1. A picture containing graphical user interface

   Description automatically generated
2. **Jatin** 🡺 Before Reading Further, some useful info:
   1. Kafka Connect is a framework for connecting Kafka with External Systems (Producer to Kafka or Kafka to Consumer).
   2. Kafka Connect Cluster is a different cluster from the Kafka Cluster.
   3. Kafka Connectors components supporting reading/writing between external systems.
3. **Agenda**:  
   In earlier lecture, we talked about the five components of **Kafka Ecosystem**.

Graphical user interface, text, application

Description automatically generated

1. Text

   Description automatically generated
2. **What is Kafka Connect?**
   1. We already studied in earlier lecture that Kafka was initially developed at **LinkedIn** to solve the **Data Integration** problem. Following is the diagram.  
      Diagram

      Description automatically generated
   2. The point is very straightforward. We can’t produce a single standalone app that does everything in an enterprise. Why?   
      Because that is practically impossible and not economical.  
      So, in a typical enterprise, you may have a bunch of independent running applications.  
      Some of them are “**custom designed and developed in-house**”.  
      Others may have been purchased from a third-party app vendors.  
      Few of the applications maybe running outside of the organization boundary and maintained by the **partners or the service providers**.  
      They all might be generating some data and owning it.   
      However, they might also need some **additional data** which is created and **owned by other systems**.  
      **For Example**:
      1. Financial Accounting Software needs data from an invoicing system.
      2. Inventory needs data from Shipment, Invoicing & Warehouse.

Diagram

Description automatically generated

* 1. Analytics would need data from all over the enterprise.  
     So, **Data Integration is a common problem**.  
     You start solving it but at some point, it becomes **a messy network** of **Data pipelines**.
  2. When you reach that state, it becomes almost impossible to maintain.
  3. LinkedIn created and used **Kafka** to simplify **these pipelines**.
  4. So, how they did it?
  5. Let’s look at the **simplified version** of the problem.
     1. You have some app (Say **invoicing app**) there with its DB where the app stores its data.
     2. **Requirement**: To take data from the DB into **Snowflake warehouse**.
     3. You decided to use **Kafka as a broker** because **Kafka** will keep your **Data Integration** simple.  
        Bringing Data from your **invoicing system** to **Kafka System** is **one time activity**.   
        **one time activity (Jatin)**: It means that if five applications need the data, invoice system will not send five times rather just one time to the broker and each app of five apps will ask the broker for the data.
     4. Once the data is in **Kafka Cluster**, you can bring the data into the **Snowflake**.
     5. If some other apps need to consume the data, those can consume from the **Broker**.
     6. All **these pipelines** are going to be **one to one link**.  
        **Jatin**: Even though this is one (**Producer**) to many (**Consumers**) as one **Producer** is sending data to many **consumers** but based on action, **Producer** is triggering **send action** just once for each **message record no matter how many consumers are there**.  
        Diagram

        Description automatically generated
     7. Let’s see if **Broker** is not there what happens and how one to one link is destroyed.
     8. Diagram

        Description automatically generated with medium confidence
  6. Now you want to bring data from **Invoicing System** to **Kafka Cluster**. But how?
  7. So, you want to create a **Producer**?
  8. Let’s try and evaluate that option.
  9. There are **two ways** to create a producer depending on the availability of your **source system**.  
     Chart, waterfall chart

     Description automatically generated
  10. **Case 01:** If you have the **source code** for the invoicing app and it is practically feasible to modify your source app then you can create **Embedded Kafka Producer** using **Kafka Producer API.**
      1. Your **Embedded Kafka Producer** becomes the part of **your source app**.
      2. It runs inside the app and sends **invoices** to the **Kafka Cluster**.
  11. **Case 02:** But if you don’t have source code for the invoicing app or not practical to modify your source code?
      1. Then you can create **Independent Kafka Producer** for reading and writing.
      2. On one side, it connects with the **source app’s DB**, reads the data and sends to the **Kafka Cluster**.  
         It’s up to you to evaluate **these options and make a decision**.
      3. If you happen to settle down to the second option of creating an **Independent Kafka Producer**, then you’re signing up to solve a problem that might have been already solved for you.  
         Yes, you’re right!!!  
         That is where **Kafka Connect** is designed for you.
      4. **Kafka Connect** is a system that you can put b/w the **Data Source and Kafka Cluster**.  
         Then all we do is to configure it to consume data from the source system and send it to the kafka cluster. No need to write a single line of code. Just configure and run and Kafka Connect will do the job for you.  
         Here **Kafka Connect** is called **“Kafka Connect Source Connector”.** When **Kafka Connect** is put b/w the **Kafka Cluster & Consumer,** then it is called **“Kafka Connect Sink Connector”.**Internally, this connector uses **Kafka Producer API & Kafka Consumer API** respectively.  
         A picture containing diagram

         Description automatically generated
      5. Then all you do is to configure it to consume the data from the Data Source & send it to the **Kafka Cluster**.  
         You don’t need to write a single line of code.
      6. Now we need to bring the data from **Kafka Cluster** to the **Snowflake Warehouse.**
      7. Same as we did above. Place **Kafka Connect** b/w **Kafka Cluster & Snowflake** and do configurations. Such connector is called  
         **“Kafka Connect Sink Connector”** which consumes data from the **Kafka Topic** and sinks data to an **external system**.  
         This connector internally uses **Kafka Consumer API**.  
          Chart, waterfall chart

         Description automatically generated  
         Source connector to pull data from the source system and send it to the Kafka Cluster and source connector internally uses the Kafka Producer API and similarly, we use the sink connector to consume data from a topic and sink it to an external system.
      8. **Kafka Connect**: A component of Kafka for connecting and moving data b/w **Kafka and external system**.  
         Chart, diagram

         Description automatically generated

1. **How Kafka Connect works? Time 07:21**
2. Actually, we already saw that **Kafka Connector** is placed b/w **Kafka Cluster** and (**Producer and Consumer**) and then we do just required **configurations** without writing a single line of code and **Kafka Connector** is responsible for reading and writing data from one system to another system.
   1. But how it is possible?
   2. I mean that there are bunch of different systems like:   
      Diagram, schematic

      Description automatically generated
   3. How it is possible for **Kafka Connect** to read and write data without any code?
   4. Well, it is possible if someone else has already written the code for you.  
      Let me explain.
   5. **Kafka developers** made a smart decision and made a **brand-new framework** for implementing **Kafka Connector**.   
      They named it **Kafka Connect Framework** and **open sourced** it.
   6. This **Kafka Connect Framework** allows you to write **connector (Basically Custom Connectors)**.
   7. These **Kafka** **connectors** are implemented in **two flavors**.
      1. **Source Connector**.
      2. **Sink Connector**.
   8. **Kafka Connect Framework** takes care of all **heavy lifting, scalability, fault-tolerance, error handling and a bunch of other things**.

* 1. But as a **connector developer**, you have to implement two **java classes**.  
     Text

     Description automatically generatedThat is all the source and Target system vendors have been doing (implementing)
  2. **Jatin**:
     1. A diagram of a application

        Description automatically generated
  3. **For Example**:Suppose, you want to bring some data from **RDBMS** to **Kafka Cluster**.

Graphical user interface, application

Description automatically generated

1. **Kafka Connect Scalability?**
2. We learnt **Producer/Consumer Scalability** in earlier lecture. We were able to scale the **Producer** by adding more **Producers** to send data in **parallel**.  
   We Scaled **Kafka Cluster** by adding more **Brokers**.  
   We also **partition** the **Kafka Topic**.  
   Similarly, we scaled the **Consumer** by adding more **Consumers** in the **Consumers Group**.  
   See the following diagram for **scalability reminder**.  
   Diagram

   Description automatically generated
   1. But how to **scale Kafka Connect**?
   2. **Kafka Connect** is in itself **a cluster**.
   3. Each **individual unit** in the **Kafka** **Connect Cluster** is called a **Connect Worker**.  
      You can think of it as a group of computers where each computer is running one **Kafka Connect Worker**.  
      Diagram

      Description automatically generated  
      See the left Side **Kafka Connect Cluster**.  
      **W**: means **Worker** (Each computer in the Kafka Connect Cluster is called Worker).  
      **ST**: **Source Task** (We used the word **source** because we’re talking about **Kafka Connect** b/w the **Data Source** means **Producer** and **Kafka Cluster**.  
      Diagram

      Description automatically generated  
      **ST**: In the above diagram is **Source Task**. You can see a bunch of **source tasks**.   
      One task may be to read from a table and another task to read from another table.   
      This way you can play with them the way your **Connector Developer** has designed and allowed.  
      **Jatin**: I’m writing this theory based on this further lecture but maybe **partially wrong**.   
      For example, take **Kafka Connector Cluster** on the **source side**.  
      This **Kafka Connector Cluster** may have
      1. Two **Kafka** **Connectors** one to read from DB table and another to read from a file.
      2. Three **Workers** (Computers).

Now, these two connectors has possibility of using three workers.  
Jatin maybe wrong about this theory 😊

* 1. Let’s talk about the **Kafka Connect Cluster** b/w **Kafka Cluster** & **Consumer**.  
     **ST**: Means **Sink Task**.  
     Similarly, you can have a bunch of Sink Tasks to share the work load. Diagram

     Description automatically generated  
     So, nothing to worry about the scalability. You can play with the number of tasks and scale the Cluster capacity by adding **Workers**.
  2. **Let’s study the following question**.  
     Diagram

     Description automatically generated   
     You can have one **Kafka Connect Cluster** and inside it, you can run as many **Connectors (Source, Sink)** as you want.  
     See the following diagram:  
     Graphical user interface, application

     Description automatically generated  
     In this example, we can have one **Source Connector &** one **Sink Connector** running in the same **Kafka Connect Cluster**.  
     If you **Kafka Connect Cluster** has some availability, you can install **Salesforce Connector** and can start that **Connector** in the **Kafka Connect Cluster**.  
     If your **Kafka Connect Cluster**  is fully utilized, you can scale it by adding more **Workers** & you can do it dynamically without stopping any existing **Connectors**.

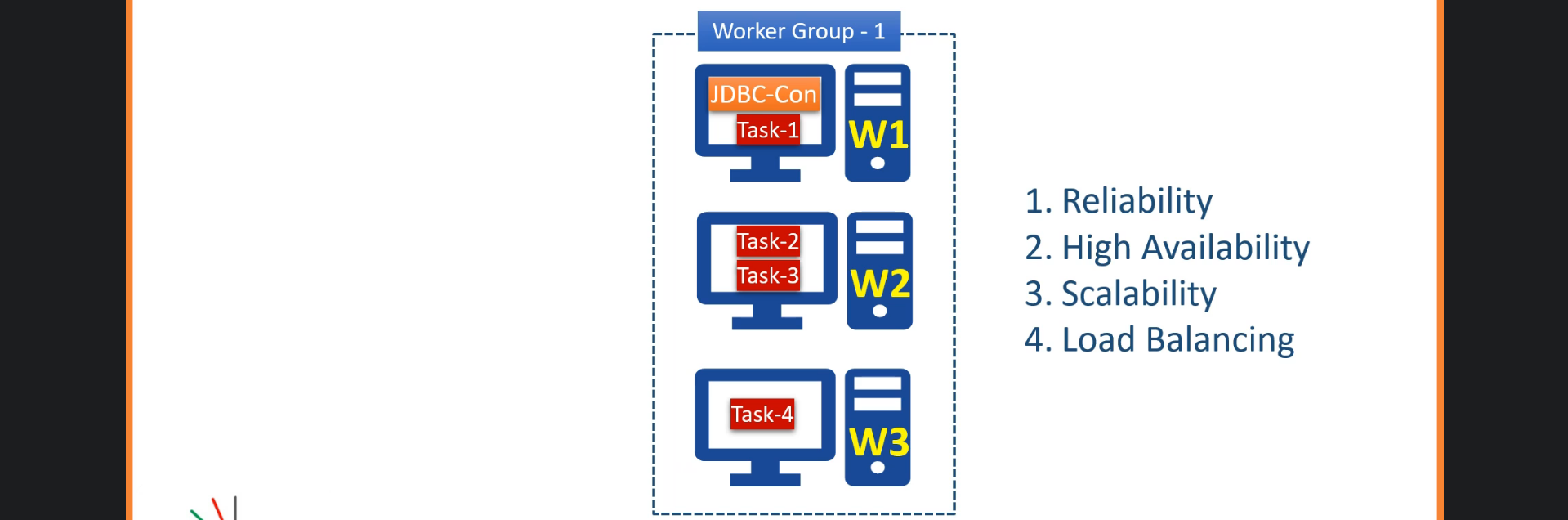
1. **Connect Transformations**?   
     
   Does it **merely copy** data? Or Can we perform some processing or **transformations**?
   1. **Kafka Connect** was designed to perform a plan copy and movement of data b/w **3rd party system** and **Kafka Cluster**.
   2. In both sides (Source and Sink), one side must be a **Kafka Cluster**.  
      Either “Source and Kafka Cluster” or “Kafka Cluster and Destination”
   3. However, **Kafka Connect** also **allows** some fundamental **Single Message Transformations** **(SMTs)**.
   4. What does it mean?
   5. It means you can apply some **transformations** or **changes** to each message on the fly   
      & this is allowed with both (**Source and Sink Connectors**).
   6. Here is a list of **common SMTs**  
      Text

      Description automatically generated
   7. You can **chain** multiple **SMTs** and play with it to restructure your records and route them to a **different topic**.
   8. **NOTE**: However, these **SMTs** are good enough to perform some real life data **validations and transformations**.
2. Now let’s come to the concluding part 🡺 **Kafka Connect Architecture**.
3. **Kafka Connect Architecture**
   1. How does it work?
   2. To understand the **Kafka Architecture**, you have to understand three things.  
      Graphical user interface

      Description automatically generated with medium confidence
   3. You already learnt that **Kafka Connect** is a **Cluster &** it runs one or more **workers**.
   4. Let’s assume you start **Kafka Connect Cluster** with three **Workers.**
   5. These **workers** are **fault tolerant,** and they use **Group ID** to form a **Cluster.**This **Group ID mechanism** is same as **Kafka Consumer Groups.**So, all you need to do is to start each **worker** with same **Group ID** and they will join hands to form a **Kafka Connect Cluster**.  
        
      **Workhorse**: a person who performs **most of the work of a group task**.  
      These **workers** are the main **workhorse** of the **Kafka Connect**.  
      That means they work like a **container process** and they will be responsible for starting & running **Connector & the task**.  
      A picture containing diagram

      Description automatically generated Graphical user interface, application, Teams

      Description automatically generated  
      Similarly, if another **Worker** joins the **Kafka Connect Cluster**, other already existing **Workers** notices it then that **Worker** will assign **tasks and connectors** to it and make sure the load is balanced.  
        
      So, in nutshell, these workers will you give
      1. Reliability.
      2. Availability.
      3. Scalability.
      4. And Load Balancing.



Let’s understand all this with an example:

Suppose you want to just copy the data from **RDBMS**.

So, you download **JDBC Source Connector** & install it within the **Kafka Connect Cluster.  
Installing means:** Its jar files and all its **dependencies** are made available to **Workers** in the **cluster**.  
Then configure the **JDBC Source** **Connector**.  
**Configuration** means providing necessary **configurations** such as

**🡺DB Connection Details**,   
**🡺A list of tables to copy**,   
**🡺frequency to poll the source for new data**,   
**🡺the maximum number of tasks** &   
**🡺many other things** depending on the Connector and requirements.  
All this configuration goes into a file.

**Kafka Connect offers two ways to start a Connector:   
🡺**With **Command Line Tool**.  
🡺**REST APIs** Offered by **Connector**.  
Diagram, application, Teams

Description automatically generated

At this stage, one the Workers will start your **Connector process**. I already told you that **Workers** are like a **Containers**, so they start and run other **processes**. So, this time, it is the **Connector Process**.   
Now, the **Connector process** is responsible for two things.

🡺The first thing is to determine **the degree of** **parallelism**. That means how many parallel tasks, can I start to copy the data from this source.  
You can’t ask this question if you don’t know the **splitting mechanism**.  
So, the first thing is to decide how to **split** “the data copying work”.  
**For example**: Let’s assume that I want to ingest data from five tables. So, I listed the five tables in the **configuration file** and started the **JDBC Connector**. Now, it is quite natural for the **JDBC Connector** to detect the **splitting mechanism**. One table per task 😊 right!!!

So, maximum number of parallelism is five in this case. However, the **splitting logic** for the different Source systems is written in the **Connector Code** (You don’t write but the **Connector Provider** is responsible) and you just give **configuration** options accordingly.

So, you must know **Connector** and configure it accordingly.  
So, the **Connector** knows that it can start five **parallel tasks** and assign one table to each task of copying data from the **Source System**.

**NOTE**: **Connector** is not going to copy data but responsible only for defining and creating a list of   
 tasks.  
 **Each task** will be **configured** to read data from an assigned list of tables.  
Graphical user interface, diagram

Description automatically generated

**Connector** will also include some additional information such as DB connection Details and other things to make sure that the task can operate as an independent process.  
Finally, the list of tasks will be given to the workers and they will start the tasks.  
So, your **tasks** will be **distributed across the workers** for balancing the **Cluster Load**.  
Now task is responsible for the followings:

1. **To connect** with the Data Source.
2. **To poll** the data at regular intervals.
3. **To collect** the records.
4. **To hand over** to the worker.

Task is responsible for only interacting with the External Systems (**Data Source and Target System**).  
The **Source Task** will read data from the **Data Source** then hands over to the Worker and the worker is responsible for sending it to the **Kafka** **Cluster** (**Servers**).   
In case of **Sink Task**, they get the **Kafka Records** from a **Worker** & the **task** is responsible for inserting the records into the **Target System**.  
NOTE: So, **Worker** is responsible for reading from & writing into **Kafka** whereas **Source Task** & **Sink Task** are responsible for reading from and writing into **Data Source** and **Target System.**

**Question**: **Why is it designed so**?

**Answer**: Well, that is **re-usable design**.  
Reading from and writing into a **Kafka Cluster** are standard activities. So, it is taken care by the **Framework**.   
Whereas there are two things which are changing as per **Source and Target Systems**.

🡺How to Split the input for parallel processing? This is taken care by **Connector class**.

🡺How to interact with external systems (**Data Source, Target System**). This is taken care by the **Task Class**.  
These are the two things that Connector Developer needs to take care of.  
Other things which are standards are taken care of by **Kafka Connect Framework** such as

1. Interacting with Kafka to read and write into.
2. Configuration.
3. Errors.
4. Monitoring Connectors & Tasks.
5. Scaling up and down.
6. Handling Failures.

Graphical user interface, diagram

Description automatically generated with medium confidence

**Jatin**:**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Starts\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  
**NOTE**: A **Kafka Connect Cluster** has following components

* + 1. **Source Connector**
    2. **Sink Connector**
    3. **Source Task (ST)**
    4. **Sink Task (ST)**
    5. **Workers**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Ends\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***