will be useful not only for data engineers, but to a

wide variety of professional working with data from databases: data scientists, ML-engineers, software developers and many more.

My name is Oleg, I'm the author of open-source "Data Engineering Book" which I publish on GitHub. I will try to explain 7 main concepts of databases as easy as possible. These concepts are:

1. relational model
2. data normalization
3. primary and foreign keys
4. indexes
5. transactions
6. replication
7. sharding

Let's get started!

Relational model

Relational model is an approach to structure and manage the data.

Table

	Field1	Field2
Record1	Value	Another value
Record2	Some value	Some another value

In this model, data is organized into **tables.** Each table

has a **schema**. It means that it has a predefined list of columns so only data satisfying the schema can be written into the table. Also, each column has a **data type** (number, string, boolean, etc). Columns of a table are usually called fields, and rows as records.

In original theory of databases, tables were called as **relations**, thus the naming of the model. Don't confuse this definition with **relations** between tables, where we usually use a key to define such relations. We will talk about keys later in this article.

Lastly, databases that follow this model are called relational. Relational databases use **SQL** (Structured Query Language) to access the data they store.

Data normalization

Normalization is a process of making your data suitable for relational databases.

Sometimes, normalization is referred as a **process of removing data redundancy**. And generally this definition usually is easy to explain and understand. Normalization helps to fight data redundancy, improve data integrity, simplify data structure, helps in finding errors.

The process of normalization is done via applying two methods:

- **synthesis** (e.g. creating new items that weren't existed before) or
- **decomposition** (improving existing data structure by split into smaller parts)

Consider a following example. Imagine that you run an electronics store and you write down every single purchase in the Excel spreadsheet. It could look something like that:

Purchase log

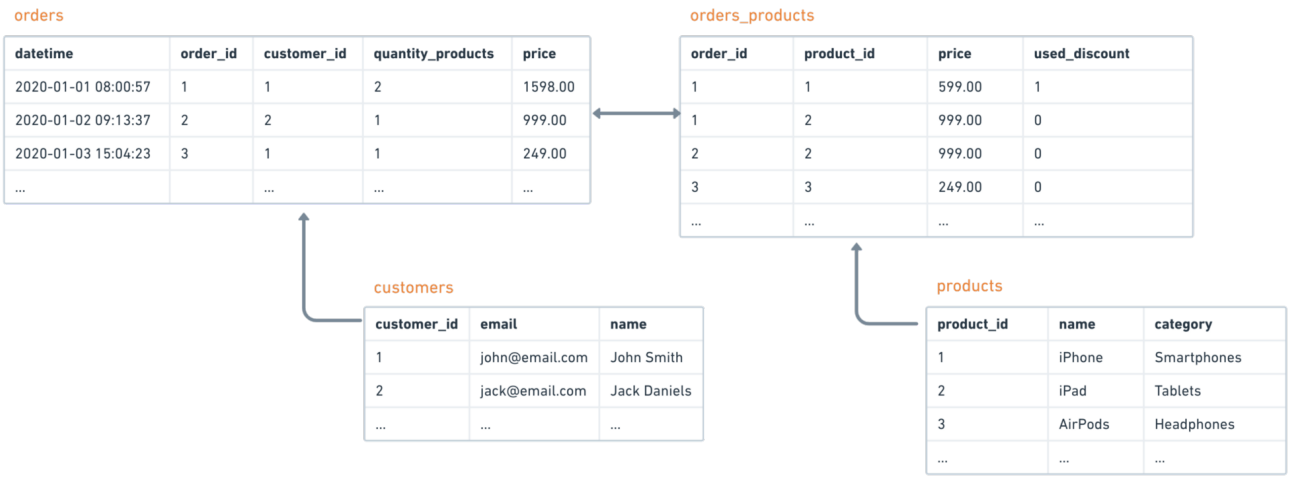
Purchase date	Customer's email	Customer's name	Purchased products	Total price, USD
2020-01-01 08:00:57	john@email.com	John Smith	iPhone, iPad	$1598.00
2020-01-02 09:13:37	jack@email.com	Jack Daniels	iPad	$999.00
2020-01-03 15:04:23	john@email.com	John Smith	AirPods	$249.00
...	...	...	...	...

While such data structure is acceptable in Excel, it may lead to a several problems when inserted to database, if you copy it 1-to-1:

- Data is redundant. For example, do you really need customer's name in here? Maybe email is enough and name can be stored in a separate table.
- It will be hard to query such data. For example, can you say how many iPhones have you sold and for which price? No, because "Total price" is already aggregated value and you don't know the cost of each item in the cart.
- Data update and removal is hard. Let's consider the first order with two items and suppose that customer has decided to return one item. What is the best way

to reflect this change in such table?

Data normalization will turn such purchase log into something like this:



In this structure, it will be easier add, update and delete entries. Also it will consume less space on a hard drive, because we replaced repeated data with identifiers and store them in separate tables.

Such normalization is not unique. There are rules which describe levels of normalization. There are 6 normal forms of data, each with its own set of rules and restrictions to the data structure.

Primary and foreign keys

Primary key is unique identifier of a record in a table.

Such key is needed when we want to create relation with another table. When table has a column containing a reference to another table, such column becomes a **foreign key**.

For example, if we have a `customers` table with a column named *customer_id*, it is a **primary key** for this table. At the same time, if table `orders` has *customer_id* column referencing this field in `customers` table, it becomes a **foreign key** for `orders` table.

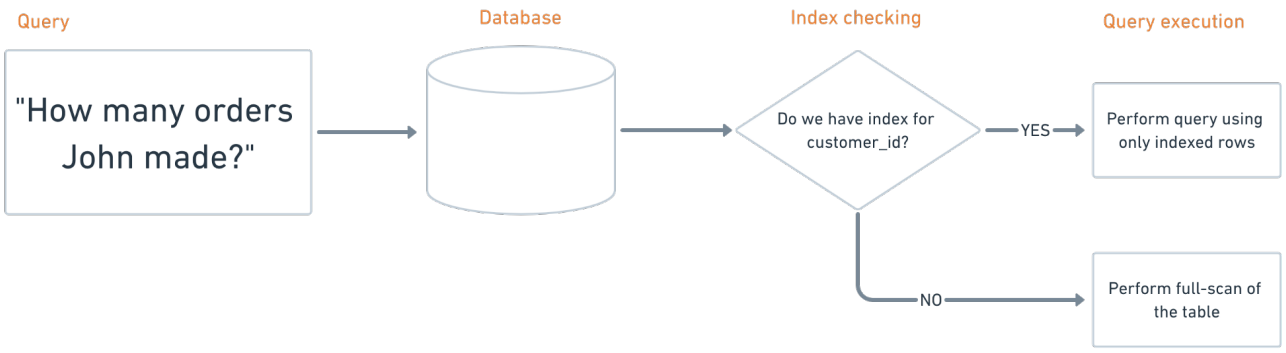
Indexes

Index is a special object inside a database which helps perform fast searches inside the data.

Imagine that you have a big table with *millions* of records and you need to find a subset of this table which satisfy a specific criteria, for example how many orders had a particular customer. In general, records are inserted to the table *without an order*. So to perform a search, the database will need to do a **full-scan**, meaning it will got

row-by-row, from beginning to end, until it find the needed rows.

To speed up the process we can **build an index** for a searched column. Such index will store **locations** of each values of the column in the table. So when performing a search using an indexed column, the database will first search the index to find locations of the data, and only then will extract needed rows using those locations.



Transactions

A term **transaction** usually means an **indivisible unit of work**. It is needed when we want to perform several operations in the database and we want to make sure that either all operations are succeeded or failed at once.

Classical example of transaction is *bank transfer*, when you need to transfer money from one account to another. Basically, there are three steps involved:

1. check that requested amount exist on account #1
2. subtract needed amount from account #1
3. add needed amount to account #2

If bank application started the transfer process, it **must guarantee** that either all three operations **are success** or all of them are **failed at once**.

To be sure that transactions are possible in a database, it should satisfy a set of requirements, known as **ACID** abbreviation.

ACID stands for Atomicity, Consistency, Isolation, Durability.

Atomicity is a guarantee that the operation (unit of work) will be fully executed. In other words, all changes to the data must be performed successfully or not performed at all.

Consistency means that the data should be in a consistent state before and after the transaction. Execution of this rule is fully on a business logic shoulders. Recall the example with bank transfer: if we subtracted $100 from account #1, we cannot add $200 to account#2 because it will cause an inconsistency.

Isolation means that parallel transaction shouldn't influence the outcome of each other. This requirement is quite expensive for databases, so in some databases

there are different levels of isolation.

Finally, **durability** guarantees that the result of the operation is persisted in database and won't be lost. For example, if user got a response that transaction is completed, he can be sure that made changes won't be discarded because of system failure or other outage.

Replication

Replication is a syncronization of our database to a different nodes or server. In other words, replication is a **process of copying** our data from one source to another.

Replication can secure us from a **data loss** if something will happen with our master copy of the data. For example, if our database is experiencing a downtime (e.g. lack of network or outage), our application won't be operational because it doesn't have any data to show to the users.

What replication gives us:

- replica is a **full copy** of our database
- changes in master copy **immediately applied** to a

replica
- if master database is down, all incoming requests can be **redirected** to replica
- a common case when all add/update/delete requests are routed to main database, but all reads to replica, making a nice **load balancing** for such architecture

There are two modes in which replicas can work: **synchronous** and **asynchronous**.

From the name, **synchronous mode** is when replica applies the same changes as on master instance and only after that user get a response from the database. Sync mode has *consistent data*, but usually *slower* in response time.

In **asynchronous mode**, master doesn't wait for the response from replica and immediately send the result of the operation back to user. Async mode may have some delays in data (be *not consistent*), but is very *fast* to response.

Sharding

Sharding is a way to split the data in a table by some key and send different parts to a different nodes.

Sharding is a **horizontal scalling**. We split a table into several **logical partitions**. Schema is the same for each partition (because we split data by rows, not by columns).

Each partitions represents a **logical shard** of the table.

After distributing across different nodes they become **physical shards**. One node of a database can hold multiple logical shards.

How sharding can be implemented? There are three common ways:

1. on **application level**. Basically, it is a job of your application to know where the needed data is stored.
2. on **database level**. The database itself decides on which node it should place the data. Of course, it is not done automatically, because the database needs a proper configuration to be provided upfront.
3. using **external coordination service**. Here you outsource sharding to a 3rd party service which decides where to store the data. Your application talks with this service instead of the database.

There are many way to implement sharding, but in all cases you need to provide a **distribution key**. This key decides how your data will be spread across the cluster. Also, there is no silver bullet about *how to choose a correct distribution key*. Two most common options for distribution key are **hash-based** or **value-based** keys.

Summary

I hope you enjoyed the article and learnt something new

along the way. If you want to know more about those topics in details, you could read “Introduction to databases” chapter of my “Data Engineering Book”.

And good luck with the interview!