Apache Kafka Foundation Course - What is Apache Kafka

<https://www.learningjournal.guru/courses/kafka/kafka-foundation-training/what-is-kafka/>

**delete.topic.enable**

If you want to delete a topic, you can use topic management tool. But by default, deleting a topic is not allowed. You can't remove a topic because the default value for this parameter is false. That is reasonable protection for production environments. But in development or testing environment, you may want to delete topics. So, if you want Kafka to allow deleting a topic, you need to set this parameter to true.

**auto.create.topics.enable**

We have already discussed auto-create topic feature. If a producer starts sending messages to a non-existent topic, Kafka will create the topic automatically and accept the data. This behaviour is suitable for dev environments. But in a production environment, you may want to implement a more controlled approach. You can set this parameter to false, and Kafka will stop creating topics automatically. You can create topics manually using the topic management tool, and no one will be able to send data to a non-existent topic.

**default.replication.factor and num.partitions**

These two parameters are quite straightforward. The default values for both of them is one, and they are effective when you have auto create topics enabled. So, if Kafka is creating your topic automatically, the new topic will have only one partition and a single copy. If you want some other values, you can change the default settings accordingly.

**log.retention.ms and log.retention.bytes**

These two are critical and not obvious. So, whatever data you send to Kafka, it is not retained by Kafka forever. Kafka is not a database. You don't send data to Kafka for storage so that you can query it later. It is a message broker. It should deliver the data to the consumer and then clean it up. There is no reason to retain messages for longer than needed.  
Kafka gives you two options to configure the retention period. The default option is retention by time, and the default retention period is seven days. So, in this case, Kafka will clean up all the messages older than seven days. If you want to change the duration, you can specify your value for log.retention.ms configuration.  
Kafka gives you another option to define this retention period. You can specify it by size. That's where the second parameter log retention bytes is applicable. But this size applies to partition. So, if you set log.retention.bytes = 1 GB, Kafka will trigger a clean-up activity when the partition size reaches to 1 GB. Remember that it is not a topic size. It is partition size.

kafka Producers common Properties configurations interview questions :

Acknowledgements-

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Diagram

Description automatically generated

Text

Description automatically generated with medium confidence

Text

Description automatically generated with medium confidence

[Diagram

Description automatically generated](https://www.learningjournal.guru/_resources/img/jpg-7x/kafka-producer-workflow.jpg)Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated**max.in.flight.requests.per.connection : 5. It means producer don’t want to wait for acknowledgement for each request it he wants only wait on every Fifth request for ack.**

Apache Kafka Foundation Course - What is Apache Kafka

Welcome to Apache Kafka Tutorial at Learning journal. In this session, I will introduce you to Kafka. We will try to understand Kafka in less than 10 minutes. I am assuming that you have at least heard about Kafka and you already know that it is an Open Source project. Kafka was initially developed at LinkedIn and later open sourced in 2011. Since then it has evolved and established itself as a standard tool for building real-time data pipelines. Now it's securing its share in real-time streaming applications as well.  
The Kafka documentation says it is a distributed streaming platform. That's good for definition. But I want to know what it can do for me or what I can do using Kafka.

Messaging System

The official documentation says that Apache Kafka is similar to enterprise messaging system. I guess, you already understand a messaging system. In a typical messaging system, there are three components.

Producer or Publisher

Broker

Consumer.

[Diagram

Description automatically generated](https://www.learningjournal.guru/_resources/img/jpg-5x/typical-messaging-system.jpg)Fig.1-Typical Messaging System

The producers are the client applications, and they send some messages.  
The Brokers receive those messages from publishers and store them.

The consumers read the message records from brokers.

Kafka Use Case

A messaging system looks very simple. Now let us look at the data integration problem in a large organization. I borrowed the below diagram from Jey Creps blog.

[Diagram

Description automatically generated](https://www.learningjournal.guru/_resources/img/jpg-5x/data-integration-problem.jpg)Fig.1-Data Integration Problem

The above diagram shows the data integration requirement in a large enterprise.  
Does it look like a mess?  
There are many source systems and multiple destination systems. And you are given a task to create data pipelines to move data among those systems. For a growing company, the number of source and destination systems keep getting bigger and bigger. Finally, your data pipeline looks like a mess. I am sure that I don't need to explain that you can't manage and maintain that kind of data pipeline. Some part of your pipeline will keep breaking every day.  
However, if we can use a messaging system for solving that kind of integration problem, the solution may be neater, and cleaner as shown below.

[Diagram

Description automatically generated](https://www.learningjournal.guru/_resources/img/jpg-5x/unified-messaging-system.jpg)Fig.1-Unified Messaging System

That's the idea discovered by the team at LinkedIn. Then they started evaluating existing messaging systems, but none of them meet their criteria to support the desired throughput and scale. Finally, they end up creating Kafka.

What is Kafka?

At the core, Kafka is a highly scalable and fault tolerant enterprise messaging system. Take a look at the [Apache Kafka diagram](https://kafka.apache.org/intro.html#_blank) from official documentation. I hope you understand the producer, consumer and the broker that the figure shows. At the top of the diagram, the Producer applications are sending messages to Kafka cluster. The Kafka cluster is nothing but a bunch of brokers running in a group of computers. They take message records from producers and store it in Kafka message log.  
At the bottom of the picture, there are consumer applications. They read messages from Kafka cluster, processes it and do whatever they want to do. They may want to send them to Hadoop, Cassandra, HBase or may be pushing it back again into Kafka for someone else to read these modified or transformed records.  
Now let us turn our focus on other two things in this diagram.

Kafka Streams

Let me ask a question. What is a stream?  
Well, I will say continuous flow of data. or you can define it as a constant stream of messages.  
Kafka, as a messaging system is so powerful regarding throughput and scalability that it allows you to handle a continuous stream of messages. If you can just plug in some stream processing framework to Kafka, it could be your backbone infrastructure to create a real-time stream processing application. And that is what right side of the diagram is trying to explain. Those are some stream processing applications. They read a continuous stream of data from Kafka, process them and then either store them back in Kafka or send them directly to other systems. Kafka provides some stream processing APIs as well. So you can do a lot of things using Kafka stream processing APIs, or you can use other stream processing frameworks like Spark streaming or Storm.

Kafka Connect

The next thing is Kafka connector. These are the most compelling features. They are ready to use connectors to import data from databases into Kafka or export data from Kafka to databases. These are not just out of the box connectors but also a framework to build specialized connectors for any other application.

Summary

Let us summarize all that we learned in this session.

Kafka is a distributed streaming platform. You can use it as an enterprise messaging system. That doesn't mean just a traditional messaging system. You can use it to simplify complex data pipelines that are made up of a vast number of consumers and producers.

You can use it as a stream processing platform. There are two parts of stream processing. Stream and a Processing framework. Kafka gives you a stream, and you can plug in a processing framework.

Kafka also provides connectors to export and import bulk data from databases and other systems.

But implementing these things is not that simple. There is no plug and play component. You need to use APIs and write a bunch of code. You need to understand some configuration parameters and tune or customize Kafka behavior according to your requirement and use case.  
We will cover all these things in this training. So, keep watching.

Welcome to Apache Kafka tutorial at Learning journal. In this session, we will talk about some basic concepts associated with Kafka. The objective of this article is to introduce you to the main terminologies and build a foundation to understand and grasp rest of the training.  
In this session, we will cover following things.

Producer

Consumer

Broker

Cluster

Topic

Partitions

Offset

Consumer Groups

We will be using these terms extensively during our discussion of Apache Kafka. It is crucial that we both, myself and you have the same understanding of these concepts. So let me explain my understanding of these terms and some other related concepts associated with these keywords.

Kafka Producer

The first item is the producer. So, what is a producer?  
The producer is an application that sends data. Some people call it data, but we will call it a message or a message record. These messages can be anything ranging from a simple string to a complex object. Ultimately it is small to the medium-size piece of data. The message may have different meaning or schema for us. But for Kafka, it is a simple array of bytes.  
For example, if I want to send a file to Kafka, I will create a producer application and push each line of the file as a message. In this case, a message is one line of text. But for Kafka, it is just an array of bytes. Similarly, If I want to send all the records from a table, I will submit each row as a message, or if I want to send the result of a query. I will create a producer application, fire a query against my database, collect the result and start throwing each row as a message. So, while working with Kafka, if you want to send some data, you have to create a producer application. It is unlikely that you get a readymade producer that fits your purpose.

Kafka Consumer

The next thing is the consumer. The consumer is again an application that receives data. If producers are sending data, they must be sending it to someone. Right? The consumers are the recipients. But remember that the producers don't send data to a recipient address. They just send it to Kafka server. And anyone who is interested in that data can come forward and take it from Kafka server. So, any application that requests data from a Kafka server is a consumer, and they can ask for data send by any producer provided they have permissions to read it.  
So just continuing the file example, If I want to read the file sent by a producer, I will create a consumer application, then I will request Kafka for the data. The Kafka server will send me some messages. I think you remember that each message is a line of text in this example.  
So, the client application will receive some lines from Kafka server, it will process them and again request for some more messages. The client keeps demanding data, and Kafka server will keep giving message records as long as new messages are coming from the producer.

Kafka Broker

Now, let’s move on and try to understand a Broker. The broker is Kafka server. It is just a meaningful name given to Kafka server. And this title makes sense as well because all that Kafka does is act as a message broker between producer and consumer. The producer and consumer don not interact directly. They use Kafka server as an agent or a broker to exchange messages.

Kafka Cluster

Let's come to the next term. The cluster. This one is simple. If you have any background in distributed systems, you already know that a Cluster is a group of computers acting together for a common purpose. Since Kafka is a distributed system, so the cluster has the same meaning for Kafka. It is merely a group of computers, each executing one instance of Kafka broker.

Kafka Topic

Next item is the topic. We learned that producer sends data to Kafka broker. Then a consumer can ask for data from the Kafka broker. But the question is, Which data?  
Let's try to understand this by a simple conversation between Broker and the consumer.  
  
Broker - I am collecting data from multiple producers, which one do you want?  
Consumer - Give the data sent by producer ABC.  
Broker - Oh Man, producer ABC is pushing three different types of records. Which one do you want?  
Consumer - Well, send me the sales data.  
Broker - Ok, so you are looking for sales data. Two more producers are sending sales data.  
Consumer - Gosh, we need to have some identification mechanism.  
  
There comes the notion of the Kafka Topic. So, the topic is an arbitrary name given to a data set. We better say that it is a unique name for a data stream.  
For example, we create a topic called Global Orders, and every point of sales may have a producer. They send their order details as a message to the single Topic named Global Orders. And a subscriber interested in Orders can subscribe to the same Topic.

Kafka Partitions

By now, you learned that the broker would store data for a topic. This data can be enormous. It may be larger than the storage capacity of a single computer. In that case, the broker may have a challenge in storing that data. One of the obvious solutions is to break it into two or more parts and distribute it to multiple computers. Kafka is a distributed system that runs on a cluster of machines. So, it is self-evident that Kafka can break a topic into partitions and store one partition on one computer. And that's what the Partition means.  
You may be wondering that how Kafka will decide on the number of partitions. I mean, some topics may be large, but others may be relatively small. So how Kafka knows that it should create 100 partitions or just ten partitions could be enough?  
The answer is simple. Kafka doesn't take that decision. We, as a developer make that decision. When we create a topic, we make that decision, and Kafka broker will create that many partitions for our Topic. But remember that every Partition sits on a single machine. You can't break it again. So, do some estimation and simple math to calculate the number of partitions.

Offsets

Let's talk about offset. The offset is simple. It is a sequence number of a message in a partition. This number is assigned as the messages arrive in a partition. And these numbers, once assigned, they never change. They are immutable. This sequencing means that Kafka stores messages in the order of arrival within a partition. The first message gets an offset zero. The next message receives an offset one and so on. But remember that there is no global offset across partitions. Offsets are local to the partition. So, if you want to locate a message, you should know three things.  
Topic name, Partition number, and an offset number. If you have these three things, you can directly locate a message.

Kafka Consumer Groups

Now we are left with the last thing. The consumer groups. We already understand the Consumer. What is Consumer Group?  
It is a group of consumers. Several Consumers form a group to share the work. You can think of it like there is one large task and you want to divide it among multiple people, so you create a group, and members of the same group share the work. Let me give you an example.

A Kafka Example

Let's assume that we have a retail chain. In every store, there are few billing counters. You want to bring all the invoices from every billing counter to your data centre. Since you learned Kafka and you find Kafka as an excellent solution to transport data from billing locations to the data centre. You decided to implement it. The first thing you might want to do is to create a producer at every billing site. These Producers will send bills as a message to a Kafka Topic. The next thing you might want to do is to create a consumer. The Consumer will read data from Kafka Topic and write them into your data centre. It sounds like a perfect solution. Right? But there is a small problem. Think of the scale. You have hundreds of producers pushing data into a single topic. How will you handle that volume and velocity?

[Diagram

Description automatically generated](https://www.learningjournal.guru/_resources/img/jpg-7x/apache-kafka-example.jpg)Fig.1-Apache Kafka Example Use Case

You learned Kafka exceptionally well. So, you decided to create large Kafka cluster and partition your Topic. Correct? So, your Topic is partitioned and distributed across the Cluster. Now several brokers are sharing the workload to receive and store data. From the source side, you have many producer and several Brokers to share the workload. What about the destination side? You have a single unfortunate consumer.  
There comes the Consumer group. You create a Consumer group and start executing many Consumers and tell them to divide the work.  
So far so good. But how do we split the work? That's not a difficult question. I have 600 partitions. And I am starting 100 consumers. So why don't each of the consumer take six partitions? We will see, if they can't handle six partitions, we will start some more Consumers in the same group. We can go up to 600 Consumers, so each consumer will have just one partition to read.  
If you followed this example correctly, you understand that partitioning and consumer group is a tool for scalability. And notice that the maximum number of Consumers in a group is the total number of partitions you have on a topic. Kafka doesn't allow more than one Consumer to read from the same partition simultaneously. This restriction is necessary to avoid double reading of records.  
Great. I hope you learned core concepts of Kafka. Now you are familiar with the essential terminology that we will be using throughout the Kafka tutorials.

Apache Kafka Foundation Course - Installing Kafka in Google Cloud

Welcome back to Apache Kafka Tutorial. This video is an update for Apache Kafka 1.x. Many of my viewers don't have access to a Linux machine. Some of them had a Linux machine, but they struggle to download and install JDK and Kafka. This video will help them to quickly setup Apache Kafka in a Google Cloud VM. Google cloud VMs are quite cheap, and if you are a first-time user, they offer one-year free access to various Cloud services. This video will help you to get quick access to latest Kafka VM in Google Cloud. I recommend all my students and followers to gain access to GCP. I have a video tutorial as well to [set up your free GCP account](https://www.learningjournal.guru/article/google-cloud/free-learning-virtual-machine/#_blank). You can take advantage of free GCP service for your learning efforts.  
Great. Let's start.

Installing Kafka in Google Cloud

I am assuming you already have access to Google Cloud Account. You can follow these steps to set up a single node Kafka VM in Google Cloud.

Login to your GCP account.

Go to GCP products and services menu.

Click Cloud Launcher

Search for Kafka. You will see multiple options. For a single node setup, I prefer to use the Google VM Image. You can also try the single node Bitnami image. There is a multi-node Bitnami Image as well. However, they designed it for production usage with larger VM configurations. For your learning purposes, the Google image is good enough.

Select the Kafka VM Image.

You will notice the Kafka version, Operating system, and other packages.

Your Kafka usage is free, but you have a cost associated with the VM for CPU, Memory and the disk space. But don't worry about it. Google is charging you on a per hour basis. You also have a free credit for a year.

Click the launch button. You can review and change some configurations, but the default settings are fair enough.

Click the Deploy button at the bottom of the page. That's it. Just wait for few minutes and GCP will launch your single node Kafka VM.

You can SSH to the VM from your deployment page, or you can go to your homepage, then visit the compute engine page, and SSH to your VM.

Once you finish your work, you can select the VM and stop it. Your billing will stop. You can come back next day, pick the VM and start it again.

You Kafka VM comes preconfigured. All services are up and running. You can start using it right away.

Kafka Quick Start

Let's do some simple things. Go to [Kafka documentation quick start page](https://kafka.apache.org/quickstart#_blank). You don't need to perform step 1 and step 2. We already completed the setup, and the services are already running. You can copy the command to create a Kafka topic and execute it. Our Kafka VM comes with appropriate PATH settings. So, we just copy the command. There is no need to specify the path.

|  |  |
| --- | --- |
|  | kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test |

Next command is to list the available topics.

|  |  |
| --- | --- |
|  | kafka-topics.sh --list --zookeeper localhost:2181 |

You can open two SSH windows. Start a console consumer in one window and a console producer in another window. Let's do that. Start the producer.

|  |  |
| --- | --- |
|  | kafka-console-producer.sh --broker-list localhost:9092 --topic test |

Start a consumer.

|  |  |
| --- | --- |
|  | kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning |

Now, you can type some messages in the producer window. You will see them arriving in the consumer window. Press Control+C to terminate the consumer and similarly the producer.  
If you no longer need the Kafka VM, navigate to deployment manager and delete it. Removing your VM directly, doesn't clean up everything. So, instead of just deleting your VM from compute instance page, make sure to delete it from your deployments page.  
Great. That's it for this session.  
Thank you for watching Learning Journal. Keep learning and Keep growing

Apache Kafka Foundation Course - Quick Start Demo

Welcome to Apache Kafka tutorial at Learning Journal. In this video, I will provide a quick start demo. We will cover following things.

Download and Install Apache Kafka

Start Kafka server

Create a topic

Start a console producer

Start a console consumer

Send and receive messages

In fact, I am going to follow quick start guide from [Apache Kafka documentation](https://kafka.apache.org/0101/documentation.html#_blank). I will also explain few things along the way, and this demo will provide a good sense of some command line tools that Kafka provides.  
If you want to play with Apache Kafka and follow the example discussed in this tutorial, you will need a Linux machine. I have a virtual machine installed on my windows box, and I am going to use it for this demo.

Download and Install Apache Kafka

The first thing is to download and install Kafka. You can find the [download link in Kafka documentation](https://www.apache.org/dyn/closer.cgi?path=/kafka/0.10.1.0/kafka_2.11-0.10.1.0.tgz#_blank). So, go ahead and download Apache Kafka using the link. I downloaded Kafka 0.10.1 release because that's the latest version at the time of recording this video. However, you can follow the [latest Kafka quick start](https://kafka.apache.org/quickstart#_blank) from the official Kafka documentation.  
The download will provide you a tar file. I created a directory named Kafka in my home directory and placed the tar file in that directory. Now I need to uncompress this file. So, let's do it.

The below command will uncompress the tar file in the current directory.

|  |  |
| --- | --- |
|  | tar -zxvf kafka\_2.11-0.10.1.0.tgz |

That's it. You installed Apache Kafka. That's all about the installation.  
If you list your current directory, you will see that a new directory is present. Change to that directory.

|  |  |
| --- | --- |
|  | cd kafka\_2.11-0.10.1.0 |

You can refer this new directory as the Kafka Home.  
If you list you Kafka home, you will see several folders.  
The bin folder contains some command line Kafka tools. We will use most of those tools in this tutorial, and I will explain those tools when we use them for the first time.  
The config directory contains all the configuration files. We will explore some configuration files as well in this tutorial.  
The libs directory contains all the jar files.  
The logs directory is the data directory. Kafka stores all the messages in that folder.

Start Kafka Server

The next step is to start the first Kafka broker. But Kafka uses Zookeeper. Those who don't know about zookeeper, let me give you a quick bite on zookeeper.  
Zookeeper is another open source project that came out from Hadoop project. Zookeeper is used to provide some coordination services for a distributed system. Since Kafka is a distributed system, and we have multiple brokers. So, we need a system to coordinate various things among these brokers. That's why we need zookeeper.  
So, before we start Kafka broker, we have to start a zookeeper.  
Kafka provides a command line tool to start a zookeeper. It's available in your bin directory. So let's go ahead and start zookeeper.

|  |  |
| --- | --- |
|  | bin/zookeeper-server-start.sh config/zookeeper.properties |

The zookeeper-server-start.sh is a shell script, and it takes one parameter. The parameter is a configuration file name. We can use the default config.  
By executing the above command, you should see a message, as the zookeeper is running on port 2181. Minimize the current terminal and start another one for starting a Kafka broker.  
Kafka provides a command line tool to start the broker. Use the below shell command to start a Broker. The shell script takes some Broker configurations, and we provide a config file with all the default values.

|  |  |
| --- | --- |
|  | bin/kafka-server-start.sh config/server.properties |

The server properties file is part of default Kafka installation, so you don't have to change anything, just use the default file.  
Once you execute the above command, you will see a lot of messages coming. Hopefully, you will see a message like started or start up complete. If you don't see such message, scroll a little up and try to find it. We just need to make sure that there are no errors and the Broker started successfully. In my case, it's started.

**Create Kafka Topic**

The next step is to create a Kafka Topic. In fact, you can skip this step because, by default, Kafka will create a topic automatically. So, when a producer sends data to a non-existent topic, Kafka will create the Topic and accepts the message. But let's create the Topic using the Topic Management tool.

|  |  |
| --- | --- |
|  | bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic MyFirstTopic1 --partitions 1 --replication-factor 1 |

The first parameter is zookeeper address and port. The next item is the create command. The topic management tool provides many functionalities. In this example, we are using it to create a Topic, so we give the create command. The next parameter is to provide a Topic name. So, I name it as MyFirstTopic1. Other two parameters are to give the number of partitions and replication factor. Let's ignore replication factor for the time being. We will cover replication factor in next video. You already know Partitions. I am creating two partitions on this topic. You might be wondering, that I have a single broker then how come we create two partitions. That's not a problem. Kafka will try to distribute partitions evenly over the available Brokers. But in our case, we just have a single Broker, so Kafka doesn't have any option other than creating both partitions on the same machine.

Start Kafka Producer and Consumer

Now, we have the last thing. Create a console producer in one terminal. Create a console consumer in another terminal. Then we will send some messages from the producer. They should appear at the Consumer.  
**So, let's start the producer using the command listed below.**

|  |  |
| --- | --- |
|  | bin/kafka-console-producer.sh --broker-list localhost:9092 --topic MyFirstTopic1 |

To send a message to Kafka, you need a Broker address. That's what the first parameter specifies. We have a Broker running on the localhost and default port 9092. You submit the above command, and a producer starts running. It is a console Producer, so whatever we type on the console, it will send that to the broker.  
Before we start sending some messages, open another terminal and start a consumer. Use **the below command to start a console Consumer.**

|  |  |
| --- | --- |
|  | bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic MyFirstTopic1 |

All the parameters are same as we used for Producer. The parameter bootstrap-server has the same meaning as broker list.  
  
Great, you should have the consumer as well as a producer running in two different terminals. You can send some text from the producer by typing messages at the console. You should see those messages arriving at the consumer terminal.  
The console producer and console consumer are of no use other than simple demonstrations and testing the functionality. In a real-life project, you must use Kafka Producer APIs and Consumer APIs and code your producers and consumers according to your requirements.

**Apache Kafka Foundation Course - Fault Tolerance**

Welcome to Apache Kafka tutorial at Learning journal. In this session, we will cover fault tolerance in Apache Kafka.  
Do you understand the term, the fault tolerance?

What is fault tolerance?

In the previous session, we learned that Kafka is a distributed system and it works on a cluster of computers. Most of the time, Kafka will spread your data in partitions over various systems in the cluster. So, if one or two systems in a cluster fail, what will happen to your data? Will you be able to read it?  
Probably not. That's a fault. Can we tolerate it?  
The term fault tolerance is very common in distributed systems. It means, making your data available even in the case of some failures.  
How to do it?  
One simple solution is to make multiple copies of the data and keep it on separate systems. So if you have three copies of a partition, and Kafka stores them on three different machines, you should be able to avoid two failures. Since you have three copies on three different systems, even if two of them fails, you can still read your data from the third system.

**Replication Factor**

There is a particular term used for making multiple copies. We call it replication factor. So, if I say, replication factor is three, that means, I am maintaining three copies of my partition. If I say replication factor is two, that means we are keeping two copies of a partition. The replication factor of three is a reasonable number. You can even set it to higher if your data is supercritical or you are using cheap machines.  
So, Kafka implements fault tolerance by applying replication to the partitions. We can define replication factor at the Topic level. We don't set a replication factor of partitions, we set it for a Topic, and it applies to all partitions within the Topic.

Understanding Kafka Replication

You may want to understand how it works in Kafka. I mean, How Kafka make these copies? Let me explain that as well.  
Kafka implements a leader and follower model.  
So, for every partition, One Broker is elected as a leader. And the Leader takes care of all client interactions. What does that mean? That means when a producer is willing to send some data. It connects to the Leader and starts sending data. It is Leader's responsibility to receive the message, store it in local disk and send back an acknowledgment to the producer.  
Similarly, when a consumer is willing to read data, it sends a request to the leader. It is leader's responsibility to send requested data back to the consumer.

For every partition, we have a leader, and the leader takes care of all requests and responses. I hope that part is clear.  
You may be wondering, that in all the above explanation, we haven't made any copy. That's where the followers come into play. So, if we create a topic with the replication factor set to three, A leader of the Topic is already maintaining the first copy. We need two more copies. So, Kafka will identify two more brokers as followers to make two copies. These Followers will copy the data from a leader. They don't talk to producer or consumer. They just copy data from a Leader. Simple, isn't it?  
Can we see all this happening? Yes, let me show you a leader and followers in action so that you get a better understanding of these concepts.

**Multi-node Kafka Cluster**

To be able to demonstrate one Leader and two Followers, I need a three-node Kafka cluster. In an ideal Cluster, we install one Broker on one computer. But for a demonstration or a development activity, we can start multiple Brokers on a single machine. So let's do it.  
I already started a Kafka cluster with my first Broker in an earlier video. Now, I am going to start two more brokers on the same machine. I hope you remember the command line tool that we used to start a Kafka broker. We used kafka-server-start.sh, and we passed the server-properties file as a parameter to the tool. We will follow the same method to start two more brokers. But before that, we will make a copy of the Broker config file and modify it. That's necessary to start new Brokers. We can't start multiple Brokers using same properties.  
So, let's make two more copies of the original properties file and modify them. Then we will use these modified files to start two more Brokers.

|  |  |
| --- | --- |
|  | cp config/server.properties config/server-1.properties |
|  | cp config/server.properties config/server-2.properties |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT12.sh)[KT12.sh](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt12-sh) hosted with ❤ by [GitHub](https://github.com/)

After executing the above commands, you should have two more property files. Now, I want to change three properties in these files. Let me explain those properties, and then, we will go ahead and modify them.

Broker id - The first property is Broker id. It's a unique identifier for the Broker. The default values for the first broker is zero, so we will change it to 1 for the second broker, and 2 for the third Broker. This change is to provide a unique identification to each broker.

Broker port - The next property is the Broker port. It's a network port number to which Broker will bind itself. The Broker will use this port number to communicate with producers and consumers. We will just increment it to whatever the default value is there. In fact, when you start brokers on separate systems, you don't need to change this port number, but since we are starting them on a single machine, we need to change it. Otherwise, all brokers will start reading and writing on the same port number.

Broker log directory - The third property is the Broker log directory. The Broker log directory is the main data directory of a Broker. We don't want all of the brokers to write into the same directory, so we need to change this value as well.

Great, go ahead and modify these properties in the files and prepare a new file for other two brokers.  
You can start two more Brokers using those two new property files.

|  |  |
| --- | --- |
|  | bin/kafka-server-start.sh config/server-1.properties |
|  | bin/kafka-server-start.sh config/server-2.properties |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT13.sh)[KT13.sh](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt13-sh) hosted with ❤ by [GitHub](https://github.com/)

Great, you should have a three node Kafka cluster running.  
We did all of this because I wanted to create a topic with a replication factor three, and show you the leader and the follower for each partition. So, let's do that.

Kafka Topic Leader and Follower

You already know how to create a topic. Create a new topic with replication factor 3. We also **make sure that we have atleast two partitions.**

|  |  |
| --- | --- |
|  | bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic TestTopicXYZ --partitions 2 --replication-factor 3 |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT14.sh)[KT14.sh](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt14-sh) hosted with ❤ by [GitHub](https://github.com/)

The kafka-topics.sh is a great tool to manage a Kafka Topic. We call it Topic management tool. This tool also provides a describe command.

|  |  |
| --- | --- |
|  | bin/kafka-topics.sh --zookeeper localhost:2181 --describe --topic TestTopicXYZ |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT15.sh)[KT15.sh](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt15-sh) hosted with ❤ by [GitHub](https://github.com/)

The output of the above command shows you the topic name. Then it tells you the number of partitions in that Topic. It shows replication factor for the Topic. It tells everything that you want to know about a Topic. Since we have two partitions on this Topic, it displays two rows, one for each partition. The video explains all this visually.  
The first item is the partition id, and the id for the first partition in our example is 0, and it is 1 for the second partition.  
The next information is the leader. So, for the first partition, Broker 1 is the leader. What does that mean? That means that the Broker 1 will store and maintain the first copy of this partition and it will also fulfil all client requests for this partition. Similarly, the Broker 2 is the leader of the second partition.  
Let's come to the next information. The next column shows the list of the replicas. For the first partition, you will see three copies, Broker 1 maintains the first copy, and that one is the leader also. Broker 2 manages second copy, and Broker 0 holds the third copy. The Broker 2 and Broker 0 are the followers.

What is the ISR?

The ISR is a list of In Sync Replicas. You might have three copies, but one of them may not be in sync with the leader. So, The ISR shows the list of replicas that are in sync with the Leader. In our case, all three are in sync.  
  
Good. So that's it for this session. We covered replication and fault tolerance in this video. We also learned to start three Brokers on a single machine.

Apache Kafka Foundation Course - Broker Configurations

Welcome to Apache Kafka tutorial at Learning journal. In the previous video, we started a multi-node cluster on a single machine. We have also seen some configuration parameters like broker id, port number, and log dirs. In this session, we will look at some more configurations.  
Apache Kafka is a highly configurable system. It provides many configurable parameters. Most of them have a reasonable default, so you don't need to worry about all of them. In this session, I will cover some key broker configurations. We can't cover everything, so I have selected some critical parameters for our discussion. But I recommend that you check the documentation for the complete list of [Kafka configuration parameters](https://kafka.apache.org/documentation/#_blank). Reading them at least once will give you some idea about available configurations. The overview of the available options will help you customize Kafka for your use case. Here is the list of parameters that we will discuss in this session.

broker.id

port

log.dirs

zookeeper.connect

delete.topic.enable

auto.create.topics.enable

default.replication.factor

num.partitions

log.retention.ms

log.retention.bytes

I am skipping first three because we have already covered them in previous session.

Zookeeper.connect

This parameter takes zookeeper connection string. The connection string is simply a hostname with a port number.

We already know that Kafka uses zookeeper for various coordination purposes, so it is critical that every broker knows the zookeeper address. This parameter is also necessary to form a cluster.  
**What do I mean by forming a Cluster?**  
Well, all brokers are running on different systems, how do they know about each other. If they don't know each other, they are not part of the cluster. So, the zookeeper is the connecting link among all brokers to form a cluster.

**delete.topic.enable**

If you want to delete a topic, you can use topic management tool. But by default, deleting a topic is not allowed. You can't remove a topic because the default value for this parameter is false. That is reasonable protection for production environments. But in development or testing environment, you may want to delete topics. So, if you want Kafka to allow deleting a topic, you need to set this parameter to true.

**auto.create.topics.enable**

We have already discussed auto-create topic feature. If a producer starts sending messages to a non-existent topic, Kafka will create the topic automatically and accept the data. This behaviour is suitable for dev environments. But in a production environment, you may want to implement a more controlled approach. You can set this parameter to false, and Kafka will stop creating topics automatically. You can create topics manually using the topic management tool, and no one will be able to send data to a non-existent topic.

**default.replication.factor and num.partitions**

These two parameters are quite straightforward. The default values for both of them is one, and they are effective when you have auto create topics enabled. So, if Kafka is creating your topic automatically, the new topic will have only one partition and a single copy. If you want some other values, you can change the default settings accordingly.

**log.retention.ms and log.retention.bytes**

These two are critical and not obvious. So, whatever data you send to Kafka, it is not retained by Kafka forever. Kafka is not a database. You don't send data to Kafka for storage so that you can query it later. It is a message broker. It should deliver the data to the consumer and then clean it up. There is no reason to retain messages for longer than needed.  
Kafka gives you two options to configure the retention period. The default option is retention by time, and the default retention period is seven days. So, in this case, Kafka will clean up all the messages older than seven days. If you want to change the duration, you can specify your value for log.retention.ms configuration.  
Kafka gives you another option to define this retention period. You can specify it by size. That's where the second parameter log retention bytes is applicable. But this size applies to partition. So, if you set log.retention.bytes = 1 GB, Kafka will trigger a clean-up activity when the partition size reaches to 1 GB. Remember that it is not a topic size. It is partition size.  
If you specify both configurations, the clean-up will start on meeting either of the criteria.  
That's it for this session. I believe you are ready to start writing some code, so in next session, we will create a custom producer and send some data to Kafka.

Apache Kafka Foundation Course - How to compile Kafka Code

Welcome back to Apache Kafka Tutorial. This video is an update for Apache Kafka 1.x. In all my tutorial videos, I am using SBT to compile and execute my examples. Many of my viewers are new to SBT. SBT is Scala build tool. However, It is equally useful for Java as well. I keep getting complains from many of my viewers about SBT installation.

Installing SBT for Kafka

In this video, I will show you step by step process to install, compile and execute one of my examples in GCP Kafka VM. So, let’s start.

SSH to your Kafka VM.

Check if you already have a Java compiler. Just type javac on the command prompt. You might see a message as command not found. You can fix this problem by installing open JDK.

Use below command to install JDK.

|  |  |
| --- | --- |
|  | sudo apt-get install openjdk-8-jdk |

Next step is to install SBT. Start your browser and visit [Scala SBT home page](https://www.scala-sbt.org/#_blank)

Navigate to SBT documentation and visit latest SBT documentation.

Look for installing SBT on Linux. Your GCP Kafka machine is using Debian OS. So, we follow the installation procedure for Debian.

First step is to setup SBT repository.

|  |  |
| --- | --- |
|  | echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.list.d/sbt.list |

Update the apt-get, and finally install SBT.

|  |  |
| --- | --- |
|  | sudo apt-get update |
|  | sudo apt-get install sbt |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT18.sh)[KT18.sh](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt18-sh) hosted with ❤ by [GitHub](https://github.com/)

That's it. You are ready to start SBT for the first time. Create a working directory, change your working directory and start SBT by typing sbt command

Compile and Execute Kafka Producer

Now let's compile and execute our first example. Follow these steps.

Create a build.sbt file, and place following content in the file.

|  |  |
| --- | --- |
|  | name := "KafkaTest" |
|  |  |
|  | scalaVersion := "2.11.8" |
|  |  |
|  | libraryDependencies ++= Seq( |
|  | "org.apache.kafka" % "kafka-clients" % "1.0.0") |

Create a source code file and copy the below code in the file.

|  |  |
| --- | --- |
|  | //File Name-SimpleProducer.java |
|  |  |
|  | import java.util.\*; |
|  |  |
|  | import org.apache.kafka.clients.producer.\*; |
|  |  |
|  | public class SimpleProducer { |
|  |  |
|  | public static void main(String[] args) throws Exception { |
|  |  |
|  | String topicName = "SimpleProducerTopic"; |
|  | String key = "Key1"; |
|  | String value = "Value-1"; |
|  |  |
|  | Properties props = new Properties(); |
|  | props.put("bootstrap.servers", "localhost:9092,localhost:9093"); |
|  | props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer"); |
|  | props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer"); |
|  |  |
|  | Producer<String, String> producer = new KafkaProducer<>(props); |
|  |  |
|  | ProducerRecord<String, String> record = new ProducerRecord<>(topicName, key, value); |
|  | producer.send(record); |
|  | producer.close(); |
|  |  |
|  | System.out.println("SimpleProducer Completed."); |
|  | } |
|  | } |

The source code is same as we used in one of my tutorial video. The code is also available in [Learning Journal GitHub repository](https://github.com/LearningJournal/ApacheKafkaTutorials#_blank).

You are now ready to compile our simple producer example. Execute the command as shown below to compile your program.

|  |  |
| --- | --- |
|  | sbt compile |

You should see a success message in the end. You can execute your program using sbt run command.

Great. If you want to see the message, you can start a console consumer. Use below command to execute the consumer.

|  |  |
| --- | --- |
|  | kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic SimpleProducerTopic --from-beginning |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT22.sh)[KT22.sh](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt22-sh) hosted with ❤ by [GitHub](https://github.com/)

We sent just one message, so your consumer shows a single message.  
Great. That's it for this session.  
Thank you for watching learning journal. Keep learning and keep growing.

Apache Kafka Foundation Course - Producer API

Welcome to Apache Kafka tutorial at Learning journal. So far, we covered a lot of theory and concepts of Apache Kafka. But Kafka is more about APIs. You can classify Kafka APIs into two parts.

Producer APIs

Consumer APIs.

In this session, we will cover internals of Producer API and also create an example producer. By the end of this video, you will have a sound understanding of Apache Kafka producer API, and you should be able to code your producers. So let's start.

Asynchronous Communication

I think, by now, you already know that we can use Kafka in several ways. We can use Kafka to solve complex data integration problems. We can use Kafka to create a series of validations, transformations and build complex data pipelines. We can use it to record information for later consumption, for example, recording clickstream. We can use it to log transactions and create applications to respond in real-time. We can also use it to collect data from your mobile phones, smart appliances, and sensors in an IOT application.  
But if you look at any of these use cases, it's all about asynchronous communication among applications. So, whatever we do with Kafka, we must have a producer that will send data to Kafka. You need to create a producer for your application to send data to Kafka. The most common method to create Kafka producer is using Kafka APIs. Since core APIs are available in Java, you must know Java to be able to understand and use them. However, even if you are not in day to day Java coding, you can still understand the concepts and internal working of Kafka during this discussion.

A Simple Kafka Producer

Let's directly jump into some code. I have listed the code for the simplest Kafka producer. Just take a look at it before we jump into an explanation.  
Git Filename - [SimpleProducer.java](https://github.com/LearningJournal/ApacheKafkaTutorials/blob/master/ProducerExamples/SimpleProducer.java#_blank)

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  |  |
|  | import org.apache.kafka.clients.producer.\*; |
|  |  |
|  | public class SimpleProducer { |
|  | public static void main(String[] args) throws Exception { |
|  |  |
|  | String key = "Key1"; |
|  | String value = "Value-1"; |
|  | String topicName = "SimpleProducerTopic"; |
|  |  |
|  | Properties props = new Properties(); |
|  | props.put("bootstrap.servers", "localhost:9092,localhost:9093"); |
|  | props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer"); |
|  | props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer"); |
|  |  |
|  | Producer<String, String> producer = new KafkaProducer<>(props); |
|  | ProducerRecord<String, String> record = new ProducerRecord<>(topicName, key, value); |
|  | producer.send(record); |
|  | producer.close(); |
|  | System.out.println("SimpleProducer Completed."); |
|  | } |
|  | } |

At line 1, the [org.apache.kafka.clients.producer](https://github.com/apache/kafka/tree/trunk/clients/src/main/java/org/apache/kafka/clients/producer) package defines Apache Kafka producer API. If you are interested in looking at the source code for the package, it's available on GitHub. You may also need to refer [producer API documentation](https://kafka.apache.org/10/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer.html#_blank)

In the above example, we want to send a string message to Kafka. Most of the time, Kafka messages are Key value pair. So, with every message, you can send a key. However, the key is not mandatory. You can send a message without a key. But in this example, at line 2, we want to create a key as Key1 and a value as Value-1. And we want to send it to the topic SimpleProducerTopic.  
If you look at the rest of the code, there are only three steps.

Step 1 - Create a KafkaProducer object - line 4.

Step 2 - Create a ProducerRecord object - line 5.

Step 3 - Send the record to the producer - line 6.

That is all that we do in a Kafka Producer.  
So, let’s look at the details of each step.  
The first step is to create a KafkaProducer object. To create this object, you need a property object with at least three mandatory configurations (line 3).  
These core configurations are.

bootstrap.servers

key.serializer

value.serializer

I hope you already know bootstrap servers. It is a list of Kafka brokers. The producer object will use this list to connect to Kafka cluster. You can specify one or more brokers in this list. The recommendation is to provide at least two brokers, so if one broker is down, the producer can connect to the other broker from the list.  
The next two properties are about Kafka message. I already mentioned in an earlier video that Kafka doesn't have any meaning for the data. A message is just an array of bytes for Kafka. In this example, we are sending a string Key and a string value. But Kafka accepts only an array of bytes. So, we need a class to convert our message key and value into an array of bytes. The activity of converting Java objects into an array of bytes is called serialization. So those two properties are to specify an appropriate serializer class for the key and value. In this example, we are using string serializer for both key and value. Kafka also provides some other serializers like IntSerializer or DoubleSerializer. If you want to send an integer key, you should use an IntSerializer for the key instead of StringSerializer. We will cover serializers in detail in another session.  
So, we define those three mandatory configurations and package them into a Java properties object. Then we pass the properties object to KafkaProducer object constructor and instantiate a producer. That was our first step.  
We are done with the first step, and now you have a Producer instantiated. We want this Producer to send some messages. The second step is to create a ProducerRecord object. The ProducerRecord object requires three things, The Topic name, Key, and the Message Value. We are passing all these three things into the constructor and instantiating a ProducerRecord. The producer record object is our message. It should be given to producer, so the producer can send it to Kafka broker.  
As a Final step, we make a call to send method on Producer object and handover the RecordObject. That's it.  
Now it's producer’s responsibility to deliver this message to the broker. After sending all your messages, you need to close the Producer object. Closing a producer is necessary to clean up all resources that producer may be using.  
In a real-life project, the Producer is a long-running process that keeps sending messages, but we took a simple example to understand the API.  
That's it for the simple producer, you can compile it and execute it. You can use console consumer to test if the message reached the Broker.

How to Compile Kafka Producer

If you want to compile and execute the given code, you can use SBT to compile and execute it as demonstrated in the video. If you are new to SBT, I have a video training on SBT as part of my Scala training. You will need a Scala build file to compile the above example. You can use the build file as given below.  
Filename – [build.sbt](https://github.com/LearningJournal/ApacheKafkaTutorials/blob/master/ProducerExamples/build.sbt)

|  |  |
| --- | --- |
|  | name := "KafkaTest" |
|  |  |
|  | libraryDependencies ++= Seq( |
|  | "org.apache.kafka" % "kafka-clients" % "0.10.1.0" |
|  | exclude("javax.jms", "jms") |
|  | exclude("com.sun.jdmk", "jmxtools") |
|  | exclude("com.sun.jmx", "jmxri") |
|  | exclude("org.slf4j", "slf4j-simple") |
|  | ) |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT24.build)[KT24.build](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt24-build) hosted with ❤ by [GitHub](https://github.com/)

In the above simple example, we created two objects.

KafkaProduce

ProducerRecord

I recommend you look at the Java Docs for these two classes.

The KafkaProducer

The KafkaProducer give you four variants of a constructor. They don't provide anything new but only offer different methods to pass in your configuration values. If you look at the signature of send method in the documentation, it returns you a Java Future object of type RecordMetadata. The RecordMetadata is a kind of acknowledgment from the broker. It contains information about your message, for example, the partition id and offset number. We will cover more details on acknowledgment in the next session.

The ProducerRecord

The ProducerRecord offers four variants of the constructor. We used one of the options and supplied the topic, message key, and message value. But there is another constructor with a more comprehensive option. It allows us to provide two more parameters.

Partition Number

Timestamp.

We will cover Kafka message partition in a separate session but let me cover some basics here.  
Kafka comes with a default partitioner. This partitioner will use the message key to determine a partition number. If you are sending few thousand messages with same message key, they all will land in the same partition. But you know that message key is optional, so if you don't send a message key, the default partitioner will evenly distribute those messages across the available partitions.  
But what about the partition number parameter in the ProducerRecord ?  
If you set a partition number in your ProducerRecord, you will disable default partitioner. It is like, you are hardcoding a partition in your message itself and don't want Kafka to determine a suitable partition for your message. If you hard code it to zero, your message will go to partition zero. We will cover some usage of this parameter in later sessions.  
Now let's come to Timestamp. Kafka gives a timestamp to every message. If you want to set a message time stamp before you send it to Kafka, you can use this parameter. If you don't set a timestamp, the broker will set it when the message reaches the broker. It is important to note the difference. The former is the time when you are sending a message to the broker, and later one is the time when the broker is receiving a message.  
That's it for this session. In next session, we will cover some more details of Kafka producer APIs.

Apache Kafka Foundation Course - Producer Workflow

Welcome to Apache Kafka tutorial at Learning journal. In the previous session, we created a Kafka producer. In this session, we will look at the internals of a Kafka producer. We will look at what is going on under the hood. We will try to understand that how a message goes from a client application to a Broker. So, let's get started.

[Diagram

Description automatically generated](https://www.learningjournal.guru/_resources/img/jpg-7x/kafka-producer-workflow.jpg)Fig.1-Kafka Producer Workflow

Producer Configurations

The first step is to create a Java properties object and package all the producer configurations that we want to set. These settings include three mandatory configurations that we learned in the previous session.

bootstrap.servers

key.serializer

value.serializer

You can also set some additional properties or even custom configs. In the example that we created earlier, we used only three basic configs, but I will create some more examples in next session with other properties including custom configs.

Producer Record

On the other side, we create a producer record and package five things in a ProducerRecord object. Those five things are listed below.

Topic Name

Partition Number

Timestamp

Key

Value

The partition number, timestamp, and key are optional depending upon your use case. The ProducerRecord object is, in fact, the message that we want to send to Kafka Broker.  
Once we have the Properties and the ProducerRecord definition, we instantiate a Producer object using the Properties object. Then we send the ProducerRecord to the producer object. That’s it. The message is handed over to the producer. When the message is handed over to the producer, following things happen.

Serialization

The Producer will apply the serializer to serialize your Key and Value. That's the first thing. You already know that serialization is converting your Key and Value objects into a byte array, and the producer will use the serializer class that we specified to accomplish this.

Partitioning

Then, it will send the record to the partitioner. The partitioner will choose a partition for the message. We already discussed earlier that the default partitioner would use your message key to determine an appropriate partition. If a message key is specified, Kafka will hash the key for getting a partition number. If you specify the same key to multiple messages, all of them will go to the same partition.  
If message key is not specified, the default partitioner will try to evenly distribute the messages to all available partitions for the topic. It uses a round robin algorithm, so few messages go to the first partition, then some of them goes to second and so on.

Partition Buffer

Once we have a partition number, the partitioner is ready to send it to Broker. But instead of sending the message immediately, the partitioner will keep the message into a partition buffer. The producer maintains an in-memory buffers for each partition and sends the records in batches. You might be wondering that what is the size of the batch? How much time the producer will linger waiting for more messages to arrive? We can configure all those things by adding appropriate configuration parameters to the properties object. I will cover them in the upcoming sessions.

Record Metadata and Retires

Finally, the producer will send a batch of records to the broker. If the broker can receive and save the message, it will send an acknowledgment in the form of RecordMetadata object. If anything goes wrong, the producer receives an error. Some errors may be recoverable with a retry, for example, suppose the leader of the partition was down, if we retry sending the batch in few milliseconds, we may have a new leader elected by that time. So, In the case of a recoverable error, the producer will retry sending the batch before it throws and exception.  
We can configure the number of retries and time between two retires. The producer will not attempt for a retry if the error is not a recoverable error.  
Great, the workflow of a producer is quite simple, and we can configure almost everything using the producer configuration parameters.  
That's it for this session. In next session, I will cover some more details of Kafka producer APIs, See you again.

Apache Kafka Foundation Course - Callback and Acks

Welcome to Apache Kafka tutorial at Learning Journal. We already created a simple producer and discussed how a message flows from producer to broker. In this session, I will discuss different approaches to implement a Kafka Producer. We will also learn about acknowledgment and call back method to handle responses from brokers. So, let's get started.  
There are three approaches to send a message to Kafka.

Fire-and-forget Producer

Send and forget is the simplest approach. In this method, we send a message to the broker and don’t care if it was successfully received or not. The example that we created earlier followed this approach.  
You might be wondering that is this a right approach? Where to use that method?  
Well, Kafka is a distributed system. It comes with inbuilt fault tolerance feature. That makes Kafka a highly available system. So most of the time, your message will reach to the broker. We also know that producer will automatically retry in case of recoverable error. So, the probability of losing your messages is thin.  
There are many use cases where you are dealing with huge volumes and losing a small portion of records is not critical. For example, if you are counting hits on a video, or collecting a twitter feed for sentiment analysis. In such use cases, even if you lose 2-3% of your tweets, it may not be a problem.  
But, it is important to understand that in fire and forget approach, you may lose some messages. So, don't use this method when you can't afford to lose your messages.

Synchronous Producer

In this approach, we send a message and wait until we get a response. In the case of a success, we get a RecordMetadata object, and in the event of failure, we get an exception. Most of the time, we don't care about the success and the RecordMetadata that we receive. We only care about exception because we want to log errors for later analysis and appropriate action. You can adopt this method if your messages are critical and you can't afford to lose anything.  
But it is important to notice that synchronous approach will slow you down. It will limit your throughput because you are waiting for every message to get acknowledged. You are sending one message and waiting for success, then you send your next message, and again wait for success. Each message will take some time to be delivered over the network. So, after every message, you will be waiting for a network delay, and the most interesting thing is that, you may not be doing anything in case of success. You care only failures, if it fails, you may want to take some action.  
We have already seen an example of fire and forget approach. Let's look at the example of synchronous send.

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  |  |
|  | import org.apache.kafka.clients.producer.\*; |
|  |  |
|  | public class SynchronousProducer { |
|  |  |
|  | public static void main(String[] args) throws Exception { |
|  |  |
|  | String topicName = "SynchronousProducerTopic"; |
|  | String key = "Key-1"; |
|  | String value = "Value-1"; |
|  |  |
|  | Properties props = new Properties(); |
|  | props.put("bootstrap.servers", "localhost:9092,localhost:9093"); |
|  | props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer"); |
|  | props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer"); |
|  |  |
|  | Producer<String, String> producer = new KafkaProducer<>(props); |
|  | ProducerRecord<String, String> record = new ProducerRecord<>(topicName, key, value); |
|  |  |
|  | try { |
|  | RecordMetadata metadata = producer.send(record).get(); |
|  | System.out.println("Message is sent to Partition no " + metadata.partition() + " and offset " + metadata.offset()); |
|  | System.out.println("SynchronousProducer Completed with success."); |
|  | } catch (Exception e) { |
|  | e.printStackTrace(); |
|  | System.out.println("SynchronousProducer failed with an exception"); |
|  | } finally { |
|  | producer.close(); |
|  | } |
|  | } |
|  | } |

The code is almost same as the one we looked earlier. We are following the same three steps that I explained earlier.

Create producer properties and instantiate producer object.

Create producer record.

And finally, hand over the message to the producer by making a call to send method.

But this time, we want to get a response and handle an exception, so we wrap it in a try-catch and finally blocks. If you look at the try block, we are still calling producer.send method.  
This send method returns a Java Future, and we call a get method on the Future object. The get method will wait till we get success or an exception.  
In the case of success, we get a record metadata object. I am printing partition number and offset number from the RecordMetadata. But in your real application, you may not be doing anything for success.  
In the case of an exception, we are just printing out the failure text and stack trace, but in your production code, you may want to log the message and exception details for later analysis. We have a final block as well, and we are closing the producer object to free up resources.  
The fire and forget approach was on one extreme, and the synchronous approach is at another extreme. I mean, in one approach, you don't care at all, and in another method, you wait for every single message. So, there is a third approach which takes the middle path.

Asynchronous Producer

In this method, we send a message and provide a call back function to receive acknowledgment. We don't wait for success and failure. The producer will callback our function with RecordMetadata and an exception object. So, if you just care about exceptions, you simply look at the exception, if it is null, don't do anything. If the exception is not null, then you know it failed, so you can record the message details for later analysis.  
Let's look at an example of the asynchronous approach.

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  |  |
|  | import org.apache.kafka.clients.producer.\*; |
|  |  |
|  | public class AsynchronousProducer { |
|  |  |
|  | public static void main(String[] args) throws Exception { |
|  | String topicName = "AsynchronousProducerTopic"; |
|  | String key = "Key1"; |
|  | String value = "Value-1"; |
|  |  |
|  | Properties props = new Properties(); |
|  | props.put("bootstrap.servers", "localhost:9092,localhost:9093"); |
|  | props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer"); |
|  | props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer"); |
|  |  |
|  | Producer<String, String> producer = new KafkaProducer<>(props); |
|  | ProducerRecord<String, String> record = new ProducerRecord<>(topicName, key, value); |
|  |  |
|  | producer.send(record, new MyProducerCallback()); |
|  | System.out.println("AsynchronousProducer call completed"); |
|  | producer.close(); |
|  |  |
|  | } |
|  | } |
|  |  |
|  | class MyProducerCallback implements Callback { |
|  |  |
|  | @Override |
|  | public void onCompletion(RecordMetadata recordMetadata, Exception e) { |
|  | if (e != null) |
|  | System.out.println("AsynchronousProducer failed with an exception"); |
|  | else |
|  | System.out.println("AsynchronousProducer call Success:"); |
|  | } |
|  | } |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT26.java)[KT26.java](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt26-java) hosted with ❤ by [GitHub](https://github.com/)

The code for asynchronous producer is again same as send and forget approach. The only difference is that we have a second parameter for the send method. A second parameter is a callback object.  
The next part of the code is the callback class. So, if you want to create a callback class, you must implement the Callback interface. Then you have just one method to override. You need to override the onCompletion method. The broker will invoke this method with an acknowledgment or an exception. The rest of the code is simple, If the exception object is not null, we have a failure else we have success.  
In this approach, you keep sending messages as fast as you can without waiting for responses, and handle failures later as they come using a callback function.

In Flight Messages

The asynchronous method appears to provide you a throughput that is as good as fire and forget approach. But there is a catch. You have a limit of in-flight messages. This limit is controlled by a configuration parameter max.in.flight.requests.per.connection. This parameter controls that how many messages you can send to the server without receiving a response. The default value is 5. You can increase this value, but there are other considerations. We will cover producer configurations in a separate session. Till then, just understand that asynchronous send gives you a better throughput compared to synchronous, but the max.in.flight.requests.per.connection limits it.  
That's it for this session. In next session, we will cover some more details of Kafka producer APIs, See you again.  
Thanks for watching Learning Journal.  
Keep learning and Keep growing.

Apache Kafka Foundation Course - Custom Partitioner

Welcome to Apache Kafka tutorial at Learning Journal. In this session, we are going to explore Kafka partitioner. We will try to understand why default partitioner is not enough and when you might need a custom partitioner. We will also look at a use case and create code for custom partitioner. I already explained that Kafka partitioner is responsible for deciding partition number for each message. We also discussed the behaviour of the default partitioner. Let me quickly recap the default partitioner.  
The default partitioner follows these rules.

If a producer specifies a partition number in the message record, use it.

If the message doesn’t hard code a partition number, but it provides a key, choose a partition based on a hash value of the key.

If no partition number or key is present, pick a partition in a round-robin fashion.

So, you can use default partitioner in three scenarios.

You don't care about which partition your data is landing, but you want the partitioner to distribute your data evenly, you will use the third rule of default partitioner.

If you already know which partition do you want to send the data, you will hard code it and use the first rule of default partitioner.

If you want your data to be distributed based on your Key, you will specify a key in your messages.

Partition Keys

But there is a catch with the key, and that is because the way hashing works. The hashing guarantees that a key will always give you the same hash number. But it doesn't ensure that two different keys will never give you the same hash number.  
So, for example, if you have three tables, and you want to send all rows from these three tables to three different partitions. I mean, data from table A goes to partition 0 and data from table B goes to partition 1 and so on. One of the obvious thought is to send table name as a key. But that will be incorrect because Table A and Table B can give the same number after hashing. I will show you this happening in the example.

Since hashing doesn’t guarantees a unique number, it is better that you manage the translation of the table name to a partition number in your producer and hard code your partition number with the message. Another alternative is to implement a custom partitioner and use your partitioner instead of using the default one. I will show you an example for custom partitioner as well.  
That's the first precaution with the key.  
There is another catch with the key. The partition number is the mod of the hash value of the key and the total number of partitions on the topic. So, if you are increasing the number of partitions for your topic, the default partitioner will start returning different numbers. That may be a problem if you are relying on your key for achieving a spaecifc partitioning.  
With these two problems, I don't find a Key to be of good use in making desired custom partitioning. If you want a specific type of partitioning, the only option is to create your own algorithm and implement it in a custom partitioner.

Kafka Partitioner Example

Let's create an example use-case and implement a custom partitioner.  
Assume, we are collecting data from a bunch of sensors. All the sensors are sending data to a single topic. I planned ten partitions for the topic. But I want three partitions dedicated for a specific sensor named TSS and remaining seven partitions for rest of the sensors. How would you achieve this?  
You can solve this requirement, and any other type of partitioning need by implementing a custom partitioner.  
Let me show you the solution code for this problem. The first thing is a producer. We created several producers earlier. I created a new one for this example. Let’s look at the producer code.

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  |  |
|  | import org.apache.kafka.clients.producer.\*; |
|  |  |
|  | public class SensorProducer { |
|  |  |
|  | public static void main(String[] args) throws Exception { |
|  |  |
|  | String topicName = "SensorTopic"; |
|  |  |
|  | Properties props = new Properties(); |
|  | props.put("bootstrap.servers", "localhost:9092,localhost:9093"); |
|  | props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer"); |
|  | props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer"); |
|  | props.put("partitioner.class", "SensorPartitioner"); |
|  | props.put("speed.sensor.name", "TSS"); |
|  |  |
|  | Producer<String, String> producer = new KafkaProducer<>(props); |
|  |  |
|  | for (int i = 0; i < 10; i++) |
|  | producer.send(new ProducerRecord<>(topicName, "SSP" + i, "500" + i)); |
|  |  |
|  | for (int i = 0; i < 10; i++) |
|  | producer.send(new ProducerRecord<>(topicName, "TSS", "500" + i)); |
|  |  |
|  | producer.close(); |
|  | System.out.println("SimpleProducer Completed."); |
|  | } |
|  | } |

The producer code is like other producers that I have already explained. The only difference is in the configuration properties. We are setting two new properties. The first one is partitioner.class property. Since we are not using default partitioner, we set this property to the class name of our custom partitioner. I will show you the code for this custom partitioner class in a minute.  
The next property speed.sensor.name is not a Kafka configuration. It is a custom property. We are using it to supply the name of the sensor that requires special treatment. I don't want to hardcode the string TSS in my custom partitioner, and custom configuration is the method of passing values to your partitioner. Rest of the code is straightforward. We send some messages for various sensors. Then we send some messages for TSS sensor.

Custom Kafka Partitioner

We need to create our class by implementing Partitioner Interface. Your custom partitioner class must implement three methods from the interface.

Configure

Partition

Close

Let’s look at the code.

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  |  |
|  | import org.apache.kafka.clients.producer.\*; |
|  | import org.apache.kafka.common.\*; |
|  | import org.apache.kafka.common.utils.\*; |
|  | import org.apache.kafka.common.record.\*; |
|  |  |
|  | public class SensorPartitioner implements Partitioner { |
|  |  |
|  | private String speedSensorName; |
|  |  |
|  | public void configure(Map<String, ?> configs) { |
|  | speedSensorName = configs.get("speed.sensor.name").toString(); |
|  | } |
|  |  |
|  | public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) { |
|  |  |
|  | List<PartitionInfo> partitions = cluster.partitionsForTopic(topic); |
|  | int numPartitions = partitions.size(); |
|  | int sp = (int) Math.abs(numPartitions \* 0.3); |
|  | int p = 0; |
|  |  |
|  | if ((keyBytes == null) || (!(key instanceof String))) |
|  | throw new InvalidRecordException("All messages must have sensor name as key"); |
|  |  |
|  | if (((String) key).equals(speedSensorName)) |
|  | p = Utils.toPositive(Utils.murmur2(valueBytes)) % sp; |
|  | else |
|  | p = Utils.toPositive(Utils.murmur2(keyBytes)) % (numPartitions - sp) + sp; |
|  |  |
|  | System.out.println("Key = " + (String) key + " Partition = " + p); |
|  | return p; |
|  | } |
|  |  |
|  | public void close() { |
|  | } |
|  | } |

The configure and the close methods are like initialization and clean-up methods. They will be called once at the time of instantiating your producer. You can initialise things in Configure method and clean up things in close method. In our example, we don't have anything to clean up. However, we have something to configure.  
We want to find the sensor name that requires three partitions. My producer is sending that name as a custom configuration. The configure method is extracting the configuration value and setting it into a private variable. I will use that variable later in the code.  
The partition method is the place where all the action happens. The producer will call this method for each message and provide all the details with every call. The input to the method is the topic name, key, value and the cluster details. With all these input parameters, we have everything that is required to calculate a partition number. All that we need to do is to return an integer as a partition number. This method is the place where we implement our algorithm for partitioning.  
Let’s try to understand the algorithm that I have implemented. I am applying my algorithm in four simple steps.

The first step is to determine the number of partitions and reserve 30% of it for TSS sensor. If I have ten partitions for the topic, this logic will reserve three partitions for TSS. The next question is, how do we get the number of partitions in the topic?  
We got a cluster object as an input, and the method partitionsForTopic will give us a list of all partitions. Then we take the size of the list. That's the number of partitions in the Topic. Then we set SP as 30% of the number of partitions. So, if I have ten partitions, SP should be set to 3.

If we don't get a message Key, throw an exception. We need the Key because the Key tells us the sensor name. Without knowing the sensor name, we can't decide that the message should go to one of the three reserved partitions or it should go to the the other bucket of seven partitions.

Next step is to determine the partition number. If the Key = TSS, then we hash the message value, divide it by 3 and take the mod as partition number. Using mod will make sure that we always get 0, 1 or 2.

If the Key != TSS then we divide it by 7 and again take the mod. The mod will be somewhere between 0 and 6. So, I am adding 3 to shift it by 3.

That all is pure maths, and I hope you get that. You might be wondering that in step 3, I am hashing message value but in step 4, I am hashing message key. Let me explain. In step 3, every time Key will be TSS. So, hashing TSS will give me same number every time, and all the TSS messages will go to the same partition. But we want to distribute it in first three partitions. So, I am hashing the message value to get a different number every time.  
In step 4, I should be hashing the message value again. However, instead of hashing message value again, I am hashing the Key. That's because I wanted to show you that why you should be careful if you want to use a Key for achieving a specific partitioning. I wanted to demonstrate that different Keys can land up in the same partition. We will observe that behaviour by executing this example. Checkout the video for the demonstration. The video shows you the compilation and execution process.  
Thanks for watching Learning Journal. Keep learning and Keep growing.

Apache Kafka Foundation Course - Custom Serializer

Welcome to Kafka tutorial at Learning Journal. In this lesson, we will discuss custom serializers.  
We already know that we need appropriate serializer for our keys and values. In all our earlier examples, we are sending strings and hence we have been using string serializers. Kafka also provides some other serializers, for example, int, double and long. But these serializers together with string serializer doesn't cover most of the use cases.

Why Custom Serializer

If you are coming from a database background, you can think of a topic like a table and each message sent to the topic like a record. Those records are not always just a single string or a number. Normally, we have multiple columns in a record. So, when working with Kafka, we need to be able to send a record of multiple columns.  
Similarly, if you are coming from an object-oriented programming background, you will see Kafka message as an object. And normally, these objects will have multiple fields and methods. We should be able to send these objects to Kafka as a message. Sending simple strings to Kafka may fulfil some requirements. But in a complex condition, you may need to send custom objects, for example, A supplier object or an invoice object. If you want to send such custom objects or a row like structure, you need to implement a custom serializer and a deserializer.  
But let me tell you that there are other better options. Best practice is to use generic serializers instead of creating custom serializer and deserializer. There are many generic serializers like Avro and protocol buffer. But to be able to understand how serializers work, we will look at one example. However, most of the time, you will be using Generic serializers like Avro. We will leave Avro for some other day and focus on custom serializer in this session.

Kafka Serializer Example

To understand the idea of serializer and deserializer, we need to create an example.In this example, we will do following things.

Create a Supplier class. We will serialize the supplier class and send the supplier object as a message to Kafka.

Create a Kafka producer. This producer will send supplier object as a Kafka record. Earlier we were sending strings, but in this example, we are going to push an object instead of a simple string.

Create a serializer to convert a supplier object into a byte array.

Create a deserializer to convert a byte array back into a supplier object. Kafka doesn't know how to serialize and deserialize our object, and so we must create a serializer and a deserializer.

Create a consumer. Finally, we will create a consumer that will read supplier objects from Kafka and just print the details on the console.

We will execute our example and observe all this working together. Let's start.

Create a Supplier Class

The first thing is the supplier class. Let's look at the code.

|  |  |
| --- | --- |
|  | import java.util.Date; |
|  |  |
|  | public class Supplier { |
|  | private int supplierId; |
|  | private String supplierName; |
|  | private Date supplierStartDate; |
|  |  |
|  | public Supplier(int id, String name, Date dt) { |
|  | this.supplierId = id; |
|  | this.supplierName = name; |
|  | this.supplierStartDate = dt; |
|  | } |
|  |  |
|  | public int getID() { |
|  | return supplierId; |
|  | } |
|  |  |
|  | public String getName() { |
|  | return supplierName; |
|  | } |
|  |  |
|  | public Date getStartDate() { |
|  | return supplierStartDate; |
|  | } |
|  | } |

The supplier class defines three variables.

Supplier id

Supplier name

Supplier date.

We also have one constructor and three methods. We will use this supplier class to instantiate supplier object and send it as a Kafka message. This code is very simple. The constructor takes three parameters and initializes the three variables. The three methods are to get the values for the corresponding variable.

Create a Kafka Serializer

Next thing is a serializer class. We already know that there is a string serializer. It is a good idea to copy the existing source code and modify it according to your requirement. So, I took the code from Kafka source repository and modified it for my example. So, let's look at the modified code.

|  |  |
| --- | --- |
|  | import org.apache.kafka.common.serialization.Serializer; |
|  | import org.apache.kafka.common.errors.SerializationException; |
|  |  |
|  | import java.io.UnsupportedEncodingException; |
|  | import java.util.Map; |
|  | import java.nio.ByteBuffer; |
|  |  |
|  | public class SupplierSerializer implements Serializer<Supplier> { |
|  | private String encoding = "UTF8"; |
|  |  |
|  | @Override |
|  | public void configure(Map<String, ?> configs, booleanisKey) { |
|  | // nothing to configure |
|  | } |
|  |  |
|  | @Override |
|  | public byte[] serialize(String topic, Supplier data) { |
|  | int sizeOfName; |
|  | int sizeOfDate; |
|  | byte[] serializedName; |
|  | byte[] serializedDate; |
|  |  |
|  | try { |
|  | if (data == null) |
|  | return null; |
|  |  |
|  | serializedName = data.getName().getBytes(encoding); |
|  | sizeOfName = serializedName.length; |
|  | serializedDate = data.getStartDate().toString().getBytes(encoding); |
|  | sizeOfDate = serializedDate.length; |
|  |  |
|  | ByteBuffer buf = ByteBuffer.allocate(4 + 4 + sizeOfName + 4 + sizeOfDate); |
|  | buf.putInt(data.getID()); |
|  | buf.putInt(sizeOfName); |
|  | buf.put(serializedName); |
|  | buf.putInt(sizeOfDate); |
|  | buf.put(serializedDate); |
|  |  |
|  | return buf.array(); |
|  |  |
|  | } catch (Exception e) { |
|  | throw new SerializationException("Error when serializing Supplier to byte[]"); |
|  | } |
|  | } |
|  |  |
|  | @Override |
|  | public void close() { |
|  | // nothing to do |
|  | } |
|  | } |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT30.java)[KT30.java](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt30-java) hosted with ❤ by [GitHub](https://github.com/)

The name for my class is SupplierSerializer. It implements Serializer interface and sets the generic type as Supplier. This interface is defined under Kafka common package. As per this interface, we need to override three methods.

Configure

Serialize

Close

If you remember the previous session where we implemented a custom partitioner, you can recall this pattern. We had a similar Interface for Partitioner. So, we already know that the configure and the close are for initialization and clean-up, and Kafka producer will call these methods only once. It will call configure when we instantiate the producer and call close when we close the producer. But in our example, we have nothing to do with configure and close methods. So, we leave them empty.  
The main action is happening in the serialize method. The code is straightforward. If the data is null, we return null because we have nothing to serialize.  
We simply convert supplier name and supplier start date into UTF8 bytes. Then we allocate a byte buffer and encode everything into the byte buffer. Since we will need to know the length of supplier name and supplier date strings at the time of deserialization, we also encode their sizes into the byte buffer. Finally, we return the byte buffer array. That's it. Done. That's what the serialization means, convert your object into bytes, and that's what we have done in the above example.

Create a Kafka Deserializer

Next part is a deserializer. Once you understand the serializer, the deserializer is simple. Let's look at the code for deserializer.

|  |  |
| --- | --- |
|  | import java.nio.ByteBuffer; |
|  | import java.util.Date; |
|  | import java.text.DateFormat; |
|  | import java.text.SimpleDateFormat; |
|  |  |
|  | import org.apache.kafka.common.errors.SerializationException; |
|  | import org.apache.kafka.common.serialization.Deserializer; |
|  |  |
|  | import java.io.UnsupportedEncodingException; |
|  | import java.util.Map; |
|  |  |
|  | public class SupplierDeserializer implements Deserializer<Supplier> { |
|  | private String encoding = "UTF8"; |
|  |  |
|  | @Override |
|  | public void configure(Map<String, ?> configs, boolean isKey) { |
|  | //Nothing to configure |
|  | } |
|  |  |
|  | @Override |
|  | public Supplier deserialize(String topic, byte[] data) { |
|  |  |
|  | try { |
|  | if (data == null) { |
|  | System.out.println("Null recieved at deserialize"); |
|  | return null; |
|  | } |
|  |  |
|  | ByteBuffer buf = ByteBuffer.wrap(data); |
|  | int id = buf.getInt(); |
|  |  |
|  | int sizeOfName = buf.getInt(); |
|  | byte[] nameBytes = new byte[sizeOfName]; |
|  | buf.get(nameBytes); |
|  |  |
|  | String deserializedName = new String(nameBytes, encoding); |
|  |  |
|  | int sizeOfDate = buf.getInt(); |
|  | byte[] dateBytes = new byte[sizeOfDate]; |
|  | buf.get(dateBytes); |
|  |  |
|  | String dateString = new String(dateBytes, encoding); |
|  | DateFormat df = new SimpleDateFormat("EEE MMM ddHH:mm:ss Z yyyy"); |
|  |  |
|  | return new Supplier(id, deserializedName, df.parse(dateString)); |
|  |  |
|  | } catch (Exception e) { |
|  | throw new SerializationException("Error when deserializing byte[] to Supplier"); |
|  | } |
|  | } |
|  |  |
|  | @Override |
|  | public void close() { |
|  | // nothing to do |
|  | } |
|  | } |

We are doing the opposite of what we did in the serializer. We deserialize every field, create a new supplier object and return it. That was all simple. That's what deserialization means, take a byte array and convert it into an object.

**Problems with Kafka Custom Serializer/Deserializer**

But the problem with this approach is managing future changes in the schema. Suppose you implemented this serializer and deserializer, and your system is functional for few months. After few months, you have a requirement to add another field in the supplier object. If you modify your Supplier object, you must change your serializer and deserializer, may be the producer and consumer as well. But the problem doesn't end there. After making new changes, you can't read your older messages because you changed the format and modified your code to read the new format. That's where generic serializers like Avro will be helpful. We will cover that part in some other session. For now, let’s continue our discussion on this example.

**Kafka Producer**

We completed supplier class, serializer, and deserializer class. Now, we need a producer to send messages and a consumer to receive messages. There is nothing new about a producer, we have already created several producers earlier.

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  | import java.text.DateFormat; |
|  | import java.text.SimpleDateFormat; |
|  |  |
|  | import org.apache.kafka.clients.producer.\*; |
|  |  |
|  | public class SupplierProducer { |
|  |  |
|  | public static void main(String[] args) throws Exception { |
|  |  |
|  | String topicName = "SupplierTopic"; |
|  |  |
|  | Properties props = new Properties(); |
|  | props.put("bootstrap.servers", "localhost:9092,localhost:9093"); |
|  | props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer"); |
|  | props.put("value.serializer", "SupplierSerializer"); |
|  |  |
|  | Producer<String, Supplier> producer = new KafkaProducer<>(props); |
|  |  |
|  | DateFormat df = new SimpleDateFormat("yyyy-MM-dd"); |
|  | Supplier sp1 = new Supplier(101, "Xyz Pvt Ltd.", df.parse("2016-04-01")); |
|  | Supplier sp2 = new Supplier(102, "Abc Pvt Ltd.", df.parse("2012-01-01")); |
|  |  |
|  | producer.send(new ProducerRecord<String, Supplier>(topicName, "SUP", sp1)).get(); |
|  | producer.send(new ProducerRecord<String, Supplier>(topicName, "SUP", sp2)).get(); |
|  |  |
|  | System.out.println("SupplierProducer Completed."); |
|  | producer.close(); |
|  |  |
|  | } |
|  | } |

The producer code in this tutorial is almost same as earlier examples. I changed the value serializer class name. I also changed the producer generics parameter. And finally, we are sending two messages using synchronous send.

Kafka Consumer

Now we need a consumer. We can't use console consumer because we need a custom deserializer to interpret our message records. Let's look at our consumer code shown below.

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  |  |
|  | import org.apache.kafka.clients.consumer.KafkaConsumer; |
|  | import org.apache.kafka.clients.consumer.ConsumerRecords; |
|  | import org.apache.kafka.clients.consumer.ConsumerRecord; |
|  |  |
|  | public class SupplierConsumer { |
|  |  |
|  | public static void main(String[] args) throws Exception { |
|  |  |
|  | String topicName = "SupplierTopic"; |
|  | String groupName = "SupplierTopicGroup"; |
|  |  |
|  | Properties props = new Properties(); |
|  | props.put("bootstrap.servers", "localhost:9092,localhost:9093"); |
|  | props.put("group.id", groupName); |
|  | props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); |
|  | props.put("value.deserializer", "SupplierDeserializer"); |
|  |  |
|  | KafkaConsumer<String, Supplier> consumer = new KafkaConsumer<>(props); |
|  | consumer.subscribe(Arrays.asList(topicName)); |
|  |  |
|  | while (true) { |
|  | ConsumerRecords<String, Supplier> records = consumer.poll(100); |
|  | for (ConsumerRecord<String, Supplier> record : records) { |
|  | System.out.println("Supplier id= " + String.valueOf(record.value().getID()) + " Supplier Name = " + record.value().getName() + " Supplier Start Date = " + record.value().getStartDate().toString()); |
|  | } |
|  | } |
|  |  |
|  | } |
|  | } |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT33.java)[KT33.java](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt33-java) hosted with ❤ by [GitHub](https://github.com/)

We are creating our first consumer. We haven't created a consumer earlier in this tutorial. Understanding consumers will take another set of sessions, so I am not going to explain the above consumer code in this lesson. That's a separate topic covered in the later lesions. But just to give you a glimpse, we implemented a loop. The loop is processing each message that the consumer received from the Kafka broker. I am only displaying all three supplier fields on the console. So, it will show you all the suppliers received on the consumer side.  
If you want to execute the code and observe the outcome. Follow the video tutorial. The video demonstrates the compilation and execution process using SBT. If you want to use SBT, you will need a build file. You can use below content in your build file to compile this example.

|  |  |
| --- | --- |
|  | name := "KafkaTest" |
|  |  |
|  | libraryDependencies ++= Seq( |
|  | "org.apache.kafka" % "kafka-clients" % "0.10.1.0" |
|  | exclude("javax.jms", "jms") |
|  | exclude("com.sun.jdmk", "jmxtools") |
|  | exclude("com.sun.jmx", "jmxri") |
|  | exclude("org.slf4j", "slf4j-simple") |
|  | ) |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT34.build)[KT34.build](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt34-build) hosted with ❤ by [GitHub](https://github.com/)

That's it for this session. We will cover some more details on Kafka producer in next session. See you again.  
Thanks for watching Learning Journal. Keep learning and keep growing.

Apache Kafka Foundation Course - Producer Configs

Welcome to Kafka tutorials at Learning Journal. We have been creating producers in our earlier sessions. So far, we have covered almost every aspect of a Kafka producer. In this session, we will conclude our discussion about Kafka Producers.  
In Kafka, almost everything is controlled using configurations. In our earlier examples, we used four configuration parameters. Three of them were mandatory and fourth parameter was for custom partitioner. Let me recap all of them.

bootstrap.servers

key.serializer

value.serializer

partitioner.class

Kafka Bootstrap Servers

The first parameter, bootstrap.servers is a list of Kafka broker URI and port number. Since it is mandatory, so, we must have at least one value specified for this parameter. The value that we provide for this parameter is used by the producer to connect to Kafka cluster. Without this value, the producer cannot reach to the cluster. You should provide at least two addresses because if the first broker is down the producer should reach out at the second address. If you have a large cluster, you can provide more than two address. There is no harm in providing 3 to 4 addresses.

Kafka Serializers & Partitioners

The second parameter is a key.serializer. This parameter takes the name of the class that you want to use for serializing your key. The third parameter is a value.serializer. This parameter takes the name of the class that you planned to use as a value serializer. You can use the same class for both key and value. If your key and value both are strings, you can use the same serializer for both. However, if you are sending record or an object, using the same serializer for both key and value does not make any sense.  
The last one is partitioner.class. So, if you are using a custom partitioner, you should specify your class name for this parameter.  
We have already used all these parameters. So, I am sure that you learned them already.  
The Kafka producer provides many configuration parameters. The complete list of producer parameter is available in Kafka documentation. Most of the parameters have a reasonable default value, so there is no need to customize many of them. I recommend that you check the documentation and read all of them at least once. We have excellent documentation, and most of them are straightforward. I will cover three important parameters in this session because they have a direct impact on the reliability and performance of Kafka.

acks

retries

max.in.flight.requests.per.connection

Remember that these are producer configurations. So, you can set these configurations using properties just like you are setting bootstrap.servers or partitioner.class. The effect of setting these properties can be seen at the producer level, not at the topic level or the server level.  
Let's start with the first configuration.

Kafka acks

The acks configuration is to configure acknowledgments. When producers send a message to Kafka broker, they get a response back from the broker. The response is a RecordMetaData object or an exception.  
This parameter acks, it can take three values: 0, 1, and all. If we set it to 0, the producer will not wait for the response. It will send the message over the network and forget. There are three side effects of acks being 0.

Possible loss of messages

High throughput

No Retries

Since producer is not waiting for the response, there is no guarantee that the server has received the record. So, you may lose some records.  
However, since the producer is not waiting for an acknowledgment, it can send data as fast as the network can support and achieve high throughput.  
The third side effect is that the producer will not even go for a retry. Kafka is a highly available system, so there is a slim possibility that you lose your record. However, understand that there is no guarantee. So, use this setting when loss of few messages is not an issue. This setting will provide you the highest possible throughput.  
If we set acks to 1, the producer will wait for the response. However, the response is sent by the leader. So, this parameter will have an impact on when the leader is going to send the response. In this case, the leader will respond after recording the message in its local storage.  
If the leader is down and message delivery fails, the producer can retry after few milliseconds.  
This option appears to be a safe choice. However, there is a catch. You still cannot guarantee that you will not lose your message.  
You might be wondering how I can lose the record if it is received at the leader? Well, You can. Because we have a single copy of the message. We are not sure that it is replicated. What if leader crashes. You will lose your message. Correct?  
Replicators are fast. They replicate it quickly. However, If the leader breaks before replica could make a copy, the message will be lost. Surprisingly, in such scenario, the messages can be lost even after successful acknowledgment.  
The chance of losing your record is thinner than the earlier option, but it is not a reliable option.  
If you want to achieve 100% reliability, it is necessary that all replicas in the ISR list should make a copy successfully before the leader sends an acknowledgment.  
That is where the all setting works.  
If we set acks parameter to all, the leader will acknowledge only after it receives an acknowledgment from all of the live replicas. This option gives you the highest reliability but costs you the highest latency.  
The all setting is the slowest option because you will be waiting for all replicas. However, you can achieve better throughput using asynchronous send.

Kafka Retries and Max in flight requests

Now the next parameter. The parameter retries is a simple one. It defines how many times the producer will retry after getting an error. The default value is 0. There is another parameter retry.backoff.ms that controls the time between two retries. The default value for this parameter is 100 milliseconds.  
The next parameter is max.in.flight.requests.per.connection. This one is crucial and often less understood. Let me try to explain it. If you are using asynchronous send with a callback function to check your errors. You are not waiting for a response, but you ultimately get the response using a call back function. So, do you know how many such messages you can send without waiting for a response? The question is, how many in-flight requests are allowed that are still not acknowledged?  
That's the number defined by max.in.flight.requests.per.connection parameter. Setting this parameter to a high value will increase memory usage, but at the same time, it will increase throughput as well. So, if you have enough memory, you may want to set it to a higher value to achieve better performance of an asynchronous send.  
There is a side effect of asynchronous send. Let's assume you send 10 requests for same partition, 5 of them were sent as the first batch and failed. Remaining five goes as a second batch and succeed. Now the producer will retry the first batch, and if it is successful, you lost your order. That's a significant side effect of asynchronous send. So, be careful if the order of delivery is critical for you. If you are looking for an ordered delivery of your messages, you have following two options.

Use synchronous send.

set max.in.flight.requests.per.connection to 1

Both options have the same result. The order is critical for some use cases, especially transactional systems, for example banks and inventory. If you are working on that kind of use case, set max.in.flight.requests.per.connection to 1.

Other Kafka Producer Configs

There are few more important properties. I recommend checking Kafka document for at least following properties.

buffer.memory

compression.type

batch.size

linger.ms

client.id

max.request.size

All these properties are relatively straightforward. However, if you have any doubts, you can reach out to me for clarification.  
The primary objective of this session was to understand the ordering guarantee of Kafka. You should have a fair idea by now that you can preserve the order in a Kafka partition, but it comes at the cost of throughput.  
That’s it for this session. In the next video, we will start exploring Kafka consumers. Thank you for watching learning journal.

Apache Kafka Foundation Course - Consumer Groups

Welcome to Kafka tutorials at Learning Journal. I hope you are following this training from the beginning. We have covered most of the basics of Kafka and explored Kafka producers in detail. Now it is time to explore consumer side of it.  
In this session, I will talk about consumer groups. I have already covered consumer groups. So, I assume that you have a fair idea about it. But before we start creating different types of consumers, it is necessary to understand some nuances of a consumer group. So, let's start.  
If your producers are pushing data to the topic at a moderate speed, a single consumer may be enough to read and process that data. However, if you want to scale up your system and read data from Kafka in parallel, you need multiple consumers reading your topic in parallel. Many applications may have a clear need for multiple producers pushing data to a topic at one end and multiple consumers reading and processing data on the other end.  
There is no complexity at the producer side. It is as simple as executing another instance of a producer. There is no coordination or sharing of information is needed among producers.  
But on the consumer side, we have various considerations. Let's discuss those factors and understand the solution that Kafka provides. Let’s start with the first question.

Kafka Consumer Group?

When I talk about parallel reading, I am speaking about one single application consuming data in parallel. It is not about multiple applications reading same Kafka topic in parallel.  
So, the question is, how to implement parallel reads in a single application.  
I think you already know the answer.  
We can do that by creating a group and starting multiple consumers in the same group. That part is simple. We will see some code examples for creating multiple consumers in the same group. But that part is straight forward.  
However, there is a concern for duplicate reads.  
If we have multiple consumers reading data in parallel from the same topic, don't you think that all of them can read the same message?  
The answer is no. Kafka provides a very simple solution for this problem. Only one consumer owns a partition at any point in time. What does that mean? Let's take an example to understand this.  
We have one topic, and there are four partitions. So, if we have only one consumer in a group, it reads from all four partitions. If you have two, each of them reads two partitions. If you have three, the arrangement may be something like a single consumer reading two partitions and others own a single partition each. So, the fundamental concept is that the consumers do not share a partition. There is no way we can read the same message more than once.

However, this solution also brings a limitation. The number of partitions on a topic is the upper limit of consumers you can have in a group. So, in our example, if you have five consumers, one of them reads nothing. Kafka won't complain that you have four partitions, but you are starting five consumers. Simply, the fifth consumer will have nothing to read.  
So far so good. I have four partitions and four consumer processes. All reading in parallel and no one is reading each other's data. So, no duplicate reads. However, I have another doubt.

How does a consumer enter and exit into a group?

This question is obvious. Isn't it? You started with one Consumer and wanted to scale up, so you added one more. Now you have two of them. Which partition should this new consumer read? Who should pull some partitions from the first consumer and assign them to the second consumer? Somebody should be there to manage this.  
This reassignment problem does not end there. Assume you have four consumers, but one crashed, so you are left with three. What should happen to that partition? Who should read it now?  
After some time, the collapsed consumer has recovered, so again you have four of them. Now, a reassignment will be required once again.  
In a real distributed application, consumers keep joining and exiting. We do not have control over that. My question is, how Kafka handles it? When a consumer joins a group, how is a partition assigned to it? Moreover, what happens to the partition when a consumer leaves the group? Who manages all of this?

Kafka Group Coordinator

The answer is simple. A group coordinator oversees all of this. So, one of the Kafka broker gets elected as a Group Coordinator. When a consumer wants to join a group, it sends a request to the coordinator. The first consumer to participate in a group becomes a leader. All other consumers joining later becomes the members of the group.  
So, we have two actors, A coordinator, and a group leader. The coordinator is responsible for managing a list of group members. So, every time a new member joins the group, or an existing member leaves the group, the coordinator modifies the list.  
On an event of membership change, the coordinator realizes that it is time to rebalance the partition assignment. Because you may have a new member, and you need to assign it some partitions, or a member left, and you need to reassign those partitions to someone else, So, every time the list is modified, the coordinator initiates a rebalance activity.

Kafka Group Leader

The group leader is responsible for executing rebalance activity. The group leader will take a list of current members, assign partitions to them and send it back to the coordinator. The Coordinator then communicates back to the members about their new partitions. The important thing to note here is, during the rebalance activity, none of the consumers are allowed to read any message.

Summary

Let us summarize it quickly.

Consumer Groups –They are used to read and process data in parallel.

Partitions are not shared - To protect duplicate reads in a group, Kafka does not allow more than one Consumers to read data from a single partition at the same time.

A Group Coordinator - A broker is designated as a group coordinator and it maintains a list of active consumers.

Rebalance - Every time the list of active consumers is modified, the coordinator orders a rebalance activity to the group leader.

The Group leader - executes a rebalance activity.

Rebalance activity is nothing but assigning partitions to individual consumers.  
That’s it for this session. Thank you for visiting learning journal. Keep learning and keep growing.

Apache Kafka Foundation Course - Creating Consumers

Hello and welcome to Kafka tutorials at Learning Journal. In our previous session, we covered Kafka consumer groups. In this session, we will look at the code for our fist Kafka consumer and try to understand some details around it. We already created a Kafka consumer in an earlier tutorial, so let us take the same example for this session.

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  |  |
|  | import org.apache.kafka.clients.consumer.KafkaConsumer; |
|  | import org.apache.kafka.clients.consumer.ConsumerRecords; |
|  | import org.apache.kafka.clients.consumer.ConsumerRecord; |
|  |  |
|  | public class SupplierConsumer { |
|  |  |
|  | public static void main(String[] args) throws Exception { |
|  |  |
|  | String topicName = "SupplierTopic"; |
|  | String groupName = "SupplierTopicGroup"; |
|  |  |
|  | Properties props = newProperties(); |
|  | props.put("bootstrap.servers", "localhost:9092,localhost:9093"); |
|  | props.put("group.id", groupName); |
|  | props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); |
|  | props.put("value.deserializer", "SupplierDeserializer"); |
|  |  |
|  | KafkaConsumer consumer = newKafkaConsumer <>(props); |
|  | consumer.subscribe(Arrays.asList(topicName)); |
|  |  |
|  | while (true) { |
|  | ConsumerRecords records = consumer.poll(100); |
|  | for (ConsumerRecordrecord: |
|  | records) { |
|  | System.out.println("Supplier id= " + |
|  | String.valueOf(record.value().getID()) + |
|  | " Supplier Name = " + record.value().getName() + |
|  | " Supplier Start Date = " + |
|  | record.value().getStartDate().toString()); |
|  | } |
|  | } |
|  | } |
|  | } |

Set Properties

Creating a Kafka consumer is very similar to creating a producer. We create a properties object and set three mandatory properties.

bootstrap.servers

key.deserializer

value.deserializer

We already know that bootstrap.servers is a list of Kafka brokers and we need this information to connect to Kafka cluster.  
In a producer, we used key and value serializers, but in a consumer, we need a deserializer. If you are sending string messages, you can use string deserializer. In this example, we are using a string deserializer for the key and a custom deserializer for the value.

The next property is group.id. In the previous session, we learned about consumer groups. You can specify your consumer group name as a value of this property. You might be wondering that don't we need to create a group first and then join the group? No, you don't need to worry about all those things as creating a group and participating in a group, who is the group coordinator and who is the group leader. All that is taken care by the API. For us, it is as simple as specifying a group name. The group name is a string, so you can choose any string name for your group.  
The group id property is not mandatory, so you can skip it. But you should know that when you are not part of any group that means you are starting an independent consumer and your code will read all the data for the topic. Since you are not part of any group, there will be no sharing of work and your consumer need to read all data and process all of it alone.

Subscribe to topic

Great, so once we set up all consumer properties, the next step is to create a Kafka Consumer object and subscribe to one or more topics.  
The subscribe method takes a list of topics so you can subscribe to multiple topics at a time. If you want to subscribe to many topics, you can also use regular expression or wildcard in this method. Subscribing to a topic means you are informing Kafka broker that you want to read data for these topics.  
After subscribing, you want to fetch some records and process them. That's what the while loop is all about. The poll method will return some messages. You process them and again fetch for some more. The parameter to the pool method is a timeout. If there is no data to poll, you don't want to be hanging there, so this value specifies how quickly you want the pool method to return with or without data.

The Poll Method

The poll function is pretty powerful and takes care of a lot of things. It handles all the coordination, partition rebalances, and heart beat for group coordinator and provides you a clean and straightforward API. When you call to poll for the first time from a consumer, it finds a group coordinator, joins the group, receives partition assignment and fetches some records from those partitions. Every time you call to poll, it will send a heartbeat to group coordinator. So, it is necessary that whatever you do in a consumer, it should be quick and efficient. If you don't poll for a while, the coordinator may assume that the consumer is dead and trigger a partition rebalance. That's all about the Poll method.

The infinite loop

You might be wondering about the infinite loop. Consumers are usually long running processes, so having an infinite loop is perfectly fine. For some batch processing systems, you may not want this kind of infinite loop. Some requirements may need a consumer to wake up every few hours, process all the records collected during that interval and sleep again for few hours. Modelling such requirement should be simple. You can use a scheduler to start your consumer,lets say every six hours, and you can exit the while loop and safely quit after processing all the records collected during the period. You should make sure to make a call to close method to clean up resources and let the coordinator know that you are leaving the group.  
We already discussed in the earlier session that exiting a consumer will initiate a partition rebalance activity. That means the partition that you were processing will go to some other consumer. There are certain clean-up activities to be performed before you exit. Those actions are necessary to make sure that there is no second processing and we don't end up with duplicate records. We will cover them in upcoming sessions.

Configuration properties

Before closing this video, I just want to highlight one more thing.  
So, in all our examples, we have been creating a property object, but ideally, you should load it from a properties file. Keeping properties in a separate file is more flexible, and you will have all your configurations outside the code. So, let us change the code and load property values from an external file.  
Java properties file is a simple text file that contains a key value pair. The first thing that I need to do is to create a file and move all the properties there. The content of the properties file for my example looks like this.

|  |  |
| --- | --- |
|  | bootstrap.servers=localhost:9092,localhost:9093 |
|  | key.deserializer=org.apache.kafka.common.serialization.StringDeserializer |
|  | value.deserializer=SupplierDeserializer |
|  | group.id=SupplierTopicGroup |

It is simple, right. We have a key and value. One pair in each line.  
So, we moved all our properties in this file.  
Now it’s time to modify the code. I have changed the code, and it looks like this.

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  | import java.io.\*; |
|  |  |
|  | import org.apache.kafka.clients.consumer.KafkaConsumer; |
|  | import org.apache.kafka.clients.consumer.ConsumerRecords; |
|  | import org.apache.kafka.clients.consumer.ConsumerRecord; |
|  |  |
|  | public class NewSupplierConsumer { |
|  |  |
|  | public static void main(String[] args) throws Exception { |
|  |  |
|  | String topicName = "SupplierTopic"; |
|  | String groupName = "SupplierTopicGroup"; |
|  | Properties props = new Properties(); |
|  | //props.put("bootstrap.servers", "localhost:9092,localhost:9093"); |
|  | //props.put("group.id", groupName); |
|  | //props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); |
|  | //props.put("value.deserializer", "SupplierDeserializer"); |
|  |  |
|  | InputStream input = null; |
|  | KafkaConsumer<String, Supplier> consumer = null; |
|  |  |
|  | try { |
|  | input = new FileInputStream("SupplierConsumer.properties"); |
|  | props.load(input); |
|  | consumer = new KafkaConsumer<>(props); |
|  | consumer.subscribe(Arrays.asList(topicName)); |
|  |  |
|  | while (true) { |
|  | ConsumerRecords<String, Supplier> records = consumer.poll(100); |
|  | for (ConsumerRecord<String, Supplier> record : records) { |
|  | System.out.println("Supplier id= " + String.valueOf(record.value().getID()) + |
|  | " Supplier Name = " + record.value().getName() + |
|  | " Supplier Start Date = " + record.value().getStartDate().toString()); |
|  | } |
|  | } |
|  | } catch (Exception ex) { |
|  | ex.printStackTrace(); |
|  | } finally { |
|  | input.close(); |
|  | consumer.close(); |
|  | } |
|  | } |
|  | } |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT37.java)[KT37.java](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt37-java) hosted with ❤ by [GitHub](https://github.com/)

I changed the name to NewSupplierConsumer. You can see that I commented out some lines. Those lines are not required as we are loading them from an external file.  
We open the properties file and the load all the key-value pairs into an object. Everything else is same.  
That’s it for this session. Thank you for watching learning journal. Keep learning and keep growing.

Apache Kafka Foundation Course - Offset Management

Welcome to Kafka tutorials at Learning Journal. In our previous session, we created our first consumer and covered some basics around poll method. In this Kafka tutorial, we will cover some internals of offset management in Apache Kafka. I will explain current offset and committed offset. I will also include an example to show synchronous and asynchronous commit.  
Let me first define the offset. The offset is a position within a partition for the next message to be sent to a consumer. Kafka maintains two types of offsets.

Current offset

Committed offset

Current Offset

Let me first explain the current offset. When we call a poll method, Kafka sends some messages to us. Let us assume we have 100 records in the partition. The initial position of the current offset is 0. We made our first call and received 20 messages. Now Kafka will move the current offset to 20. When we make our next request, it will send some more messages starting from 20 and again move the current offset forward. The offset is a simple integer number that is used by Kafka to maintain the current position of a consumer. That's it. The current offset is a pointer to the last record that Kafka has already sent to a consumer in the most recent poll. So, the consumer doesn't get the same record twice because of the current offset.

Committed Offset

Now let us come to committed offset, this offset is the position that a consumer has confirmed about processing. What does that mean? After receiving a list of messages, we want to process it. Right? This processing may be just storing them into HDFS. Once we are sure that we have successfully processed the record, we may want to commit the offset. So, the committed offset is a pointer to the last record that a consumer has successfully processed. For example, the consumer received 20 records. It is processing them one by one, and after processing each record, it is committing the offset. We will see a code example of this in a while.  
So, in summary.

Current offset -> Sent Records -> This is used to avoid resending same records again to the same consumer.

Committed offset -> Processed Records -> It is used to avoid resending same records to a new consumer in the event of partition rebalance.

The committed offset is critical in the case of partition rebalance.  
In the event of rebalancing. When a new consumer is assigned a new partition, it should ask a question. Where to start? What is already processed by the previous owner? The answer to the question is the committed offset.

How to commit an offset?

Now, since we understand both the offsets maintained by Kafka, the next question is, How to commit an offset?  
There are two ways to do it.

Auto commit

Manual-commit

The commit has a significant impact on the client application, so we need to choose an appropriate method based on our use case. Let us look at the auto-commit approach.

Auto Commit

Auto-commit is the easiest method. You can control this feature by setting two properties.

enable.auto.commit

auto.commit.interval.ms

The first property is by default true. So auto-commit is enabled by default. You can turn it off by setting enable.auto.commit to false. The second property defines the interval of auto-commit. The default value for this property is five seconds. So, in a default configuration, when you make a call to the poll method, it will check if it is time to commit. If you have passed five seconds since the previous call, the consumer will commit the last offset. So, Kafka will commit your current offset every five seconds.  
The auto-commit is a convenient option, but it may cause second processing of records. Let us understand it with an example.  
You have some messages in the partition, and you made your first poll request. You received 10 messages hence the consumer increases the current offset to 10. You take four seconds to process these ten messages and make a new call. Since you haven't passed five seconds, the consumer will not commit the offset. You received another set of records, and for some reason rebalance is triggered at this moment. First ten records are already processed, but nothing is committed yet. Right? The rebalance is triggered. So, the partition goes to a different consumer. Since we don't have a committed offset, the new owner of partition should start reading from the beginning and process first ten records again.  
You might be thinking that let's reduce the commit frequency to four seconds. You can lower the incidence of commit by setting the auto-commit interval to a lower value, but you can't guarantee to eliminate repeat processing.

Manual Commit

The solution to this particular problem is a manual commit. So, we can configure the auto-commit off and manually commit after processing the records. There are two approaches to manual commit.

Commit Sync

Commit async

I hope you already understand the difference between synchronous and asynchronous. Synchronous commit is a straightforward and reliable method, but it is a blocking method. It will block your call for completing a commit operation, and it will also retry if there are recoverable errors.  
Asynchronous commit will send the request and continue. The drawback is that commitAsync will not retry. But there is a valid reason for such behaviour. Let's understand it with an example.  
Let us assume that you are trying to commit an offset as seventy-five. It failed for some recoverable reason, and you want to retry it after few seconds. Since this was an asynchronous call, so without knowing that your previous commit is waiting, you initiated another commit. This time it is to commit-100 Your commit 100 is successful while commit-75 waits for a retry. What do you want to do? Obviously, you don't want to commit 75 after commit 100. That may cause problems. So, they designed asynchronous commit to not to retry. However, this behaviour is not an issue because you know that if one commit fails for a recoverable reason, the next higher order commit will succeed.  
Now we understand automatic and manual commits. It’s time to write some code and see how to implement it.

Manual commit example

In this example, we will use asynchronous commit. But in the case of an error, we want to make sure that we commit before we close and exit. So, we will use synchronous commit before we close our consumer.

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  | import java.io.\*; |
|  |  |
|  | import org.apache.kafka.clients.consumer.KafkaConsumer; |
|  | import org.apache.kafka.clients.consumer.ConsumerRecords; |
|  | import org.apache.kafka.clients.consumer.ConsumerRecord; |
|  |  |
|  | public class ManualConsumer { |
|  |  |
|  | public static void main(String[] args) throws Exception { |
|  |  |
|  | String topicName = "SupplierTopic"; |
|  | String groupName = "SupplierTopicGroup"; |
|  |  |
|  | Properties props = new Properties(); |
|  | props.put("bootstrap.servers", "localhost:9092,localhost:9093"); |
|  | props.put("group.id", groupName); |
|  | props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); |
|  | props.put("value.deserializer", "SupplierDeserializer"); |
|  | props.put("enable.auto.commit", "false"); |
|  |  |
|  | KafkaConsumer<String, Supplier> consumer = null; |
|  |  |
|  | try { |
|  | consumer = new KafkaConsumer<>(props); |
|  | consumer.subscribe(Arrays.asList(topicName)); |
|  |  |
|  | while (true) { |
|  | ConsumerRecords<String, Supplier> records = consumer.poll(100); |
|  | for (ConsumerRecord<String, Supplier> record : records) { |
|  | System.out.println("Supplier id= " + String.valueOf(record.value().getID()) + |
|  | " Supplier Name = " + record.value().getName() + |
|  | " Supplier Start Date = " + record.value().getStartDate().toString()); |
|  | } |
|  | consumer.commitAsync(); |
|  | } |
|  | } catch (Exception ex) { |
|  | ex.printStackTrace(); |
|  | } finally { |
|  | consumer.commitSync(); |
|  | consumer.close(); |
|  | } |
|  | } |
|  | } |

The code is straightforward, and we have already seen it earlier. There is nothing new except two new lines. The first one is asynchronous commit and the second one is synchronous commit. In this example, I am manually committing my current offset before pulling the next set of records.  
You may be wondering that does it solve my problem completely.  
I mean, I got 100 records in the first poll. After processing all 100 records, I am committing my current offset.

What if a rebalance occurs after processing 50 records?

What if an exception occurs after processing 50 records?

I leave these two questions for you to think and post me an answer as a comment or start a discussion on these two issues.  
Let me give you a hint.  
You can fix both above problems if you know how to commit a particular offset instead of committing current offset. Kafka offset management and handling rebalance gracefully is the most critical part of implementing appropriate Kafka consumers.  
In the next session, we will see a more involved example and learn how to commit an appropriate offset and handle a rebalance more gracefully.  
That’ it for this session. Thank you for watching learning journal. Keep learning and keep growing.

Apache Kafka Foundation Course - Rebalance Listener

Welcome to Kafka tutorials at Learning Journal. In the previous session, we learned synchronous and asynchronous commit. But both methods were used to commit the latest offset. In this session, we will cover committing a particular offset and also learn about rebalance listener. Let me give you an example.

Kafka Rebalance scenario

Suppose, we have a situation where we got a lot of data using the poll method, and it is going to take some reasonable time to complete the processing for all the records. If you are taking a lot of time to process your records, you will have two types of risks.

The first risk is the delay in next pool, because you are busy processing data from the last call. If you don't poll for a long, the group coordinator might assume that you are dead and trigger a rebalance activity. You don't want that to happen, Right? Because you were not dead, you were computing.

The second risk is also related to rebalancing. The coordinator triggers a rebalance activity for some other reason while you are processing an extensive list of messages.

In both the cases, rebalance is triggered either because you didn't poll for a while or something else went wrong. Your current partitions will be taken away from you and reassigned to some other consumer. In such cases, you would want to commit whatever you have already processed before the ownership of the partition is taken away from you. And the new owner of the partition is supposed to start consuming it from the last committed offset.  
How can you do this? Obviously, you will need to know at least two things.

How to commit a particular offset?  
So, you can keep committing intermediate offsets instead of having committed the current offset in the end.

How to know that a rebalance is triggered?  
So, you can commit whatever you already processed.

The synchronous and asynchronous commit that we learned earlier is committing the latest offset given by the last poll. We don't want that. I will show you an example that will maintain a current offset of processed records and commit the current offset when a rebalancing is triggered.  
The answer to the second question is simple. Kafka API allows us to specify a ConsumerRebalanceListener class. This class offers two methods.

onPartitionsRevoked

onPartitionsAssigned

The Kafka will call the onPartitionsRevoked method just before it takes away your partitions. So, this is where you can commit your current offset.  
The Kafka will call the onPartitionsAssigned method right after the rebalancing is complete and before you start consuming records from the new partitions. In this example, we don't have anything to do, but there are scenarios when you may want to use this method. We will discuss that situation in the next session.

Kafka Random Producer

Let's look at the complete example code.I have created a new producer for this example.

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  |  |
|  | import org.apache.kafka.clients.producer.\*; |
|  |  |
|  | public class RandomProducer { |
|  |  |
|  | public static void main(String[] args) throws InterruptedException { |
|  |  |
|  | String topicName = "RandomProducerTopic"; |
|  | String msg; |
|  |  |
|  | Properties props = new Properties(); |
|  | props.put("bootstrap.servers", "localhost:9092,localhost:9093"); |
|  | props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer"); |
|  | props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer"); |
|  |  |
|  | Producer<String, String> producer = new KafkaProducer<>(props); |
|  | Random rg = new Random(); |
|  | Calendar dt = Calendar.getInstance(); |
|  | dt.set(2016, 1, 1); |
|  | try { |
|  | while (true) { |
|  | for (int i = 0; i < 100; i++) { |
|  | msg = dt.get(Calendar.YEAR) + "-" + |
|  | dt.get(Calendar.MONTH) + "-" + |
|  | dt.get(Calendar.DATE) + "," + |
|  | rg.nextInt(1000); |
|  | producer.send(new ProducerRecord<String, String>(topicName, 0, "Key", msg)).get(); |
|  | msg = dt.get(Calendar.YEAR) + "-" + |
|  | dt.get(Calendar.MONTH) + "-" + |
|  | dt.get(Calendar.DATE) + "," + |
|  | rg.nextInt(1000); |
|  | producer.send(new ProducerRecord<String, String>(topicName, 1, "Key", msg)).get(); |
|  | } |
|  | dt.add(Calendar.DATE, 1); |
|  | System.out.println("Data Sent for " + |
|  | dt.get(Calendar.YEAR) + "-" + |
|  | dt.get(Calendar.MONTH) + "-" + |
|  | dt.get(Calendar.DATE)); |
|  | } |
|  | } catch (Exception ex) { |
|  | System.out.println("Intrupted"); |
|  | } finally { |
|  | producer.close(); |
|  | } |
|  |  |
|  | } |
|  | } |

I am not going to explain the producer in detail because we have already learned almost everything about producers and you already understand the code. This producer will send data to the topic named RandomProducerTopic. I have already created this topic, and it has two partitions. If you look at those two send method calls, you will notice that we are sending the first message to partition zero and then next message to partition one. I am making these calls in an infinite loop, so the producer will keep sending data to both the partitions. This flow of continuous messages to both the partitions will help us simulate a rebalance and understand the behaviour of consumers.

Kafka Replacing Consumer

Now, let's look at the Consumer. You already know how to create a consumer. The first step is to setup all necessary properties. The second action is to instantiate a consumer object and subscribe to the topics. The final step is to poll the messages in a loop and process them. But in this example, we want to implement a rebalance listener.

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  |  |
|  | import org.apache.kafka.clients.consumer.\*; |
|  | import org.apache.kafka.common.\*; |
|  |  |
|  | public class RandomConsumer { |
|  |  |
|  | public static void main(String[] args) throws Exception { |
|  |  |
|  | String topicName = "RandomProducerTopic"; |
|  | KafkaConsumer<String, String> consumer = null; |
|  |  |
|  | String groupName = "RG"; |
|  | Properties props = new Properties(); |
|  | props.put("bootstrap.servers", "localhost:9092,localhost:9093"); |
|  | props.put("group.id", groupName); |
|  | props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); |
|  | props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); |
|  | props.put("enable.auto.commit", "false"); |
|  |  |
|  | consumer = new KafkaConsumer<>(props); |
|  | RebalanceListnerrebalanceListner = new RebalanceListner(consumer); |
|  |  |
|  | consumer.subscribe(Arrays.asList(topicName), rebalanceListner); |
|  | try { |
|  | while (true) { |
|  | ConsumerRecords<String, String> records = consumer.poll(100); |
|  | for (ConsumerRecord<String, String> record : records) { |
|  | /\*System.out.println("Topic:"+ record.topic() + |
|  | " Partition:" + record.partition() + |
|  | " Offset:" + record.offset() + " Value:"+ record.value());\*/ |
|  | // Do some processing and save it to Database |
|  | rebalanceListner.addOffset(record.topic(), record.partition(), record.offset()); |
|  | } |
|  | //consumer.commitSync(rebalanceListner.getCurrentOffsets()); |
|  | } |
|  | } catch (Exception ex) { |
|  | System.out.println("Exception."); |
|  | ex.printStackTrace(); |
|  | } finally { |
|  | consumer.close(); |
|  | } |
|  | } |
|  |  |
|  | } |

So, let's first understand the responsibilities of the listener. We want it to take care of two things.

Maintain a list of offsets that are processed and ready to be committed.

Commit the offsets when partitions are going away.

So, we want to maintain our personal list of offsets instead of relying on the current offsets that are managed by Kafka. This list will give us a complete freedom on what we want to commit. Right? We will look at the code for a listener in a minute, but let's see how we are using it in our consumer.  
So, we instantiate a listener object. The Listener should have access to the consumer object for executing a commit, and that's why we provide a consumer reference to the listener. Then the listener object is given to the Kafka on a subscribe method call. By doing this, we make sure that Kafka will invoke the listener's onPartitionsRevoked method.  
So far so good, we have setup a rebalance listener and Kafka will invoke the listener before taking our partitions, and we will conveniently commit before we lose them.  
The rest is simple. The poll method will fetch some records. We process them one by one in the for-loop. After processing each message, we tell our listener that this particular offset is ready to be committed. The listener will not commit them immediately. It will just maintain a list of latest offsets per topic per partition that it should commit.  
Once you finish processing all the messages and you are ready to make the next poll, you should commit the offsets and reset the list. That's it for the consumer.

Kafka Rebalance Listener example

Let's look at the code for the listener.

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  |  |
|  | import org.apache.kafka.clients.consumer.\*; |
|  | import org.apache.kafka.common.\*; |
|  |  |
|  | public class RebalanceListner implements ConsumerRebalanceListener { |
|  | private KafkaConsumer consumer; |
|  | private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap(); |
|  |  |
|  | public RebalanceListner(KafkaConsumer con) { |
|  | this.consumer = con; |
|  | } |
|  |  |
|  | public void addOffset(String topic, int partition, long offset) { |
|  | currentOffsets.put(new TopicPartition(topic, partition), new OffsetAndMetadata(offset, "Commit")); |
|  | } |
|  |  |
|  | public Map<TopicPartition, OffsetAndMetadata> getCurrentOffsets() { |
|  | return currentOffsets; |
|  | } |
|  |  |
|  | public void onPartitionsAssigned(Collection<TopicPartition> partitions) { |
|  | System.out.println("Following Partitions Assigned ...."); |
|  | for (TopicPartition partition : partitions) |
|  | System.out.println(partition.partition() + ","); |
|  | } |
|  |  |
|  | public void onPartitionsRevoked(Collection<TopicPartition> partitions) { |
|  | System.out.println("Following Partitions Revoked ...."); |
|  | for (TopicPartition partition : partitions) |
|  | System.out.println(partition.partition() + ","); |
|  |  |
|  |  |
|  | System.out.println("Following Partitions commited ...."); |
|  | for (TopicPartition tp : currentOffsets.keySet()) |
|  | System.out.println(tp.partition()); |
|  |  |
|  | consumer.commitSync(currentOffsets); |
|  | currentOffsets.clear(); |
|  | } |
|  | } |

Your listener class must implement ConsumerRebalanceListener interface. Then I have a private variable to store a reference to the consumer. The constructor will set this variable. Then I have another variable of type Map. This variable is to maintain the offsets. Topic name and partition number is the key for the Map data structure.It will just keep the latest offset for the topic and partition. These offsets are ready to get committed. That's how the Map data structure works. Right? You add an item on the key, and it replaces the old one. I hope you already understand that.  
We have three methods to handle the map of offsets. Those are straightforward, and I don't think they need any explanation. The onPartitionsAssigned method will print the list of partitions that are assigned. We don't have anything else to be done in that method.  
The onPartitionsRevoked method will also print the list of partitions that are going away. Then, it will commit and reset the list. That's it. Now, it's time to execute the example and show you the effect of rebalancing.  
The video demonstrates the process in detail. However, I have also listed the steps below.

Create the Topic with two partitions.

Start the producer in one terminal. The producer will make sure that the consumers will have some messages to read.

Start a consumerin another terminal.

You can observethe output messages. It shows that theconsumer has got both the partitions. Partition 0 and partition 1. This one is the only consumer, so it should read both the partitions.

Now if you start another consumer.What should happen? Following activities will take place.

A rebalance will be triggered. Because you have a new consumer and it should have some partition to read.

Kafka will revoke all partitions from the first consumer because the list of consumers in a group is modified.

Then, both consumers will get new partition assignment, and each of them should get one partition.

So, let's start the second consumer in a new terminal.  
You should observe the output messages. You must notice that Kafka revoked both the partitions.But, before losing them, we committed both the partitions. The rebalance listener has taken care of the commit. You should also observe that both the consumers have got new partition assignment.

If you kill one of those consumers, what will happen? You know it. Right? Another rebalance activity will be triggered, and Kafka will assign both the partitions to the surviving consumer. You can try that yourself.

That's it for this session. Thank you for watching learning journal. Keep watching and keep growing

Apache Kafka Foundation Course - Exactly Once Processing

Welcome to Kafka tutorials at Learning Journal. We have learned quite a lot about Kafka subscribers. In this session, we will study how to take full control of your Kafka subscriber. So, let's start.  
In our earlier sessions, we learned about two things that are managed by Kafka.

Automatic group management & Partition assignment.

Offset and consumer positions control.

Let me quickly recap both things.We already learned that to become a member of a group you just need to provide a group name and subscribe to the topics. That's it. And Kafka will make you a member of a consumer group. But have you thought about it? Why do you want to become a member of a Group? What are the benefits of creating a Consumer Group? You already know the answer. Right? It gives following advantages.

Allows you to parallel process a topic.

Automatically manages partition assignment.

Detects entry/exit/failure of a consumer and perform partition rebalancing activity.

Correct? So, most of the time, you want all these features. But there is a downside of automatic partition assignment. You don't have control over the partition assignment. I mean, you can't decide which partitions will be assigned to which consumer. Kafka takes this decision. So, if you have built a solution with a custom partitioning strategy and you want to make use of the custom partitioning, this automatic partition assignment will create problems for you. To understand it in a better way, I must take you back to my custom partitioning example.

Problems with Kafka Consumer Groups

You remember, we used a custom partitioner to make sure that all the data for TSS sensor will land into first three partitions. You have that system in place for collecting data. Now you have a real-time statistics calculation requirement for TSS sensor. So, you need to create a consumer that reads only TSS data and calculates required statistics. Making sense? Pause for a minute and think about it. Can you solve this problem?  
One way is that you read all the data and discard everything else except TSS and then calculate your stats. But you don't want to do unnecessary work. After all, it's a real-time system and every millisecond counts. You might be thinking that you should have created a separate topic for TSS. Then, it was easy. Right? You can subscribe to TSS topic and rest is simple. But this system was designed in this way for a reason, and now you don't have a luxury of redesigning.  
You can find a solution easily if you have the flexibility to create a consumer and assign these three partitions to yourself. That's it. Your Consumer will read these three partitions and rest will be simple.  
I think this example makes a point that in certain use cases, you may need the flexibility to take control of the partition assignment instead of relying on Kafka to do a random assignment. I will show you a relevant code example in a minute.  
But let me come back to offset management and consumer positioning as well. So, we already know that Kafka maintains a current offset and a committed offset for each consumer. We have also seen an example for rebalance listener where we take partial control over the offset. I am not sure if you already noticed that we still have a problem in that implementation.  
Let me take you back to that example and explain the problem.

|  |  |
| --- | --- |
|  | //Code from random consumer example |
|  | while(true){ |
|  | ConsumerRecords<String, String> records=consumer.poll(100); |
|  | for(ConsumerRecord<String, String>record:records){ |
|  | /\*System.out.println("Topic:"+ record.topic() + |
|  | " Partition:" + record.partition() + |
|  | " Offset:" + record.offset() + |
|  | " Value:"+ record.value());\*/ |
|  | //Step - 1 |
|  | // Do some processing and save it to Database |
|  | rebalanceListner.addOffset(record.topic(),record.partition(),record.offset()); |
|  | } |
|  | //Step - 2 |
|  | //consumer.commitSync(rebalanceListner.getCurrentOffsets()); |
|  | } |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT42.java)[KT42.java](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt42-java) hosted with ❤ by [GitHub](https://github.com/)

So, this is the consumer that we created. Just look at the for-loop. First step is to process the record and save the result in the database. Then in the next step, we keep the offset because we finished processing that message, and we will commit the offset later. Right? But thosetwo steps are two operations, and they are not atomic. It is possible that we save the record in the database and the consumer crashes before committing the offset.  
So, the rebalance listener will allow us to perform clean up and commit before the partition goes away, but it can't help us in synching the processed record and committed offset. There is no other way to achieve it except making SaveToDatabase and commit offset as a single atomic transaction.  
Great, now we understand that we may have use cases where we need to take full control of two things.

Partition assignment - That means, we don't want Kafka to assign partitions to different consumers. We want to take the control and assign desired partitions to ourselves.

Committed offset management - That means, we don't want Kafka to store the offset. We want to maintain the committed offset somewhere outside of Kafka. This approach allows us to develop transactional systems.

So, how to do that? Let's create an example.

Exactly once processing Example

So, I will create a consumer that will assign three TSS partitions to itself. Then, it will start reading all messages from those three partitions and insert each message into a database table. It will not commit the offset back to Kafka. Instead, it will update the current offset into another database table. The insert and update statements will be part of a single transaction, so either both will complete, or both will fail.  
I will use MySQL as a database for this demo. If you don't have MySQL installed and configured, you can follow instructions listed below.

Install MySQL server using below command. The Yum command will download and install MySQL server.

|  |  |
| --- | --- |
|  | yum install mysql-server |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT43.sh)[KT43.sh](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt43-sh) hosted with ❤ by [GitHub](https://github.com/)

Once Installation is over, start the MySQL server.

|  |  |
| --- | --- |
|  | service mysqld start |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT44.sh)[KT44.sh](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt44-sh) hosted with ❤ by [GitHub](https://github.com/)

MySQL service is up, let's make it secure. The below command will help you setup password for the MySQL root account.

|  |  |
| --- | --- |
|  | mysql\_secure\_installation |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT45.sh)[KT45.sh](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt45-sh) hosted with ❤ by [GitHub](https://github.com/)

Now we can start MySQL command line tool.

mysql -u root -p

Now we need to do following things.

Create a database.

Create a table to insert TSS sensor data.

Create a table to update TSS offsets.

Insert 3 rows for each TSS partition for initial offsets.

I created a simple SQL script to do all of this. The script is simple, and I assume it doesn't require any explanation. Let's execute the script.

|  |  |
| --- | --- |
|  | source tss.sql |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT46.sh)[KT46.sh](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt46-sh) hosted with ❤ by [GitHub](https://github.com/)

At this stage, you should have the table for sensor data but no data inside the table. The Consumer will insert the data, so we wanted an empty table.

You should have initial offsets table with initial values set to zero. So, the consumer should start reading from the beginning and keep updating as it reads.

The code for the script file is shown below.

|  |  |
| --- | --- |
|  | create database test; |
|  | use test; |
|  | create table tss\_data(skey varchar(50), svalue varchar(50)); |
|  | create table tss\_offsets(topic\_name varchar(50),partition int, offset int); |
|  | insert into tss\_offsets values('SensorTopic1',0,0); |
|  | insert into tss\_offsets values('SensorTopic1',1,0); |
|  | insert into tss\_offsets values('SensorTopic1',2,0); |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT47.sql)[KT47.sql](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt47-sql) hosted with ❤ by [GitHub](https://github.com/)

Now, it's time to look at the Consumer code.

Exactly once processing – Kafka Consumer

Let’s start with the main method. As always, we set up properties and then instantiate a Kafka consumer object. Other than mandatory properties, we have an additional property enable.auto.commit. This property will disable the auto-commit feature, and the consumer will not commit offsets automatically back to Kafka broker.

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  |  |
|  | import org.apache.kafka.clients.consumer.\*; |
|  | import org.apache.kafka.common.\*; |
|  |  |
|  | import java.sql.\*; |
|  |  |
|  | public class SensorConsumer { |
|  |  |
|  | public static void main(String[] args) throws Exception { |
|  | String topicName = "SensorTopic"; |
|  | KafkaConsumer<String, String> consumer = null; |
|  | int rCount; |
|  |  |
|  | Properties props = new Properties(); |
|  | props.put("bootstrap.servers", "localhost:9092,localhost:9093"); |
|  | props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); |
|  | props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); |
|  | props.put("enable.auto.commit", "false"); |
|  |  |
|  | consumer = new KafkaConsumer<>(props); |
|  | TopicPartition p0 = new TopicPartition(topicName, 0); |
|  | TopicPartition p1 = new TopicPartition(topicName, 1); |
|  | TopicPartition p2 = new TopicPartition(topicName, 2); |
|  |  |
|  | consumer.assign(Arrays.asList(p0, p1, p2)); |
|  | System.out.println("Current position p0=" + consumer.position(p0) |
|  | + " p1=" + consumer.position(p1) |
|  | + " p2=" + consumer.position(p2)); |
|  |  |
|  | consumer.seek(p0, getOffsetFromDB(p0)); |
|  | consumer.seek(p1, getOffsetFromDB(p1)); |
|  | consumer.seek(p2, getOffsetFromDB(p2)); |
|  | System.out.println("New positions po=" + consumer.position(p0) |
|  | + " p1=" + consumer.position(p1) |
|  | + " p2=" + consumer.position(p2)); |
|  |  |
|  | System.out.println("Start Fetching Now"); |
|  | try { |
|  | do { |
|  | ConsumerRecords<String, String> records = consumer.poll(1000); |
|  | System.out.println("Record polled " + records.count()); |
|  | rCount = records.count(); |
|  | for (ConsumerRecord<String, String> record : records) { |
|  | saveAndCommit(consumer, record); |
|  | } |
|  | } while (rCount > 0); |
|  | } catch (Exception ex) { |
|  | System.out.println("Exception in main."); |
|  | } finally { |
|  | consumer.close(); |
|  | } |
|  | } |
|  |  |
|  | private static long getOffsetFromDB(TopicPartition p) { |
|  | long offset = 0; |
|  | try { |
|  | Class.forName("com.mysql.jdbc.Driver"); |
|  | Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "pandey"); |
|  |  |
|  | String sql = "select offset from tss\_offsets where topic\_name='" |
|  | + p.topic() + "' and partition=" + p.partition(); |
|  | Statement stmt = con.createStatement(); |
|  | ResultSet rs = stmt.executeQuery(sql); |
|  | if (rs.next()) |
|  | offset = rs.getInt("offset"); |
|  | stmt.close(); |
|  | con.close(); |
|  | } catch (Exception e) { |
|  | System.out.println("Exception in getOffsetFromDB"); |
|  | } |
|  | return offset; |
|  | } |
|  |  |
|  | private static void saveAndCommit(KafkaConsumer<String, String> c, ConsumerRecord<String, String> r) { |
|  | System.out.println("Topic=" + r.topic() + " Partition=" + r.partition() + " Offset=" + r.offset() |
|  | + " Key=" + r.key() + " Value=" + r.value()); |
|  | try { |
|  | Class.forName("com.mysql.jdbc.Driver"); |
|  | Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "pandey"); |
|  | con.setAutoCommit(false); |
|  |  |
|  | String insertSQL = "insert into tss\_data values(?,?)"; |
|  | PreparedStatement psInsert = con.prepareStatement(insertSQL); |
|  | psInsert.setString(1, r.key()); |
|  | psInsert.setString(2, r.value()); |
|  |  |
|  | String updateSQL = "update tss\_offsets set offset=? where topic\_name=? and partition=?"; |
|  | PreparedStatement psUpdate = con.prepareStatement(updateSQL); |
|  | psUpdate.setLong(1, r.offset() + 1); |
|  | psUpdate.setString(2, r.topic()); |
|  | psUpdate.setInt(3, r.partition()); |
|  |  |
|  | psInsert.executeUpdate(); |
|  | psUpdate.executeUpdate(); |
|  | con.commit(); |
|  | con.close(); |
|  | } catch (Exception e) { |
|  | System.out.println("Exception in saveAndCommit"); |
|  | } |
|  | } |
|  | } |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT48.java)[KT48.java](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt48-java) hosted with ❤ by [GitHub](https://github.com/)

If we want automatic group management, we should subscribe to the topic and Kafka will automatically assign partitions to the consumer. But in this example, we don't want Kafka to assign partitions to us. So, we create three partition objects, one each for the three partitions that we want to read, then we self-assign these three partitions.  
Good, so the first part is taken care, we have three partitions, and we are ready to read data from those partitions. The next thing is to adjust the offset. The next three lines of code will set the offset position for three partitions. The getOffsetFromDB will read offset positions from MySQL database. The current values in the database are zero, so it will return zero. That's it. We are all set. We assigned partitions, and we adjusted offset to appropriate positions. All we need to do is read and process.  
The loop will keep polling Kafka till we are getting records. The saveAndCommit method will save each record in the database and alsoupdate offsets.  
The code for saveAndCommit is plain JDBC code. We set auto commit false, insert data, update offset, and finally, execute commit. That makes the method an atomic transaction and we can achieve exactly-once processing scenario.  
Kafka efficiently provides us at least once processing. But to achieve exactly-once processing, we need to do some extra work as we did in this example.  
You can test the code. The video demonstrates the execution and test scenario. Checkout the video for the demo.  
In this session, we learned to take control of partition assignment and offset management from Apache Kafka. Automatic group and offset management are convenient options, but by taking control in our hand, we have all the power to support complex requirements.  
That's it for this session. See you again in the next session. Thank you for watching learning journal. Keep learning and keep growing.

Apache Kafka Foundation Course - Schema Evolution - Part 1

Hello and welcome to Kafka tutorials at Learning Journal. We have learned almost every concept associated with Kafka producers and consumers. We learned Java APIs for Kafka and created several examples to put them to use. In this session, we will cover a suitable method to handle schema evolution in Apache Kafka.  
Schema evolution is all about dealing with changes in your message record over time. Let's assume that you are collecting clickstream and your original schema for each click is something like this.

Session id - An identifier for session

Browser - An identifier for the browser

Campaign - A custom identifier for a running campaign

Channel - A custom identifier for the section of the site

Referrer - A first hit referrer (ex - facebook.com)

IP - An IP address from your ISP

So, you have producers that are sending this data to Kafka, and you have consumers that are reading this data from Kafka and doing some analysis. We have already seen this kind of example in one of my earlier videos. If you missed that, go back and watch custom serializer and deserializer. I am sure you can deliver a solution to this requirement by implementing a custom serializer and a deserializer.  
You had this system in place for few months, and later you decided to upgrade your schema to something like this.

Session id - An identifier for session

Browser - An identifier for the browser

Campaign - A custom identifier for a running campaign

Channel - A custom identifier for the section of the site

Entry URL - A first hit referrer URL

IP - An IP address from your ISP

Language - An identifier for the language

OS ID - An identifier for the operating system

The problem starts now. If you are changing your schema, you need to create new producers because you want to send some new fields. Right? But I have two more questions.

Do I need to change all current producers?  
I mean, I am fine to create new producers to include additional fields. But I don't want the system to break if I am not upgrading all of them.

Do I need to change my existing consumers?  
Again, I am happy to create some new consumers to work with newly added fields. But my current consumers are doing good, and they have nothing to do with new attributes. I don't want to change them.

So, I don't want to change my current producers and consumers because that will be too much of work. So, what do I want? In fact, I want to support both old and new schema simultaneously. Can I support both versions of schemas?  
In a standard case, if you change your schema, you have to change everything, your producers, consumers, serializer, and deserializer. The problem doesn't end there. After making these changes, you can't read old messages because you changed the code and any attempt to read old messages using new code will raise an exception.  
In summary, I need to have a combination of old and new producers as well as a mix of old and new consumers. Kafka should be able to store both types of messages on the same topic. Consumers should be able to read both types of messages without any error.  
That's what is a typical schema evolution problem. How do you handle this schema evolution problem in Apache Kafka?  
But do you know, how do we take care of it in other systems?  
The industry solution to handling schema evolution is to include schema information with the data. So, when someone is writing data, they write schema and data both. And when someone wants to read that data, they first read schema and then read data based on the schema. If we follow this approach, we can keep changing schema as frequently as required without worrying to change our code because we are always reading schema before reading a message.  
There are pre-built and reusable serialization systems to help us and simplify the whole process of translating messages according to schemas and embedding schema information in the message record. Avro is one of them. It is the most popular serialization system for Hadoop and its ecosystem.

Apache Avro for Kafka

Kafka follows the same approach and uses Avro to handle schema evolution problem. Let me give you a brief introduction to Avro and them we will create an example to see how all of this works in Kafka.  
But, before that, you may need to download [Avro tools](http://central.maven.org/maven2/org/apache/avro/avro-tools/1.8.1/avro-tools-1.8.1.jar#_blank)because we will be using it in the upcoming example. If the direct URL is not working for any reason, you can go to the [Avro release page](http://avro.apache.org/releases.html#_blank) and follow the download link from there.  
Ok, we need to understand how Avro works. Then we will explore how to implement it in Kafka. Avro is a data serialization system, and it offers you four things.

Allows you to define a schema for your data.

Generates code for your schema. (Optional)

Provide APIs to serialize your data according to the schema and embed schema information in the data.

Provide APIs to extract schema information and deserialize your data based on the schema.

That's it. That's all a serialization system can give you. Let's see how it works. We will follow the four steps listed above.  
If you have some data, and you want to serialize it, the first thing is to define a schema for your data. Avro schemas are defined using JSON. So, let's define a schema for our clickstream record.  
So, here is my schema file.  
File name:- ClickRecordV1.avsc

|  |  |
| --- | --- |
|  | { "type": "record", |
|  | "name": "ClickRecord", |
|  | "fields": [ |
|  | {"name": "session\_id", "type": "string"}, |
|  | {"name": "browser", "type": ["string", "null"]}, |
|  | {"name": "campaign", "type": ["string", "null"]}, |
|  | {"name": "channel", "type": "string"}, |
|  | {"name": "referrer", "type": ["string", "null"], "default": "None"}, |
|  | {"name": "ip", "type": ["string", "null"]} |
|  | ] |
|  | } |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT49.json)[KT49.json](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt49-json) hosted with ❤ by [GitHub](https://github.com/)

The name of my schema is ClickRecord, and there are six items. The first field name is session\_id, and the data type is string. Session id and channel are mandatory. However, the browser, campaign and IP address can have a null value. The referrer has a default value none.  
It is a simple schema. However, Avro supports many other complex data types. For more details, you can refer to [Avro specification](https://avro.apache.org/docs/current/spec.html#_blank)  
Let's come back to the second step. Generate code for your schema. This step is simple. We will use Avro tool to generate code. Place your Avro schema file and the Avro tool in the same directory and execute the Avro tool using below command.

|  |  |
| --- | --- |
|  | java -jar avro-tools-1.8.1.jar compile schema ClickRecordV1.avsc |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT50.sh)[KT50.sh](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt50-sh) hosted with ❤ by [GitHub](https://github.com/)

To execute Avro tool,give it a command to compile schema. Then give the schema file name and the destination directory. Hit enter and Done. This command will generate java file in the same directory. That's your source code generated by Avro tool. If you quickly look at the generated Java code, you should realize that it creates a Java class for ClickRecord with get and set methods for each field. That’s it. We finished step 2.  
Let's come back to step 3 and 4.Avro provides API for serializing your data. So, you can use ClickRecord class generated in step 2 to create your data objects. And then you can use Avro APIs as stated in step 3 to serialize them and send to a Kafka broker. That's what a Kafka producer and a serializer do. So, at the producer end, I have to create a ClickRecord object and use a Avro serializer.  
Similarly, on the receiving end, I can use Avro APIs as stated in step 4 to deserialize those messages back to ClickRecord objects. That's what a deserializer does. So, at the receiver end, I have to use aAvro deserializer that extracts schema information from the message and use Avro APIs to deserialize my data back to the ClickRecord object.  
The point I want to make is that everything ultimately goes to an AvroSerializer and an AvroDeserializer. Producers and Consumers will only use the generated class to create data objects. Serializer and Deserializer take care of rest. So, Let me redefine these four steps regarding Kafka implementation.

Define an Avro Schema for your message record.

Generate a source code for your Avro Schema.

Create a producer and use KafkaAvroSerializer.

Create a consumer and use KafkaAvroDeserializer.

Kafka Avro Producer

The whole process of sending Kafka messages is quite straight forward. KafkaAvroSerializer and KafkaAvroDeserializer take all the complications away from a developer. You don't have to learn and use Avro APIs to serialize and deserialize your data. All that is already done and bundled into Kafka-Avro-Serializer module developed by the confluent team. Impressive! Isn't it? Now, let's look at the code for a producer.

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  |  |
|  | import org.apache.kafka.clients.producer.\*; |
|  |  |
|  | public class AvroProducer { |
|  |  |
|  | public static void main(String[] args) throws Exception { |
|  |  |
|  | String topicName = "AvroClicks"; |
|  | String msg; |
|  |  |
|  | Properties props = new Properties(); |
|  | props.put("bootstrap.servers", "localhost:9092,localhost:9093"); |
|  | props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer"); |
|  | props.put("value.serializer", "io.confluent.kafka.serializers.KafkaAvroSerializer"); |
|  | props.put("schema.registry.url", "http://localhost:8081"); |
|  |  |
|  | Producer<String, ClickRecord> producer = new KafkaProducer<>(props); |
|  | ClickRecord cr = new ClickRecord(); |
|  | try { |
|  | cr.setSessionId("10001"); |
|  | cr.setChannel("HomePage"); |
|  | cr.setIp("192.168.0.1"); |
|  |  |
|  | producer.send(new ProducerRecord<String, ClickRecord>(topicName, cr.getSessionId().toString(), cr)).get(); |
|  |  |
|  | System.out.println("Complete"); |
|  | } catch (Exception ex) { |
|  | ex.printStackTrace(System.out); |
|  | } finally { |
|  | producer.close(); |
|  | } |
|  |  |
|  | } |
|  | } |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT51.java)[KT51.java](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt51-java) hosted with ❤ by [GitHub](https://github.com/)

The code looks familiar, right. We have a topic AvroClicks, and we want to send our ClickRecords to this Topic. Like every other producer, we create three mandatory properties and set appropriate values. Our message keys are going to be a string, so we configure the key serializer class as StringSerializer. Our message values are going to be according to Avro Schema, so we set the value serializer class as KafkaAvroSerializer. The confluent team developed this class, and it takes care of all the serialization activity. It protects us from learning Avro APIs and takes care of all the complexities of serialization work.  
The next property is schema registry. This one is new. It's a new component developed by the confluent team. You might be wondering about the purpose of this tool.

Kafka Schema Registry

So far, we learned that KafkaAvroSerializer would take care of all the serialization work at producer end and KafkaAvroDeserializer will take care of all the deserialization work at the consumer end. But how do they communicate with each other about the schema? The deserializer should know the schema. Without knowing the schema, it can't deserialize the raw bytes back to an object. That's where the schema registry is useful. The KafkaAvroSerializer will store the schema details into the schema registry and include an ID of the schema into the message record. When KafkaAvroDeserializer receives a message, it takes the Schema ID from the message and gets schema details from the registry. Once we have the schema details and message bytes, it is simple to deserialize them. That's where we use the schema registry.  
Let's come back to our producer code.  
After setting up all the properties, everything else is straight forward.We create a ClickRecord object, set values for the fields and send it to Kafka Broker.  
I hope you remember that we generated the source code for the ClickRecord class using Avro tool. Now, we are using it in our producer. That's all we do in a producer.

Kafka Avro Consumer

The consumer code is also familiar.

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  |  |
|  | import org.apache.kafka.clients.consumer.\*; |
|  |  |
|  | public class AvroConsumer { |
|  |  |
|  | public static void main(String[] args) throws Exception { |
|  |  |
|  | String topicName = "AvroClicks"; |
|  | String groupName = "RG"; |
|  |  |
|  | Properties props = new Properties(); |
|  | props.put("bootstrap.servers", "localhost:9092,localhost:9093"); |
|  | props.put("group.id", groupName); |
|  | props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); |
|  | props.put("value.deserializer", "io.confluent.kafka.serializers.KafkaAvroDeserializer"); |
|  | props.put("schema.registry.url", "http://localhost:8081"); |
|  | props.put("specific.avro.reader", "true"); |
|  |  |
|  | KafkaConsumer<String, ClickRecord> consumer = new KafkaConsumer<>(props); |
|  | consumer.subscribe(Arrays.asList(topicName)); |
|  | try { |
|  | while (true) { |
|  | ConsumerRecords<String, ClickRecord> records = consumer.poll(100); |
|  | for (ConsumerRecord<String, ClickRecord> record : records) { |
|  | System.out.println("Session id=" + record.value().getSessionId() |
|  | + " Channel=" + record.value().getChannel() |
|  | + " Referrer=" + record.value().getReferrer()); |
|  | } |
|  | } |
|  | } catch (Exception ex) { |
|  | ex.printStackTrace(); |
|  | } finally { |
|  | consumer.close(); |
|  | } |
|  | } |
|  |  |
|  | } |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT52.java)[KT52.java](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt52-java) hosted with ❤ by [GitHub](https://github.com/)

We setup properties including schema registry URL and value deserializer as KafkaAvroDeserializer. We have another additional property specific.avro.reader=true. If we are using Avro schema and code generation for the schema, this property is mandatory. In fact, there is another generic method to use Avro schema without using Avro code generation tool, but I am not covering that part in this video. So, if you are generating code for your Avro schema, set this property as true.  
After setting all the properties, we create a consumer instance, subscribe to the topics and start polling and processing messages. KafkaAvroDeserializer takes care of all the pain of deserializing messages and converting it back to the ClickRecord object.  
We are ready to compile and execute the producer and consumer and see all of this working.

Confluent Platform for Kafka

But this example is dependent on the confluent platform. The schema registry and Kafka Avro serializer module is part of the Confluent platform. To execute this example, we need to download and install an open source version of the confluent platform.  
This session is getting too long, so I am splitting it into two parts.  
In this first part, we covered the notion of schema evolution and looked at Avro as a solution to the problem of schema evolution. We created a schema, generated code for the schema using Avro tool. Then we learned how a producer and consumer would use Avro schema. We talked about schema registry and its purpose.  
In the next session, we will download and install confluent platform.We will then compile and execute our code.  
But the whole point of using Avro is to see how to handle changes in the schema. So, we will modify our schema and learn how can we support both old and new schema in a system where both old and new producers and consumers are alive.  
Thank you for watching Learning Journal. Keep learning and keep growing.

Apache Kafka Foundation Course - Schema Evolution - Part 2

Welcome to Kafka Tutorial at Learning Journal. In the previous session, we talked about schema evolution problem. We learned Avro basics and created a producer and consumer that uses Avro schema.In this session, we will Install and configure open source version of the Confluent platform and execute our producer and consumer. We will also modify our schema and create a new version of the prior schema. Then, We will create a new producer to send some messages based on the new schema.  
This example will show us the schema evolution in action. We will see old producer, new producer and old consumer working together in the same system. Finally, we will create a new consumer as well. This new consumer will be able to read old and new both types of messages without any exception.By the end of this session, we will have a system where old and new both versions of the schema will be working with older and newer types of producers and consumers.  
So, let’s start.

Confluent Open Source

The confluent platform is, in fact, Apache Kafka packaged together with additional components. All these elements together with Apache Kafka make it a flexible and powerful streaming platform.There are many ways to download and install confluent platform. However, I am going to use RPM packages via Yum. You can follow [confluent documentation](http://docs.confluent.io/3.1.1/installation.html#_blank) , and install confluent platform. For an installation demo, you can watch the video.  
The current confluent platform at the time of recording this video is version is 3.1.1, and it comes with Kafka 0.10.1. Both versions are the latest releases at the time of recording this tutorial. I am expecting the same APIs to be released as Kafka 1.0 in near future.  
Once you installed Confluent platform, jump over to [quick start guide](http://docs.confluent.io/3.1.1/quickstart.html#_blank). Follow the quick start guide to start all necessary services. We need to start zookeeper, Kafka server and schema registry. I covered all of that in an earlier video for Apache Kafka installation and demo. We are doing similar things except starting schema registry as well. Once you have all the services running, you are ready to compile and execute our Avro producer and consumer.

To Compile our producer and consumer, we need to include Avro and Kafka-Avro-serializer dependencies. I have an SBT build file that contains these dependencies.

|  |  |
| --- | --- |
|  | name := "AvroTest" |
|  |  |
|  | val repositories = Seq( |
|  | "confluent" at "http://packages.confluent.io/maven/", |
|  | Resolver.sonatypeRepo("public") |
|  | ) |
|  |  |
|  | libraryDependencies ++= Seq( |
|  | "org.apache.avro" % "avro" % "1.8.1", |
|  | "io.confluent" % "kafka-avro-serializer" % "3.1.1", |
|  | "org.apache.kafka" % "kafka-clients" % "0.10.1.0" |
|  | exclude("javax.jms", "jms") |
|  | exclude("com.sun.jdmk", "jmxtools") |
|  | exclude("com.sun.jmx", "jmxri") |
|  | exclude("org.slf4j", "slf4j-simple") |
|  | ) |
|  |  |
|  | resolvers += "confluent" at "http://packages.confluent.io/maven/" |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT53.build)[KT53.build](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt53-build) hosted with ❤ by [GitHub](https://github.com/)

With the help of above build file, you can compile and execute the Avro producer and consumer. The video contains a demo for the same.

Schema Evolution in Kafka

So far, we learned that how can we use Avro schema in our producers and consumers. But the whole point of using Avro is to support evolving schemas. So, let's change our schema. Here is the new version of my schema.  
File Name:-ClickRecordV2.avsc

|  |  |
| --- | --- |
|  | {"type": "record", |
|  | "name": "ClickRecord", |
|  | "fields": [ |
|  | {"name": "session\_id", "type": "string"}, |
|  | {"name": "browser", "type": ["string", "null"]}, |
|  | {"name": "campaign", "type": ["string", "null"]}, |
|  | {"name": "channel", "type": "string"}, |
|  | {"name": "entry\_url", "type": ["string", "null"], "default": "None"}, |
|  | {"name": "ip", "type": ["string", "null"]}, |
|  | {"name": "language", "type": ["string", "null"], "default": "None"}, |
|  | {"name": "os", "type": ["string", "null"],"default": "None"} |
|  | ] |
|  | } |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT54.json)[KT54.json](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt54-json) hosted with ❤ by [GitHub](https://github.com/)

Note that the schema name and type don't change. The name and type of schema are still same as earlier. However, I changed the record structure. The first change is to remove referrer field and the second change is to add three new attributes. I have made these changes in the schema but let me warn you that you are not free to evolve your schema in a random fashion. There are some rules. Avro specification defines some rules for compatibility. You should refer to [Avro specification](http://avro.apache.org/docs/1.8.1/spec.html#_blank) for more details about compatibility guideline.  
So, we modified our schema. We need to generate code for this new schema. You can do that using below command. We performed this activity in the earlier lesson.

|  |  |
| --- | --- |
|  | java -jar avro-tools-1.8.1.jar compile schema ClickRecordV2.avsc |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT55.sh)[KT55.sh](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt55-sh) hosted with ❤ by [GitHub](https://github.com/)

Now, we need to create a new producer and send some messages in the new format.

Kafka Producer

So, here is my code for the new producer.

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  |  |
|  | import org.apache.kafka.clients.producer.\*; |
|  |  |
|  | public class ClickRecordProducerV2 { |
|  |  |
|  | public static void main(String[] args) throws Exception { |
|  |  |
|  | String topicName = "AvroClicks"; |
|  | String msg; |
|  |  |
|  | Properties props = new Properties(); |
|  | props.put("bootstrap.servers", "localhost:9092,localhost:9093"); |
|  | props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer"); |
|  | props.put("value.serializer", "io.confluent.kafka.serializers.KafkaAvroSerializer"); |
|  | props.put("schema.registry.url", "http://localhost:8081"); |
|  |  |
|  | Producer<String, ClickRecord> producer = new KafkaProducer<>(props); |
|  | ClickRecord cr = new ClickRecord(); |
|  | try { |
|  | cr.setSessionId("10001"); |
|  | cr.setChannel("HomePage"); |
|  | cr.setIp("192.168.0.1"); |
|  | cr.setLanguage("Spanish"); |
|  | cr.setOs("Mac"); |
|  | cr.setEntryUrl("http://facebook.com/myadd"); |
|  |  |
|  | producer.send(new ProducerRecord<String, ClickRecord>(topicName, cr.getSessionId().toString(), cr)).get(); |
|  | System.out.println("Complete"); |
|  | } catch (Exception ex) { |
|  | ex.printStackTrace(System.out); |
|  | } finally { |
|  | producer.close(); |
|  | } |
|  |  |
|  | } |
|  | } |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT56.java)[KT56.java](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt56-java) hosted with ❤ by [GitHub](https://github.com/)

The code is same as earlier, but this time, I am creating a ClickRecord object using new schema code. Then I am setting some fields and finally sending it to Kafka broker. This Producer will send a message in the new format.

Testing Schema Evolution in Kafka

Now, it's time to open three terminals. I will execute a consumer in one terminal. That wouldbe an old consumer. Then, I will start a new producer in one terminal and see if the old consumer can read the message sent by the new producer.  
The video shows that the old consumer can read a message that came with an evolved schema.  
You can start an old producer and check if your consumer is still able to read the old message as well. The video demo shows that use case successfully.  
So,we have seen that using Avro and schema registry, we can quickly build a system where producers and consumers can write and read messages using an evolving schema, and different versions of messages can co-exist in the same system.  
I have a written an example code for the new consumer as well. I am not going to explain it because it is essentially same as old consumer. The only difference is that it uses source code generated from the new schema. I leave it for you to try and test yourself. The code is listed below.

|  |  |
| --- | --- |
|  | import java.util.\*; |
|  |  |
|  | import org.apache.kafka.clients.consumer.\*; |
|  |  |
|  |  |
|  | public class ClickRecordConsumerV2 { |
|  |  |
|  | public static void main(String[] args) throws Exception { |
|  |  |
|  | String topicName = "AvroClicks"; |
|  |  |
|  | String groupName = "RG"; |
|  | Properties props = new Properties(); |
|  | props.put("bootstrap.servers", "localhost:9092,localhost:9093"); |
|  | props.put("group.id", groupName); |
|  | props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); |
|  | props.put("value.deserializer", "io.confluent.kafka.serializers.KafkaAvroDeserializer"); |
|  | props.put("schema.registry.url", "http://localhost:8081"); |
|  | props.put("specific.avro.reader", "true"); |
|  |  |
|  | KafkaConsumer<String, ClickRecord> consumer = new KafkaConsumer<>(props); |
|  | consumer.subscribe(Arrays.asList(topicName)); |
|  | try { |
|  | while (true) { |
|  | ConsumerRecords<String, ClickRecord> records = consumer.poll(100); |
|  | for (ConsumerRecord<String, ClickRecord> record : records) { |
|  | System.out.println("Session id=" + record.value().getSessionId() |
|  | + " Channel=" + record.value().getChannel() |
|  | + " Entry URL=" + record.value().getEntryUrl() |
|  | + " Language=" + record.value().getLanguage()); |
|  | } |
|  | } |
|  | } catch (Exception ex) { |
|  | ex.printStackTrace(); |
|  | } finally { |
|  | consumer.close(); |
|  | } |
|  | } |
|  |  |
|  | } |

[view raw](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2/raw/a0861292bf88916d33decb0de85ef8de65845df7/KT57.java)[KT57.java](https://gist.github.com/LearningJournal/48e57ed088dbca117327a207c1704eb2#file-kt57-java) hosted with ❤ by [GitHub](https://github.com/)

That's it for this session. Thank you for watching Learning Journal. Keep learning and keep growing

Oracle Sql Queries : -

<https://www.complexsql.com/complex-sql-queries-examples-with-answers/>

Q5. What are joins in SQL?

A JOIN clause is used to combine rows from two or more tables, based on a related column between them. It is used to merge two tables or retrieve data from there. There are 4 types of joins, as you can refer to below:

Chart, bubble chart

Description automatically generated

Inner join: [Inner Join in SQL](https://www.edureka.co/blog/sql-joins-types#_blank) is the most common type of join. It is used to return all the rows from multiple tables where the join condition is satisfied.

Left Join:  Left Join in SQL is used to return all the rows from the left table but only the matching rows from the right table where the join condition is fulfilled.

Right Join: Right Join in SQL is used to return all the rows from the right table but only the matching rows from the left table where the join condition is fulfilled.

Full Join: Full join returns all the records when there is a match in any of the tables. Therefore, it returns all the rows from the left-hand side table and all the rows from the right-hand side table.

Q6. What is the difference between CHAR and VARCHAR2 datatype in SQL?

Both Char and Varchar2 are used for characters datatype but varchar2 is used for character strings of variable length whereas Char is used for strings of fixed length. For example, char(10) can only store 10 characters and will not be able to store a string of any other length whereas varchar2(10) can store any length i.e 6,8,2 in this variable.

Q7. What is a Primary key?

A screenshot of a computer

Description automatically generated with medium confidenceA [Primary key in SQL](https://www.edureka.co/blog/primary-key-in-sql/#_blank) is a column (or collection of columns) or a set of columns that uniquely identifies each row in the table.

Uniquely identifies a single row in the table

Null values not allowed

Example- In the Student table, Stu\_ID is the primary key.

Q8. What are Constraints?

[Constraints in SQL](https://www.edureka.co/blog/sql-constraints/#_blank) are used to specify the limit on the data type of the table. It can be specified while creating or altering the table statement. The sample of constraints are:

NOT NULL

CHECK

DEFAULT

UNIQUE

PRIMARY KEY

FOREIGN KEY

Q9. What is the difference between DELETE and TRUNCATE statements?

|  |  |
| --- | --- |
| DELETE vs TRUNCATE | |
| DELETE | TRUNCATE |
| Delete command is used to delete a row in a table. | Truncate is used to delete all the rows from a table. |
| You can rollback data after using delete statement. | You cannot rollback data. |
| It is a DML command. | It is a DDL command. |
| It is slower than truncate statement. | It is faster. |

Q10. What is a Unique key?

Uniquely identifies a single row in the table.

Multiple values allowed per table.

Null values allowed.

Apart from this SQL Interview Questions blog, if you want to get trained from professionals on this technology, you can opt for [structured training from edureka!](https://www.edureka.co/sql-essentials-training#_blank)

Q11. What is a Foreign key in SQL?

Foreign key maintains referential integrity by enforcing a link between the data in two tables.

The foreign key in the child table references the primary key in the parent table.

The [foreign key constraint](https://www.edureka.co/blog/foreign-key-sql/#_blank) prevents actions that would destroy links between the child and parent tables.

**Q12.** **What do you mean by data integrity?**

Data Integrity defines the accuracy as well as the consistency of the data stored in a database. It also defines integrity constraints to enforce business rules on the data when it is entered into an application or a database.

**Q13.** **What is the difference between clustered and non-clustered index in SQL?**

The differences between the clustered and non clustered index in SQL are :

Clustered index is used for easy retrieval of data from the database and its faster whereas reading from non clustered index is relatively slower.

Clustered index alters the way records are stored in a database as it sorts out rows by the column which is set to be clustered index whereas in a non clustered index, it does not alter the way it was stored but it creates a separate object within a table which points back to the original table rows after searching.

One table can only have one clustered index whereas it can have many non clustered index.

**Q18. What is an Index?**

An index refers to a performance tuning method of allowing faster retrieval of records from the table. An index creates an entry for each value and hence it will be faster to retrieve data.

**Q19. Explain different types of index in SQL.**

There are three [types of index in SQL](https://www.edureka.co/blog/index-in-sql/#_blank) namely:

**Unique Index:**

This index does not allow the field to have duplicate values if the column is unique indexed. If a primary key is defined, a unique index can be applied automatically.

**Clustered Index:**

This index reorders the physical order of the table and searches based on the basis of key values. Each table can only have one clustered index.

**Non-Clustered Index:**

Non-Clustered Index does not alter the physical order of the table and maintains a logical order of the data. Each table can have many nonclustered indexes.

https://www.giantstride.gr/en/sql-indexing-part2/

**Q23. What is the ACID property in a database?**

ACID stands for Atomicity, Consistency, Isolation, Durability. It is used to ensure that the data transactions are processed reliably in a database system.

**Atomicity:** Atomicity refers to the transactions that are completely done or failed where transaction refers to a single logical operation of a data. It means if one part of any transaction fails, the entire transaction fails and the database state is left unchanged.

**Consistency:** Consistency ensures that the data must meet all the validation rules. In simple words,  you can say that your transaction never leaves the database without completing its state.

**Isolation:** The main goal of isolation is concurrency control.

**Durability:** Durability means that if a transaction has been committed, it will occur whatever may come in between such as power loss, crash or any sort of error.

https://www.geeksforgeeks.org/acid-properties-in-dbms/

**Q24. What do you mean by “Trigger” in SQL?**

[Trigger in SQL](https://www.edureka.co/blog/triggers-in-sql/#_blank) is a special type of stored procedures that are defined to execute automatically in place or after data modifications. It allows you to execute a batch of code when an insert, update or any other query is executed against a specific table.

**Q56.** **What is a Stored Procedure?**

A Stored Procedure is a function which consists of many SQL statements to access the database system. Several SQL statements are consolidated into a stored procedure and execute them whenever and wherever required which saves time and avoid writing code again and again.

**Q57. List some advantages and disadvantages of Stored Procedure?**

**Advantages:**

A Stored Procedure can be used as a modular programming which means create once, store and call for several times whenever it is required. This supports faster execution. It also reduces network traffic and provides better security to the data.

Disadvantage:

The only disadvantage of Stored Procedure is that it can be executed only in the database and utilizes more memory in the database server.

**Type of SQL Keys** :Multiple types of Keys are supported by the SQL Server.

The following are the list of SQL Keys: <https://www.guru99.com/dbms-keys.html>

**Primary Key ,**[**Unique Key**](https://www.educba.com/unique-key-in-sql/) **,Candidate Key ,Alternate Key ,Composite Key , Super Key ,Foreign Key**

For Example

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Customer Table | | | | |
| cust\_id | cust\_name | cust\_address | cust\_aadhaar\_number | cust\_pan\_number |
| 100001 | Sunil Kumar | Noida | 372464389211 | ADSFS3456K |
| 100002 | Ankit Gupta | Gr Noida | 442289458453 | CGHAD7583L |
| 100003 | Suresh Yadav | New Delhi | 878453444144 | NMKRT2278O |
| 100004 | Nilam Singh | Lucknow | 227643441123 | HFJFD3876U |
| 100005 | Amal Rawat | Ghaziabad | 932571156735 | CBMVA9734A |
| 100006 | Harsh Saxena | Kanpur | 1453534363319 | TRYUC2568H |

Below given the “Order” table having the related data corresponding to the “cust\_id” from the Customer Table.

|  |  |  |
| --- | --- | --- |
| Order Table | | |
| cust\_id | order\_month\_year | order\_amount |
| 100001 | 2019 – Jan | $100,000 |
| 100002 | 2019 – Jan | $120,000 |
| 100003 | 2019 – Jan | $100,000 |
| 100004 | 2019 – Jan | $110,000 |
| 100001 | 2019 – Feb | $105,000 |
| 100002 | 2019 – Feb | $125,000 |

Now, we will go through one by one on each of the Key:

1. Primary Key

Primary Key is a field that can be used to identify all the tuples uniquely in the database. Only one of the columns can be declared as a primary key. A Primary Key can not have a NULL value.

Example: In the above given relational table, “cust\_id” is the Primary Key as it can identify all the row uniquely from the table.

2. Unique Key

Unique Key can be a field or set of fields that can be used to uniquely identify the tuple from the database. One or more fields can be declared as a unique Key. The unique Key column can also hold the NULL value. Use of Unique Key improves the performance of data retrieval. It makes searching for records from the database much more faster & efficient.

Example: In the above given relational table, “cust\_aadhaar\_number”, “cust\_pan\_number” are the Unique Key as it can allow one value as a NULL in the column

3. Candidate Key

Candidate Key can be a column or group of columns that can qualify for the Unique Key. Every table has at least one Candidate Key. A table may have one or more Candidate Key. Each Candidate Key can work as a Primary Key if required in certain scenarios.

Example: In the above given relational table, “cust\_id”, “cust\_aadhaar\_number”, “cust\_pan\_number” are the Candidate Key as it can identify all the row uniquely from the table. These columns also qualify the criteria to be a Primary Key.

 4. Alternate Key

Alternate Key is that Key which can be used as a Primary Key if required. Alternate Key also qualifies to be a Primary Key but for the time being, It is not the Primary Key.

Example: In the above given relational table, “cust\_aadhaar\_number”, “cust\_pan\_number” are the Alternate Key as both of the columns can be a Primary Key but not yet selected for the Primary Key.

5. Composite Key

Composite Key is also known as Compound Key / Concatenated Key. Composite Key refers to a group of two or more columns that can be used to identify a tuple from the table uniquely. A group of the column in combination with each other can identify a row uniquely but a single column of that group doesn’t promise to identify the row uniquely.

Example: In the above given relational table i.e. Order Table, “cust\_id”, “order\_month\_year” group of these columns used in combination to identify the tuple uniquely in the Order Table. The individual column of this table is not able to identify the tuple uniquely from the Order table.

6. Super Key

Super Key is a combination of columns, each column of the table remains dependent on it. Super Key may have some more columns in the group which may or may not be necessary to identify the tuple uniquely from the table. Candidate Key is the subset of the Super Key. Candidate Key is also known as minimal Super Key.

Example: In the above given relational table, Primary Key, Candidate Key & Unique Key is the Super Key. As a single column of Customer Table i.e ‘cust\_id’ is sufficient to identify the tuples uniquely from the table. Any set of the column which contains ‘cust\_aadhaar\_number’, ‘cust\_pan\_number’ is a Super Key.

7. Foreign Key

A foreign key is a column which is known as Primary Key in the other table i.e. A Primary Key in a table can be referred to as a Foreign Key in another table. Foreign Key may have duplicate & NULL values if it is defined to accept NULL values.

Example: In the above given relational table, ‘cust\_id’ is Primary Key in the Customer table but ‘cust\_id’ in the Order table known as a ‘Foreign Key’. Foreign Key in a table always becomes the Primary Key on the other table.

select all row which has empty or null value in Column(COL\_NAME)

SELECT \* FROM TABLE WHERE COL\_NAME IS NULL OR LENGTH(TRIM (COL\_NAME)) = 0

MongoDB :-

[https://docs.mongodb.com/manual/tutorial/](https://docs.mongodb.com/manual/tutorial/query-array-of-documents/)

<https://www.guru99.com/what-is-mongodb.html>

What is MongoDB?

MongoDB is a document-oriented NoSQL database used for high volume data storage. Instead of using tables and rows as in the traditional relational databases, MongoDB makes use of collections and documents. Documents consist of key-value pairs which are the basic unit of data in MongoDB. Collections contain sets of documents and function which is the equivalent of relational database tables. MongoDB is a database which came into light around the mid-2000s.

MongoDB Features

Schema-less Database: It is the great feature provided by the MongoDB. A Schema-less database means one collection can hold different types of documents in it. Or in other words, in the MongoDB database, a single collection can hold multiple documents and these documents may consist of the different numbers of fields, content, and size. It is not necessary that the one document is similar to another document like in the relational databases. Due to this cool feature, MongoDB provides great flexibility to databases.

Document Oriented: In MongoDB, all the data stored in the documents instead of tables like in RDBMS. In these documents, the data is stored in fields(key-value pair) instead of rows and columns which make the data much more flexible in comparison to RDBMS. And each document contains its unique object id.

Scalabiltiy: MongoDB provides horizontal scalability with the help of sharding. Sharding means to distribute data on multiple servers, here a large amount of data is partitioned into data chunks using the shard key, and these data chunks are evenly distributed across shards that reside across many physical servers. It will also add new machines to a running database.

Replication: MongoDB provides high availability and redundancy with the help of replication, it creates multiple copies of the data and sends these copies to a different server so that if one server fails, then the data is retrieved from another server.

Aggregation: It allows to perform operations on the grouped data and get a single result or computed result. It is similar to the SQL GROUPBY clause. It provides three different aggregations i.e, aggregation pipeline, map-reduce function, and single-purpose aggregation methods

High Performance: The performance of MongoDB is very high and data persistence as compared to another database due to its features like scalability, indexing, replication, etc.

Ad hoc queries : MongoDB supports searching by field, range queries, and regular expression searches. Queries can be made to return specific fields within documents.

Indexing : Indexes can be created to improve the performance of searches within MongoDB. Any field in a MongoDB document can be indexed.

Load balancing : MongoDB uses the concept of sharding to scale horizontally by splitting data across multiple MongoDB instances. MongoDB can run over multiple servers, balancing the load and/or duplicating data to keep the system up and running in case of hardware failure.

Key Components of MongoDB Architecture

Below are a few of the common terms used in MongoDB

\_id – This is a field required in every MongoDB document. The \_id field represents a unique value in the MongoDB document. The \_id field is like the document's primary key. If you create a new document without an \_id field, MongoDB will automatically create the field. So for example, if we see the example of the above customer table,

Mongo DB will add a 24 digit unique identifier to each document in the collection. ObjectId

ObjectIds are small, likely unique, fast to generate, and ordered. ObjectId values are 12 bytes in length, consisting of:

4-byte timestamp value, representing the ObjectId's creation, measured in seconds since the Unix epoch

5-byte random value

3-byte incrementing counter, initialized to a random value

| \_Id | CustomerID | CustomerName | OrderID |
| --- | --- | --- | --- |
| 563479cc8a8a4246bd27d784 | 11 | Guru99 | 111 |
| 563479cc7a8a4246bd47d784 | 22 | Trevor Smith | 222 |
| 563479cc9a8a4246bd57d784 | 33 | Nicole | 333 |

Collection – This is a grouping of MongoDB documents. A collection is the equivalent of a table which is created in any other RDMS such as Oracle or MS SQL. A collection exists within a single database. As seen from the introduction collections don't enforce any sort of structure.

Cursor – This is a pointer to the result set of a query. Clients can iterate through a cursor to retrieve results.

Database – This is a container for collections like in RDMS wherein it is a container for tables. Each database gets its own set of files on the file system. A MongoDB server can store multiple databases.

Document - A record in a MongoDB collection is basically called a document. The document, in turn, will consist of field name and values.

Field - A name-value pair in a document. A document has zero or more fields. Fields are analogous to columns in relational databases.

The following diagram shows an example of Fields with Key value pairs. So in the example below CustomerID and 11 is one of the key value pair's defined in the document.

[Diagram

Description automatically generated](https://www.guru99.com/images/MongoDB/112015_1051_Introductio2.png)

JSON – This is known as[JavaScript](https://www.guru99.com/interactive-javascript-tutorials.html)Object Notation. This is a human-readable, plain text format for expressing structured data. JSON is currently supported in many programming languages.

Just a quick note on the key difference between the \_id field and a normal collection field. The \_id field is used to uniquely identify the documents in a collection and is automatically added by MongoDB when the collection is created.

How mongoDB is different from RDBMS ?  
Some major differences in between MongoDB and the RDBMS are as follows:

|  |  |
| --- | --- |
| MongoDB | RDBMS |
| It is a non-relational and document-oriented database. | It is a relational database. |
| It is suitable for hierarchical data storage. | It is not suitable for hierarchical data storage. |
| It has a dynamic schema. | It has a predefined schema. |
| It centers around the CAP theorem (Consistency, Availability, and Partition tolerance). | It centers around ACID properties (Atomicity, Consistency, Isolation, and Durability). |
| In terms of performance, it is much faster than RDBMS. | In terms of performace, it is slower than MongoDB. |

Table

Description automatically generated

Advantages of MongoDB :

It is a schema-less NoSQL database. You need not to design the schema of the database when you are working with MongoDB.

It does not support join operation.

It provides great flexibility to the fields in the documents.

It contains heterogeneous data.

It provides high performance, availability, scalability.

It supports Geospatial efficiently.

It is a document oriented database and the data is stored in BSON documents.

It also supports multiple document ACID transition(string from MongoDB 4.0).

It does not require any SQL injection.

It is easily integrated with Big Data Hadoop

Disadvantages of MongoDB :

It uses high memory for data storage.

You are not allowed to store more than 16MB data in the documents.

The nesting of data in BSON is also limited you are not allowed to nest data more than 100 levels.

What is NoSQL?

NoSQL Database is a non-relational Data Management System, that does not require a fixed schema. It avoids joins, and is easy to scale. The major purpose of using a NoSQL database is for distributed data stores with humongous data storage needs. NoSQL is used for Big data and real-time web apps. For example, companies like Twitter, Facebook and Google collect terabytes of user data every single day.

NoSQL database stands for "Not Only SQL" or "Not SQL." Though a better term would be "NoREL", NoSQL caught on. Carl Strozz introduced the NoSQL concept in 1998.

Traditional RDBMS uses SQL syntax to store and retrieve data for further insights. Instead, a NoSQL database system encompasses a wide range of database technologies that can store structured, semi-structured, unstructured and polymorphic data. Let's understand about NoSQL with a diagram in this NoSQL database tutorial:

Chart

Description automatically generated

Why NoSQL?

The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data. The system response time becomes slow when you use RDBMS for massive volumes of data.

To resolve this problem, we could "scale up" our systems by upgrading our existing hardware. This process is expensive.

The alternative for this issue is to distribute database load on multiple hosts whenever the load increases. This method is known as "scaling out."

[Diagram

Description automatically generated](https://www.guru99.com/images/1/101818_0537_NoSQLTutori2.png)

NoSQL database is non-relational, so it scales out better than relational databases as they are designed with web applications in mind.

Features of NoSQL

Non-relational

NoSQL databases never follow the [relational model](https://www.guru99.com/relational-data-model-dbms.html)

Never provide tables with flat fixed-column records

Work with self-contained aggregates or BLOBs

Doesn't require object-relational mapping and data normalization

No complex features like query languages, query planners,

referential integrity joins, ACID

Schema-free

NoSQL databases are either schema-free or have relaxed schemas

Do not require any sort of definition of the schema of the data

Offers heterogeneous structures of data in the same domain

[Diagram

Description automatically generated](https://www.guru99.com/images/1/101818_0537_NoSQLTutori3.png)

NoSQL is Schema-Free

Types of NoSQL Databases

NoSQL Databases are mainly categorized into four types: Key-value pair, Column-oriented, Graph-based and Document-oriented. Every category has its unique attributes and limitations. None of the above-specified database is better to solve all the problems. Users should select the database based on their product needs.

Types of NoSQL Databases:

Key-value Pair Based

Column-oriented Graph

Graphs based

Document-oriented

[Graphical user interface, diagram, application

Description automatically generated](https://www.guru99.com/images/1/101818_0537_NoSQLTutori5.png)

What is the CAP Theorem?

CAP theorem is also called brewer's theorem. It states that is impossible for a distributed data store to offer more than two out of three guarantees

Consistency

Availability

Partition Tolerance

Consistency:

The data should remain consistent even after the execution of an operation. This means once data is written, any future read request should contain that data. For example, after updating the order status, all the clients should be able to see the same data.

Availability:

The database should always be available and responsive. It should not have any downtime.

Partition Tolerance:

Partition Tolerance means that the system should continue to function even if the communication among the servers is not stable. For example, the servers can be partitioned into multiple groups which may not communicate with each other. Here, if part of the database is unavailable, other parts are always unaffected.

The use Command

MongoDB use DATABASE\_NAME is used to create database. The command will create a new database if it doesn't exist, otherwise it will return the existing database.

Syntax

Basic syntax of use DATABASE statement is as follows −

use DATABASE\_NAME

Example

If you want to use a database with name <mydb>, then use DATABASE statement would be as follows −

>use mydb

switched to db mydb

To check your currently selected database, use the command db

>db

mydb

If you want to check your databases list, use the command show dbs.

>show dbs

local 0.78125GB

test 0.23012GB

Your created database (mydb) is not present in list. To display database, you need to insert at least one document into it.

>db.movie.insert({"name":"tutorials point"})

>show dbs

local 0.78125GB

mydb 0.23012GB

test 0.23012GB

In MongoDB default database is test. If you didn't create any database, then collections will be stored in test database.

The dropDatabase() Method

MongoDB db.dropDatabase() command is used to drop a existing database.

Syntax

Basic syntax of dropDatabase() command is as follows −

db.dropDatabase()

This will delete the selected database. If you have not selected any database, then it will delete default 'test' database.

Example

First, check the list of available databases by using the command, show dbs.

>show dbs

local 0.78125GB

mydb 0.23012GB

test 0.23012GB

>

If you want to delete new database <mydb>, then dropDatabase() command would be as follows −

>use mydb

switched to db mydb

>db.dropDatabase()

>{ "dropped" : "mydb", "ok" : 1 }

>

Now check list of databases.

>show dbs

local 0.78125GB

test 0.23012GB

The createCollection() Method

MongoDB db.createCollection(name, options) is used to create collection.

Syntax

Basic syntax of createCollection() command is as follows −

db.createCollection(name, options)

In the command, name is name of collection to be created. Options is a document and is used to specify configuration of collection.

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| Name | String | Name of the collection to be created |
| Options | Document | (Optional) Specify options about memory size and indexing |

Options parameter is optional, so you need to specify only the name of the collection. Following is the list of options you can use −

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| capped | Boolean | (Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also. |
| autoIndexId | Boolean | (Optional) If true, automatically create index on \_id field.s Default value is false. |
| size | number | (Optional) Specifies a maximum size in bytes for a capped collection. If capped is true, then you need to specify this field also. |
| max | number | (Optional) Specifies the maximum number of documents allowed in the capped collection. |

While inserting the document, MongoDB first checks size field of capped collection, then it checks max field.

Examples

Basic syntax of createCollection() method without options is as follows −

>use test

switched to db test

>db.createCollection("mycollection")

{ "ok" : 1 }

>

You can check the created collection by using the command show collections.

>show collections

mycollection

system.indexes

The following example shows the syntax of createCollection() method with few important options −

> db.createCollection("mycol", { capped : true, autoIndexID : true, size : 6142800, max : 10000 } ){

"ok" : 0,

"errmsg" : "BSON field 'create.autoIndexID' is an unknown field.",

"code" : 40415,

"codeName" : "Location40415"

}

>

In MongoDB, you don't need to create collection. MongoDB creates collection automatically, when you insert some document.

>db.tutorialspoint.insert({"name" : "tutorialspoint"}),

WriteResult({ "nInserted" : 1 })

>show collections

mycol

mycollection

system.indexes

tutorialspoint

>

The drop() Method :-

MongoDB's db.collection.drop() is used to drop a collection from the database.

Basic syntax of drop() command is as −db.COLLECTION\_NAME.drop()

MongoDB supports many datatypes. Some of them are −

String − This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.

Integer − This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.

Boolean − This type is used to store a boolean (true/ false) value.

Double − This type is used to store floating point values.

Min/ Max keys − This type is used to compare a value against the lowest and highest BSON elements.

Arrays − This type is used to store arrays or list or multiple values into one key.

Timestamp − ctimestamp. This can be handy for recording when a document has been modified or added.

Object − This datatype is used for embedded documents.

Null − This type is used to store a Null value.

Symbol − This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.

Date − This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.

Object ID − This datatype is used to store the document’s ID.

Binary data − This datatype is used to store binary data.

Code − This datatype is used to store JavaScript code into the document.

Regular expression − This datatype is used to store regular expression.

Add MongoDB Array using insert() with Example

The "insert" command can also be used to insert multiple documents into a collection at one time. The below code example can be used to insert multiple documents at a time.

The following example shows how this can be done,

Step 1) Create a[JavaScript](https://www.guru99.com/interactive-javascript-tutorials.html)variable called myEmployee to hold the array of documents

Step 2) Add the required documents with the Field Name and values to the variable

Step 3) Use the insert command to insert the array of documents into the collection

var myEmployee=

[

{

"Employeeid" : 1,

"EmployeeName" : "Smith"

},

{

"Employeeid" : 2,

"EmployeeName" : "Mohan"

},

{

"Employeeid" : 3,

"EmployeeName" : "Joe"

},

];

db.Employee.insert(myEmployee);

What is Primary Key in MongoDB?

In MongoDB, \_id field as the primary key for the collection so that each document can be uniquely identified in the collection. The \_id field contains a unique ObjectID value.

By default when inserting documents in the collection, if you don't add a field name with the \_id in the field name, then MongoDB will automatically add an Object id field as shown below

[Text

Description automatically generated](https://www.guru99.com/images/MongoDB/112115_0607_Introductio11.png)

When you query the documents in a collection, you can see the ObjectId for each document in the collection.

If you want to ensure that MongoDB does not create the \_id Field when the collection is created and if you want to specify your own id as the \_id of the collection, then you need to explicitly define this while creating the collection.

When explicitly creating an id field, it needs to be created with \_id in its name.

Let's look at an example on how we can achieve this.

db.Employee.insert({\_id:10, "EmployeeName" : "Smith"})

The insert() Method

To insert data into MongoDB collection, you need to use MongoDB's insert() or save() method.

Syntax

The basic syntax of insert() command is as follows −

>db.COLLECTION\_NAME.insert(document)

Example

> db.users.insert({

... \_id : ObjectId("507f191e810c19729de860ea"),

... title: "MongoDB Overview",

... description: "MongoDB is no sql database",

... by: "tutorials point",

... url: "http://www.tutorialspoint.com",

... tags: ['mongodb', 'database', 'NoSQL'],

... likes: 100

... })

WriteResult({ "nInserted" : 1 })

>

Here mycol is our collection name, as created in the previous chapter. If the collection doesn't exist in the database, then MongoDB will create this collection and then insert a document into it.

In the inserted document, if we don't specify the \_id parameter, then MongoDB assigns a unique ObjectId for this document.

\_id is 12 bytes hexadecimal number unique for every document in a collection. 12 bytes are divided as follows −

\_id: ObjectId(4 bytes timestamp, 3 bytes machine id, 2 bytes process id, 3 bytes incrementer)

You can also pass an array of documents into the insert() method as shown below:.

> db.createCollection("post")

> db.post.insert([

{

title: "MongoDB Overview",

description: "MongoDB is no SQL database",

by: "tutorials point",

url: "http://www.tutorialspoint.com",

tags: ["mongodb", "database", "NoSQL"],

likes: 100

},

{

title: "NoSQL Database",

description: "NoSQL database doesn't have tables",

by: "tutorials point",

url: "http://www.tutorialspoint.com",

tags: ["mongodb", "database", "NoSQL"],

likes: 20,

comments: [

{

user:"user1",

message: "My first comment",

dateCreated: new Date(2013,11,10,2,35),

like: 0

}

]

}

])

BulkWriteResult({

"writeErrors" : [ ],

"writeConcernErrors" : [ ],

"nInserted" : 2,

"nUpserted" : 0,

"nMatched" : 0,

"nModified" : 0,

"nRemoved" : 0,

"upserted" : [ ]

})

>

To insert the document you can use db.post.save(document) also. If you don't specify \_id in the document then save() method will work same as insert() method. If you specify \_id then it will replace whole data of document containing \_id as specified in save() method.

The insertOne() method

If you need to insert only one document into a collection you can use this method.

Syntax

The basic syntax of insert() command is as follows −

>db.COLLECTION\_NAME.insertOne(document)

Example

Following example creates a new collection named empDetails and inserts a document using the insertOne() method.

> db.createCollection("empDetails")

{ "ok" : 1 }

> db.empDetails.insertOne(

{

First\_Name: "Radhika",

Last\_Name: "Sharma",

Date\_Of\_Birth: "1995-09-26",

e\_mail: "radhika\_sharma.123@gmail.com",

phone: "9848022338"

})

{

"acknowledged" : true,

"insertedId" : ObjectId("5dd62b4070fb13eec3963bea")

}

>

The insertMany() method

You can insert multiple documents using the insertMany() method. To this method you need to pass an array of documents.

Example

Following example inserts three different documents into the empDetails collection using the insertMany() method.

> db.empDetails.insertMany(

[

{

First\_Name: "Radhika",

Last\_Name: "Sharma",

Date\_Of\_Birth: "1995-09-26",

e\_mail: "radhika\_sharma.123@gmail.com",

phone: "9000012345"

},

{

First\_Name: "Rachel",

Last\_Name: "Christopher",

Date\_Of\_Birth: "1990-02-16",

e\_mail: "Rachel\_Christopher.123@gmail.com",

phone: "9000054321"

},

{

First\_Name: "Fathima",

Last\_Name: "Sheik",

Date\_Of\_Birth: "1990-02-16",

e\_mail: "Fathima\_Sheik.123@gmail.com",

phone: "9000054321"

}

]

)

{

"acknowledged" : true,

"insertedIds" : [

ObjectId("5dd631f270fb13eec3963bed"),

ObjectId("5dd631f270fb13eec3963bee"),

ObjectId("5dd631f270fb13eec3963bef")

]

}

The find() Method

To query data from MongoDB collection, you need to use MongoDB's find() method.

Syntax

The basic syntax of find() method is as follows −

>db.COLLECTION\_NAME.find()

find() method will display all the documents in a non-structured way.

Example

Assume we have created a collection named mycol as −

> use sampleDB

switched to db sampleDB

> db.createCollection("mycol")

{ "ok" : 1 }

>

And inserted 3 documents in it using the insert() method as shown below −

> db.mycol.insert([

{

title: "MongoDB Overview",

description: "MongoDB is no SQL database",

by: "tutorials point",

url: "http://www.tutorialspoint.com",

tags: ["mongodb", "database", "NoSQL"],

likes: 100

},

{

title: "NoSQL Database",

description: "NoSQL database doesn't have tables",

by: "tutorials point",

url: "http://www.tutorialspoint.com",

tags: ["mongodb", "database", "NoSQL"],

likes: 20,

comments: [

{

user:"user1",

message: "My first comment",

dateCreated: new Date(2013,11,10,2,35),

like: 0

}

]

}

])

Following method retrieves all the documents in the collection −

> db.mycol.find()

{ "\_id" : ObjectId("5dd4e2cc0821d3b44607534c"), "title" : "MongoDB Overview", "description" : "MongoDB is no SQL database", "by" : "tutorials point", "url" : "http://www.tutorialspoint.com", "tags" : [ "mongodb", "database", "NoSQL" ], "likes" : 100 }

{ "\_id" : ObjectId("5dd4e2cc0821d3b44607534d"), "title" : "NoSQL Database", "description" : "NoSQL database doesn't have tables", "by" : "tutorials point", "url" : "http://www.tutorialspoint.com", "tags" : [ "mongodb", "database", "NoSQL" ], "likes" : 20, "comments" : [ { "user" : "user1", "message" : "My first comment", "dateCreated" : ISODate("2013-12-09T21:05:00Z"), "like" : 0 } ] }

>

The pretty() Method

To display the results in a formatted way, you can use pretty() method.

Syntax

>db.COLLECTION\_NAME.find().pretty()

Example

Following example retrieves all the documents from the collection named mycol and arranges them in an easy-to-read format.

> db.mycol.find().pretty()

{

"\_id" : ObjectId("5dd4e2cc0821d3b44607534c"),

"title" : "MongoDB Overview",

"description" : "MongoDB is no SQL database",

"by" : "tutorials point",

"url" : "http://www.tutorialspoint.com",

"tags" : [

"mongodb",

"database",

"NoSQL"

],

"likes" : 100

}

{

"\_id" : ObjectId("5dd4e2cc0821d3b44607534d"),

"title" : "NoSQL Database",

"description" : "NoSQL database doesn't have tables",

"by" : "tutorials point",

"url" : "http://www.tutorialspoint.com",

"tags" : [

"mongodb",

"database",

"NoSQL"

],

"likes" : 20,

"comments" : [

{

"user" : "user1",

"message" : "My first comment",

"dateCreated" : ISODate("2013-12-09T21:05:00Z"),

"like" : 0

}

]

}

The findOne() method

Apart from the find() method, there is findOne() method, that returns only one document.

Syntax

>db.COLLECTIONNAME.findOne()

Example

Following example retrieves the document with title MongoDB Overview.

> db.mycol.findOne({title: "MongoDB Overview"})

{

"\_id" : ObjectId("5dd6542170fb13eec3963bf0"),

"title" : "MongoDB Overview",

"description" : "MongoDB is no SQL database",

"by" : "tutorials point",

"url" : "http://www.tutorialspoint.com",

"tags" : [

"mongodb",

"database",

"NoSQL"

],

"likes" : 100

}

RDBMS Where Clause Equivalents in MongoDB

To query the document on the basis of some condition, you can use following operations.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Syntax | Example | RDBMS Equivalent |
| Equality | {<key>:{$eg;<value>}} | db.mycol.find({"by":"tutorials point"}).pretty() | where by = 'tutorials point' |
| Less Than | {<key>:{$lt:<value>}} | db.mycol.find({"likes":{$lt:50}}).pretty() | where likes < 50 |
| Less Than Equals | {<key>:{$lte:<value>}} | db.mycol.find({"likes":{$lte:50}}).pretty() | where likes <= 50 |
| Greater Than | {<key>:{$gt:<value>}} | db.mycol.find({"likes":{$gt:50}}).pretty() | where likes > 50 |
| Greater Than Equals | {<key>:{$gte:<value>}} | db.mycol.find({"likes":{$gte:50}}).pretty() | where likes >= 50 |
| Not Equals | {<key>:{$ne:<value>}} | db.mycol.find({"likes":{$ne:50}}).pretty() | where likes != 50 |
| Values in an array | {<key>:{$in:[<value1>, <value2>,……<valueN>]}} | db.mycol.find({"name":{$in:["Raj", "Ram", "Raghu"]}}).pretty() | Where name matches any of the value in :["Raj", "Ram", "Raghu"] |
| Values not in an array | {<key>:{$nin:<value>}} | db.mycol.find({"name":{$nin:["Ramu", "Raghav"]}}).pretty() | Where name values is not in the array :["Ramu", "Raghav"] or, doesn’t exist at all |

AND in MongoDB

Syntax

To query documents based on the AND condition, you need to use $and keyword. Following is the basic syntax of AND −

>db.mycol.find({ $and: [ {<key1>:<value1>}, { <key2>:<value2>} ] })

Example

Following example will show all the tutorials written by 'tutorials point' and whose title is 'MongoDB Overview'.

> db.mycol.find({$and:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pretty()

{

"\_id" : ObjectId("5dd4e2cc0821d3b44607534c"),

"title" : "MongoDB Overview",

"description" : "MongoDB is no SQL database",

"by" : "tutorials point",

"url" : "http://www.tutorialspoint.com",

"tags" : [

"mongodb",

"database",

"NoSQL"

],

"likes" : 100

}

>

For the above given example, equivalent where clause will be ' where by = 'tutorials point' AND title = 'MongoDB Overview' '. You can pass any number of key, value pairs in find clause.

OR in MongoDB

Syntax

To query documents based on the OR condition, you need to use $or keyword. Following is the basic syntax of OR −

>db.mycol.find(

{

$or: [

{key1: value1}, {key2:value2}

]

}

).pretty()

Example

Following example will show all the tutorials written by 'tutorials point' or whose title is 'MongoDB Overview'.

>db.mycol.find({$or:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pretty()

{

"\_id": ObjectId(7df78ad8902c),

"title": "MongoDB Overview",

"description": "MongoDB is no sql database",

"by": "tutorials point",

"url": "http://www.tutorialspoint.com",

"tags": ["mongodb", "database", "NoSQL"],

"likes": "100"

}

>

Using AND and OR Together

Example

The following example will show the documents that have likes greater than 10 and whose title is either 'MongoDB Overview' or by is 'tutorials point'. Equivalent SQL where clause is 'where likes>10 AND (by = 'tutorials point' OR title = 'MongoDB Overview')'

>db.mycol.find({"likes": {$gt:10}, $or: [{"by": "tutorials point"},

{"title": "MongoDB Overview"}]}).pretty()

{

"\_id": ObjectId(7df78ad8902c),

"title": "MongoDB Overview",

"description": "MongoDB is no sql database",

"by": "tutorials point",

"url": "http://www.tutorialspoint.com",

"tags": ["mongodb", "database", "NoSQL"],

"likes": "100"

}

>

NOR in MongoDB

Syntax

To query documents based on the NOT condition, you need to use $not keyword. Following is the basic syntax of NOT −

>db.COLLECTION\_NAME.find(

{

$not: [

{key1: value1}, {key2:value2}

]

}

)

Example

Assume we have inserted 3 documents in the collection empDetails as shown below −

db.empDetails.insertMany(

[

{

First\_Name: "Radhika",

Last\_Name: "Sharma",

Age: "26",

e\_mail: "radhika\_sharma.123@gmail.com",

phone: "9000012345"

},

{

First\_Name: "Rachel",

Last\_Name: "Christopher",

Age: "27",

e\_mail: "Rachel\_Christopher.123@gmail.com",

phone: "9000054321"

},

{

First\_Name: "Fathima",

Last\_Name: "Sheik",

Age: "24",

e\_mail: "Fathima\_Sheik.123@gmail.com",

phone: "9000054321"

}

]

)

Following example will retrieve the document(s) whose first name is not "Radhika" and last name is not "Christopher"

> db.empDetails.find(

{

$nor:[

{"First\_Name": "Radhika"},

{"Last\_Name": "Christopher"}

]

}

).pretty()

{

"\_id" : ObjectId("5dd631f270fb13eec3963bef"),

"First\_Name" : "Fathima",

"Last\_Name" : "Sheik",

"Age" : "24",

"e\_mail" : "Fathima\_Sheik.123@gmail.com",

"phone" : "9000054321"

}

NOT in MongoDB

Syntax

To query documents based on the NOT condition, you need to use $not keyword following is the basic syntax of NOT −

>db.COLLECTION\_NAME.find(

{

$NOT: [

{key1: value1}, {key2:value2}

]

}

).pretty()

Example

Following example will retrieve the document(s) whose age is not greater than 25

> db.empDetails.find( { "Age": { $not: { $gt: "25" } } } )

{

"\_id" : ObjectId("5dd6636870fb13eec3963bf7"),

"First\_Name" : "Fathima",

"Last\_Name" : "Sheik",

"Age" : "24",

"e\_mail" : "Fathima\_Sheik.123@gmail.com",

"phone" : "9000054321"

}

MongoDB's update() and save() methods are used to update document into a collection. The update() method updates the values in the existing document while the save() method replaces the existing document with the document passed in save() method.

MongoDB Update() Method

The update() method updates the values in the existing document.

Syntax

The basic syntax of update() method is as follows −

>db.COLLECTION\_NAME.update(SELECTION\_CRITERIA, UPDATED\_DATA)

Example

Consider the mycol collection has the following data.

{ "\_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}

{ "\_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}

{ "\_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}

Following example will set the new title 'New MongoDB Tutorial' of the documents whose title is 'MongoDB Overview'.

>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB Tutorial'}})

WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

>db.mycol.find()

{ "\_id" : ObjectId(5983548781331adf45ec5), "title":"New MongoDB Tutorial"}

{ "\_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}

{ "\_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}

>

By default, MongoDB will update only a single document. To update multiple documents, you need to set a parameter 'multi' to true.

>db.mycol.update({'title':'MongoDB Overview'},

{$set:{'title':'New MongoDB Tutorial'}},{multi:true})

MongoDB Save() Method

The save() method replaces the existing document with the new document passed in the save() method.

Syntax

The basic syntax of MongoDB save() method is shown below −

>db.COLLECTION\_NAME.save({\_id:ObjectId(),NEW\_DATA})

Example

Following example will replace the document with the \_id '5983548781331adf45ec5'.

>db.mycol.save(

{

"\_id" : ObjectId("507f191e810c19729de860ea"),

"title":"Tutorials Point New Topic",

"by":"Tutorials Point"

}

)

WriteResult({

"nMatched" : 0,

"nUpserted" : 1,

"nModified" : 0,

"\_id" : ObjectId("507f191e810c19729de860ea")

})

>db.mycol.find()

{ "\_id" : ObjectId("507f191e810c19729de860e6"), "title":"Tutorials Point New Topic",

"by":"Tutorials Point"}

{ "\_id" : ObjectId("507f191e810c19729de860e6"), "title":"NoSQL Overview"}

{ "\_id" : ObjectId("507f191e810c19729de860e6"), "title":"Tutorials Point Overview"}

>

MongoDB findOneAndUpdate() method

The findOneAndUpdate() method updates the values in the existing document.

Syntax

The basic syntax of findOneAndUpdate() method is as follows −

>db.COLLECTION\_NAME.findOneAndUpdate(SELECTIOIN\_CRITERIA, UPDATED\_DATA)

Example

Assume we have created a collection named empDetails and inserted three documents in it as shown below −

> db.empDetails.insertMany(

[

{

First\_Name: "Radhika",

Last\_Name: "Sharma",

Age: "26",

e\_mail: "radhika\_sharma.123@gmail.com",

phone: "9000012345"

},

{

First\_Name: "Rachel",

Last\_Name: "Christopher",

Age: "27",

e\_mail: "Rachel\_Christopher.123@gmail.com",

phone: "9000054321"

},

{

First\_Name: "Fathima",

Last\_Name: "Sheik",

Age: "24",

e\_mail: "Fathima\_Sheik.123@gmail.com",

phone: "9000054321"

}

]

)

Following example updates the age and email values of the document with name 'Radhika'.

> db.empDetails.findOneAndUpdate(

{First\_Name: 'Radhika'},

{ $set: { Age: '30',e\_mail: 'radhika\_newemail@gmail.com'}}

)

{

"\_id" : ObjectId("5dd6636870fb13eec3963bf5"),

"First\_Name" : "Radhika",

"Last\_Name" : "Sharma",

"Age" : "30",

"e\_mail" : "radhika\_newemail@gmail.com",

"phone" : "9000012345"

}

MongoDB updateOne() method

This methods updates a single document which matches the given filter.

Syntax

The basic syntax of updateOne() method is as follows −

>db.COLLECTION\_NAME.updateOne(<filter>, <update>)

Example

> db.empDetails.updateOne(

{First\_Name: 'Radhika'},

{ $set: { Age: '30',e\_mail: 'radhika\_newemail@gmail.com'}}

)

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }

>

MongoDB updateMany() method

The updateMany() method updates all the documents that matches the given filter.

Syntax

The basic syntax of updateMany() method is as follows −

>db.COLLECTION\_NAME.update(<filter>, <update>)

Example

> db.empDetails.updateMany(

{Age:{ $gt: "25" }},

{ $set: { Age: '00'}}

)

{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }

You can see the updated values if you retrieve the contents of the document using the find method as shown below −

> db.empDetails.find()

{ "\_id" : ObjectId("5dd6636870fb13eec3963bf5"), "First\_Name" : "Radhika", "Last\_Name" : "Sharma", "Age" : "00", "e\_mail" : "radhika\_newemail@gmail.com", "phone" : "9000012345" }

{ "\_id" : ObjectId("5dd6636870fb13eec3963bf6"), "First\_Name" : "Rachel", "Last\_Name" : "Christopher", "Age" : "00", "e\_mail" : "Rachel\_Christopher.123@gmail.com", "phone" : "9000054321" }

{ "\_id" : ObjectId("5dd6636870fb13eec3963bf7"), "First\_Name" : "Fathima", "Last\_Name" : "Sheik", "Age" : "24", "e\_mail" : "Fathima\_Sheik.123@gmail.com", "phone" : "9000054321" }

>

The remove() Method

MongoDB's remove() method is used to remove a document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag.

deletion criteria − (Optional) deletion criteria according to documents will be removed.

justOne − (Optional) if set to true or 1, then remove only one document.

Syntax

Basic syntax of remove() method is as follows −

>db.COLLECTION\_NAME.remove(DELLETION\_CRITTERIA)

Example

Consider the mycol collection has the following data.

{\_id : ObjectId("507f191e810c19729de860e1"), title: "MongoDB Overview"},

{\_id : ObjectId("507f191e810c19729de860e2"), title: "NoSQL Overview"},

{\_id : ObjectId("507f191e810c19729de860e3"), title: "Tutorials Point Overview"}

Following example will remove all the documents whose title is 'MongoDB Overview'.

>db.mycol.remove({'title':'MongoDB Overview'})

WriteResult({"nRemoved" : 1})

> db.mycol.find()

{"\_id" : ObjectId("507f191e810c19729de860e2"), "title" : "NoSQL Overview" }

{"\_id" : ObjectId("507f191e810c19729de860e3"), "title" : "Tutorials Point Overview" }

Remove Only One

If there are multiple records and you want to delete only the first record, then set justOne parameter in remove() method.

>db.COLLECTION\_NAME.remove(DELETION\_CRITERIA,1)

Remove All Documents

If you don't specify deletion criteria, then MongoDB will delete whole documents from the collection. This is equivalent of SQL's truncate command.

> db.mycol.remove({})

WriteResult({ "nRemoved" : 2 })

> db.mycol.find()

>

MongoDB – Projection :-

In MongoDB, projection means selecting only the necessary data rather than selecting whole of the data of a document. If a document has 5 fields and you need to show only 3, then select only 3 fields from them.

The find() Method

MongoDB's find() method, explained in [MongoDB Query Document](https://www.tutorialspoint.com/mongodb/mongodb_query_document.htm#_blank) accepts second optional parameter that is list of fields that you want to retrieve. In MongoDB, when you execute find() method, then it displays all fields of a document. To limit this, you need to set a list of fields with value 1 or 0. 1 is used to show the field while 0 is used to hide the fields.

Syntax

The basic syntax of find() method with projection is as follows −

>db.COLLECTION\_NAME.find({},{KEY:1})

Example

Consider the collection mycol has the following data −

{\_id : ObjectId("507f191e810c19729de860e1"), title: "MongoDB Overview"},

{\_id : ObjectId("507f191e810c19729de860e2"), title: "NoSQL Overview"},

{\_id : ObjectId("507f191e810c19729de860e3"), title: "Tutorials Point Overview"}

Following example will display the title of the document while querying the document.

>db.mycol.find({},{"title":1,\_id:0})

{"title":"MongoDB Overview"}

{"title":"NoSQL Overview"}

{"title":"Tutorials Point Overview"}

>

Please note \_id field is always displayed while executing find() method, if you don't want this field, then you need to set it as 0.

In this chapter, we will learn how to limit records using MongoDB.

The Limit() Method

To limit the records in MongoDB, you need to use limit() method. The method accepts one number type argument, which is the number of documents that you want to be displayed.

Syntax

The basic syntax of limit() method is as follows −

>db.COLLECTION\_NAME.find().limit(NUMBER)

Example

Consider the collection myycol has the following data.

{\_id : ObjectId("507f191e810c19729de860e1"), title: "MongoDB Overview"},

{\_id : ObjectId("507f191e810c19729de860e2"), title: "NoSQL Overview"},

{\_id : ObjectId("507f191e810c19729de860e3"), title: "Tutorials Point Overview"}

Following example will display only two documents while querying the document.

>db.mycol.find({},{"title":1,\_id:0}).limit(2)

{"title":"MongoDB Overview"}

{"title":"NoSQL Overview"}

>

If you don't specify the number argument in limit() method then it will display all documents from the collection.

MongoDB Skip() Method

Apart from limit() method, there is one more method skip() which also accepts number type argument and is used to skip the number of documents.

Syntax

The basic syntax of skip() method is as follows −

>db.COLLECTION\_NAME.find().limit(NUMBER).skip(NUMBER)

Example

Following example will display only the second document.

>db.mycol.find({},{"title":1,\_id:0}).limit(1).skip(1)

{"title":"NoSQL Overview"}

>

Please note, the default value in skip() method is 0.

The sort() Method

To sort documents in MongoDB, you need to use sort() method. The method accepts a document containing a list of fields along with their sorting order. To specify sorting order 1 and -1 are used. 1 is used for ascending order while -1 is used for descending order.

Syntax

The basic syntax of sort() method is as follows −

>db.COLLECTION\_NAME.find().sort({KEY:1})

Example

Consider the collection myycol has the following data.

{\_id : ObjectId("507f191e810c19729de860e1"), title: "MongoDB Overview"}

{\_id : ObjectId("507f191e810c19729de860e2"), title: "NoSQL Overview"}

{\_id : ObjectId("507f191e810c19729de860e3"), title: "Tutorials Point Overview"}

Following example will display the documents sorted by title in the descending order.

>db.mycol.find({},{"title":1,\_id:0}).sort({"title":-1})

{"title":"Tutorials Point Overview"}

{"title":"NoSQL Overview"}

{"title":"MongoDB Overview"}

>

Please note, if you don't specify the sorting preference, then sort() method will display the documents in ascending order.

MongoDB – Indexing:-

Indexes support the efficient resolution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. This scan is highly inefficient and require MongoDB to process a large volume of data.

Indexes are special data structures, that store a small portion of the data set in an easy-to-traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index.

The createIndex() Method

To create an index, you need to use createIndex() method of MongoDB.

Syntax

The basic syntax of createIndex() method is as follows().

>db.COLLECTION\_NAME.createIndex({KEY:1})

Here key is the name of the field on which you want to create index and 1 is for ascending order. To create index in descending order you need to use -1.

Example

>db.mycol.createIndex({"title":1})

{

"createdCollectionAutomatically" : false,

"numIndexesBefore" : 1,

"numIndexesAfter" : 2,

"ok" : 1

}

>

In createIndex() method you can pass multiple fields, to create index on multiple fields.

>db.mycol.createIndex({"title":1,"description":-1})

>

This method also accepts list of options (which are optional). Following is the list −

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| background | Boolean | Builds the index in the background so that building an index does not block other database activities. Specify true to build in the background. The default value is false. |
| unique | Boolean | Creates a unique index so that the collection will not accept insertion of documents where the index key or keys match an existing value in the index. Specify true to create a unique index. The default value is false. |
| name | string | The name of the index. If unspecified, MongoDB generates an index name by concatenating the names of the indexed fields and the sort order. |
| sparse | Boolean | If true, the index only references documents with the specified field. These indexes use less space but behave differently in some situations (particularly sorts). The default value is false. |
| expireAfterSeconds | integer | Specifies a value, in seconds, as a TTL to control how long MongoDB retains documents in this collection. |
| weights | document | The weight is a number ranging from 1 to 99,999 and denotes the significance of the field relative to the other indexed fields in terms of the score. |
| default\_language | string | For a text index, the language that determines the list of stop words and the rules for the stemmer and tokenizer. The default value is English. |
| language\_override | string | For a text index, specify the name of the field in the document that contains, the language to override the default language. The default value is language. |

The dropIndex() method

You can drop a particular index using the dropIndex() method of MongoDB.

Syntax

The basic syntax of DropIndex() method is as follows().

>db.COLLECTION\_NAME.dropIndex({KEY:1})

Here key is the name of the file on which you want to create index and 1 is for ascending order. To create index in descending order you need to use -1.

Example

> db.mycol.dropIndex({"title":1})

{

"ok" : 0,

"errmsg" : "can't find index with key: { title: 1.0 }",

"code" : 27,

"codeName" : "IndexNotFound"

}

The dropIndexes() method

This method deletes multiple (specified) indexes on a collection.

Syntax

The basic syntax of DropIndexes() method is as follows() −

>db.COLLECTION\_NAME.dropIndexes()

Example

Assume we have created 2 indexes in the named mycol collection as shown below −

> db.mycol.createIndex({"title":1,"description":-1})

Following example removes the above created indexes of mycol −

>db.mycol.dropIndexes({"title":1,"description":-1})

{ "nIndexesWas" : 2, "ok" : 1 }

>

The getIndexes() method

This method returns the description of all the indexes int the collection.

Syntax

Following is the basic syntax od the getIndexes() method −

db.COLLECTION\_NAME.getIndexes()

Example

Assume we have created 2 indexes in the named mycol collection as shown below −

> db.mycol.createIndex({"title":1,"description":-1})

Following example retrieves all the indexes in the collection mycol −

> db.mycol.getIndexes()

[

{

"v" : 2,

"key" : {

"\_id" : 1

},

"name" : "\_id\_",

"ns" : "test.mycol"

},

{

"v" : 2,

"key" : {

"title" : 1,

"description" : -1

},

"name" : "title\_1\_description\_-1",

"ns" : "test.mycol"

}

]

>

MongoDB - Aggregation

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. In SQL count(\*) and with group by is an equivalent of MongoDB aggregation.

The aggregate() Method

For the aggregation in MongoDB, you should use aggregate() method.

Syntax

Basic syntax of aggregate() method is as follows −

>db.COLLECTION\_NAME.aggregate(AGGREGATE\_OPERATION)

Example

In the collection you have the following data −

{

\_id: ObjectId(7df78ad8902c)

title: 'MongoDB Overview',

description: 'MongoDB is no sql database',

by\_user: 'tutorials point',

url: 'http://www.tutorialspoint.com',

tags: ['mongodb', 'database', 'NoSQL'],

likes: 100

},

{

\_id: ObjectId(7df78ad8902d)

title: 'NoSQL Overview',

description: 'No sql database is very fast',

by\_user: 'tutorials point',

url: 'http://www.tutorialspoint.com',

tags: ['mongodb', 'database', 'NoSQL'],

likes: 10

},

{

\_id: ObjectId(7df78ad8902e)

title: 'Neo4j Overview',

description: 'Neo4j is no sql database',

by\_user: 'Neo4j',

url: 'http://www.neo4j.com',

tags: ['neo4j', 'database', 'NoSQL'],

likes: 750

},

Now from the above collection, if you want to display a list stating how many tutorials are written by each user, then you will use the following aggregate() method −

> db.mycol.aggregate([{$group : {\_id : "$by\_user", num\_tutorial : {$sum : 1}}}])

{ "\_id" : "tutorials point", "num\_tutorial" : 2 }

{ "\_id" : "Neo4j", "num\_tutorial" : 1 }

>

Sql equivalent query for the above use case will be select by\_user, count(\*) from mycol group by by\_user.

In the above example, we have grouped documents by field by\_user and on each occurrence of by user previous value of sum is incremented. Following is a list of available aggregation expressions.

|  |  |  |
| --- | --- | --- |
| Expression | Description | Example |
| $sum | Sums up the defined value from all documents in the collection. | db.mycol.aggregate([{$group : {\_id : "$by\_user", num\_tutorial : {$sum : "$likes"}}}]) |
| $avg | Calculates the average of all given values from all documents in the collection. | db.mycol.aggregate([{$group : {\_id : "$by\_user", num\_tutorial : {$avg : "$likes"}}}]) |
| $min | Gets the minimum of the corresponding values from all documents in the collection. | db.mycol.aggregate([{$group : {\_id : "$by\_user", num\_tutorial : {$min : "$likes"}}}]) |
| $max | Gets the maximum of the corresponding values from all documents in the collection. | db.mycol.aggregate([{$group : {\_id : "$by\_user", num\_tutorial : {$max : "$likes"}}}]) |
| $push | Inserts the value to an array in the resulting document. | db.mycol.aggregate([{$group : {\_id : "$by\_user", url : {$push: "$url"}}}]) |
| $addToSet | Inserts the value to an array in the resulting document but does not create duplicates. | db.mycol.aggregate([{$group : {\_id : "$by\_user", url : {$addToSet : "$url"}}}]) |
| $first | Gets the first document from the source documents according to the grouping. Typically this makes only sense together with some previously applied “$sort”-stage. | db.mycol.aggregate([{$group : {\_id : "$by\_user", first\_url : {$first : "$url"}}}]) |
| $last | Gets the last document from the source documents according to the grouping. Typically this makes only sense together with some previously applied “$sort”-stage. | db.mycol.aggregate([{$group : {\_id : "$by\_user", last\_url : {$last : "$url"}}}]) |

Pipeline Concept

In UNIX command, shell pipeline means the possibility to execute an operation on some input and use the output as the input for the next command and so on. MongoDB also supports same concept in aggregation framework. There is a set of possible stages and each of those is taken as a set of documents as an input and produces a resulting set of documents (or the final resulting JSON document at the end of the pipeline). This can then in turn be used for the next stage and so on.

Following are the possible stages in aggregation framework −

$project − Used to select some specific fields from a collection.

$match − This is a filtering operation and thus this can reduce the amount of documents that are given as input to the next stage.

$group − This does the actual aggregation as discussed above.

$sort − Sorts the documents.

$skip − With this, it is possible to skip forward in the list of documents for a given amount of documents.

$limit − This limits the amount of documents to look at, by the given number starting from the current positions.

$unwind − This is used to unwind document that are using arrays. When using an array, the data is kind of pre-joined and this operation will be undone with this to have individual documents again. Thus with this stage we will increase the amount of documents for the next stage.

MongoDB - Sharding

Sharding is the process of storing data records across multiple machines and it is MongoDB's approach to meeting the demands of data growth. As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput. Sharding solves the problem with horizontal scaling. With sharding, you add more machines to support data growth and the demands of read and write operations.

Why Sharding?

In replication, all writes go to master node

Latency sensitive queries still go to master

Single replica set has limitation of 12 nodes

Memory can't be large enough when active dataset is big

Local disk is not big enough

Vertical scaling is too expensive

Sharding in MongoDB

The following diagram shows the Sharding in MongoDB using sharded cluster.

Diagram

Description automatically generated

In the following diagram, there are three main components −

Shards − Shards are used to store data. They provide high availability and data consistency. In production environment, each shard is a separate replica set.

Config Servers − Config servers store the cluster's metadata. This data contains a mapping of the cluster's data set to the shards. The query router uses this metadata to target operations to specific shards. In production environment, sharded clusters have exactly 3 config servers.

Query Routers − Query routers are basically mongo instances, interface with client applications and direct operations to the appropriate shard. The query router processes and targets the operations to shards and then returns results to the clients. A sharded cluster can contain more than one query router to divide the client request load. A client sends requests to one query router. Generally, a sharded cluster have many query routers.

MongoDB - Relationships

Relationships in MongoDB represent how various documents are logically related to each other. Relationships can be modeled via Embedded and Referenced approaches. Such relationships can be either 1:1, 1:N, N:1 or N:N.

Let us consider the case of storing addresses for users. So, one user can have multiple addresses making this a 1:N relationship.

Following is the sample document structure of user document −

{

"\_id":ObjectId("52ffc33cd85242f436000001"),

"name": "Tom Hanks",

"contact": "987654321",

"dob": "01-01-1991"

}

Following is the sample document structure of address document −

{

"\_id":ObjectId("52ffc4a5d85242602e000000"),

"building": "22 A, Indiana Apt",

"pincode": 123456,

"city": "Los Angeles",

"state": "California"

}

Modeling Embedded Relationships

In the embedded approach, we will embed the address document inside the user document.

> db.users.insert({

{

"\_id":ObjectId("52ffc33cd85242f436000001"),

"contact": "987654321",

"dob": "01-01-1991",

"name": "Tom Benzamin",

"address": [

{

"building": "22 A, Indiana Apt",

"pincode": 123456,

"city": "Los Angeles",

"state": "California"

},

{

"building": "170 A, Acropolis Apt",

"pincode": 456789,

"city": "Chicago",

"state": "Illinois"

}

]

}

})

This approach maintains all the related data in a single document, which makes it easy to retrieve and maintain. The whole document can be retrieved in a single query such as −

>db.users.findOne({"name":"Tom Benzamin"},{"address":1})

Note that in the above query, db and users are the database and collection respectively.

The drawback is that if the embedded document keeps on growing too much in size, it can impact the read/write performance.

<https://www.tutorialspoint.com/mongodb/>

Oracle Sql Queries

Sql Queries interview questions :-

q) Which operator is used in query for pattern matching?

LIKE operator is used for pattern matching, and it can be used as -.

1. % - Matches zero or more characters.

2. \_(Underscore) – Matching exactly one character.

Example -.

Select \* from Student where studentname like 'a%'

Select \* from Student where studentname like 'ami\_'

q) What is the difference between TRUNCATE and DROP statements?

TRUNCATE removes all the rows from the table, and it cannot be rolled back. DROP command removes a table from the database and operation cannot be rolled back.

Below is simple query to find the employee whose salary is highest.

select max(salary) from Employee

SELECT \* FROM employee where salary =(select max(salary) from Employee)

select \* from test\_data ;

-- finding highest maths number

select max(maths) as math from test\_data ;

-- finding second highest maths number

select max(maths) as math from test\_data where maths < (select max(maths) as math from test\_data);

-- -- finding third highest maths number

select \* from (select \* from (select \* from test\_data order by maths desc) where ROWNUM <= 3 order by maths asc) where rownum <=1 ;

SELECT student\_name, maths FROM test\_data e1 WHERE 3-1 = (SELECT COUNT(DISTINCT maths) FROM test\_data e2 WHERE e2.maths > e1.maths);

--- find all count off all student whose name is same

select student\_name , count(\*) from (select \* from test\_data order by maths desc) group by student\_name ;

-- -- finding third highest maths number

SELECT \* FROM ( SELECT e.\*, ROW\_NUMBER() OVER (ORDER BY maths DESC) rn FROM test\_data e ) WHERE rn = 3;

SELECT student\_name, maths FROM test\_data e1 WHERE 3-1 = (SELECT COUNT(DISTINCT maths) FROM test\_data e2 WHERE e2.maths > e1.maths);

SELECT e.\*, ROW\_NUMBER() OVER (ORDER BY maths DESC) rn FROM test\_data e;

--- add new column

alter table test\_data add (departmentId number(4));

--SQL Query to find Max Salary from each department.

select departmentid , max(maths) from test\_data group by departmentId ;

--Write SQL Query to find duplicate rows based on student name in a database? and then write SQL query to delete them?

select \* from test\_data where (student\_name) in (select e.student\_name from test\_data e group by e.student\_name having count(\*)>1 ) order by student\_name;

DELETE FROM (select \* from test\_data where (student\_name) in (select e.student\_name from test\_data e group by e.student\_name having count(\*)>1 ) order by student\_name) ;

SELECT \*

FROM TABLE A

WHERE EXISTS (

SELECT 1 FROM TABLE

WHERE COLUMN\_NAME = A.COLUMN\_NAME

AND ROWID < A.ROWID

)

<https://artoftesting.com/sql-queries-for-interview>

<https://www.complexsql.com/complex-sql-queries-examples-with-answers/complex-sql-for-interviews/>

MySql---------------------

Table

Description automatically generated

**Q. SQL Query to Find All Employees Who Are Also Managers**

select \* from Employees e where e.employee\_id in ( select manager\_id from Employees );

**Q. SQL Query to Find All Employees Who Are Not Managers**

select \* from Employees e where e.employee\_id not in ( select manager\_id from Employees );

**Q. How to find Nth highest salary from a table**

select sl.salary from (select distinct e.salary from Employee as e order by e.salary desc limit 3) as sl order by sl.salary

asc limit 1;

select \* from (select \* from employees order by salary desc limit 3) as e1 order by e1.salary asc limit 1 ;

**Q. Displaying Department Name Having Least Number of Employees**

select e.department,count(\*) as employeeCount from Employee e group by e.department order by employeeCount asc limit 1;

**Q. Displaying Department Name Having Max Number of Employees**

select e.department,count(\*) as employeeCount from Employee e group by e.department order by employeeCount desc limit 1;

[ select grp1.department , grp1.employeeCount from ( select e.department,count(\*) as employeeCount from Employee e group by e.department order by employeeCount desc

) grp1 where grp1.employeeCount = (select max(grp.employeeCount) as mx from ( select e.department,count(\*) as employeeCount from Employee e group by e.department order by employeeCount desc ) as grp) ]

**Q. SQL Query to Find Names of the Employees Whose Department Have Number of Employees Less equal 2**

select e.name from employee e where e.department in ( select e1.department from employee e1 group by e1.department having count(\*)<=2 );

**Q. SQL Query to Find Number of Employees According to Gender Whose DOB is Between a Given Range**

SELECT Gender,count(Gender) FROM department

WHERE DateOfBirth between '1995-01-01' and '1996-12-31'

GROUP BY gender;

**Q. SQL Self Join Example - SQL Query to Find Employees Earning More Than Managers - LeetCode Solution.**

select e1.name from employees e1 join employees e2 on e1.employee\_id = e2.manager\_id where e1.salary> e2.salary ;

**Q. print the name of the employee and their manager :**

SELECT e1.name as Manager, e2.name FROM Employees e1 JOIN Employees e2 ON e1.employee\_id = e2.Manager\_Id;

SELECT e1.name, e2.name as Manager FROM Employees e1 , Employees e2 where e1.manager\_id =e2.employee\_id;

**Q. print all employee who are managers :**

SELECT distinct b.name FROM Employees a, Employees b WHERE a.manager\_id = b.employee\_id ;

**Q. Count All Employees Under Each Manager**

SELECT sup.employee\_id,sup.name,count(sub.employee\_id) AS number\_of\_employees FROM employees sub

JOIN employees sup ON sub.manager\_id = sup.employee\_id GROUP BY sup.employee\_id, sup.name;

**Q. SQL Query to Find the Highest Salary of Each Department**

SELECT DEPT\_ID, MAX(SALARY) FROM department1 GROUP BY DEPT\_ID;

**Q. How to Get the Highest Salary of Each Department?**

SELECT DepartmentID, MAX(Salary)

FROM Employee

GROUP BY DepartmentID

SELECT DepartmentName, MAX(Salary)

FROM Employee e RIGHT JOIN Department d ON e.DepartmentId = d.DepartmentID

GROUP BY DepartmentName

SELECT

Department.name AS 'Department',

Employee.name AS 'Employee',

Salary

FROM

Employee

JOIN

Department ON Employee.DepartmentId = Department.Id

WHERE

(Employee.DepartmentId , Salary) IN

( SELECT

DepartmentId, MAX(Salary)

FROM

Employee

GROUP BY DepartmentId

);

**Q. Employees with higher salary than their department average? [duplicate]**

SELECT name, department\_id, salary

FROM employees e

WHERE salary > (select avg(salary) from employees e2 where e2.department\_id = e.department\_id);

**Q. SQL query :- detail of employees whose salary is greater than average salary in respective departments**

SELECT empno, ename, deptno

FROM emp e

WHERE sal > (SELECT AVG(sal)

FROM emp e1

WHERE e.deptno = e1.deptno

GROUP BY deptno)

ORDER BY deptno;

Select empname

From employee emp

Join (select avg(salary) as bucks, deptid

from employee group by deptid) dept

On dept.deptid = emp.deptid

And dept.bucks

SELECT DEPARTMENT\_NAME,AVG(SALARY) AS

AVERAGE\_SALARY FROM COMPANY GROUP BY DEPARTMENT\_NAME;

**Q. Find employees who earn more than the average salary within their department**

select last\_name, salary, department\_id from employees o

where salary >(select avg(salary) from employees e where e.department\_id =o.department\_id);

**Q. List of employees who are not managers**

SELECT E.LAST\_NAME,E.EMPLOYEE\_ID FROM EMPLOYEES EWHERE EMPLOYEE\_ID not in

(SELECT MANAGER\_ID FROM EMPLOYEES where MANAGER\_ID is not null)

**Q. How to remove duplicate rows from table?**

**1. Delete Duplicate Record Using Delete Join**

**DELETE S1 FROM student\_contacts AS S1**

**INNER JOIN student\_contacts AS S2**

**WHERE S1.id < S2.id AND S1.email = S2.email;**

**2. now check all duplicate records from the table.**

**SELECT name, email, COUNT(name)**

**FROM student\_contacts**

**GROUP BY name**

**HAVING COUNT(name) > 1;**

<https://www.complexsql.com/complex-sql-queries-examples-with-answers/complex-sql-for-interviews/>

<https://www.geeksforgeeks.org/sql-query-to-find-all-employees-who-are-also-managers/>