Project Milestone-- Data Storage Implementation: KV + relational

- Kafka Connect is a framework that uses Connectors to connect Kafka to other systems including databases, key-value stores, search indexes, and file systems
- Kafka Connectors are ready-to-use components that allow us to import data from external systems into Kafka topics as well as export data from Kafka topics to external systems
- For common data sources and sinks, we can either use existing connection implementations or create our own
- A source connector is a software component that collects data from a system
- Databases, streaming tables, and message brokers are all examples of source systems
- A source connector might also collect metrics from application servers and store them in Kafka topics allowing for low-latency stream processing
- A sink connector sends data from Kafka topics to other systems which could be indexes like Elasticsearch, batch systems like Hadoop, or databases of any kind
- The advantages of using Kafka Connectors with data storage:
    - Data Centric Pipeline
        - To pull or push data to Kafka, Connect leverages relevant data abstractions
    - Flexibility and Scalability
        - Connect can be used with streaming and batch-oriented systems on a single node (standalone) or as a scaled-out service for an entire business (distributed)
    - Reusability and Extensibility
        - Connect adapts current connectors or extends them to meet your specific requirements, reducing time to market
- The cluster can be managed via Kafka Connect's REST API
- This offers APIs for viewing connection settings and task status as well as modifying their present behavior (for example, changing configuration and restarting tasks)
- Avro
    - value.converter=io.confluent.connect.avro.AvroConverter
- Protobuf
    - value.converter=io.confluent.connect.protobuf.ProtobufConverter
- String
    - value.converter=org.apache.kafka.connect.storage.StringConverter
- JSON
    - value.converter=org.apache.kafka.connect.json.JsonConverter
- JSON Schema

    - value.converter=io.confluent.connect.json.JsonSchemaConverter
- ByteArray
    - value.converter=org.apache.kafka.connect.converters.ByteArrayConverter
- A key-value database is a non-relational database that stores data using a simple key-value mechanism
- Data is stored in a key-value database as a collection of key-value pairs with a key serving as a unique identifier
- Both keys and values can be any type of object from simple to sophisticated compound objects
- Advantages
    - Scalability
        - Key-value stores are well-suited to large data sets and can support a steady stream of read/write operations
        - It is very scalable as a result of this property
        - Partitions, replication, and auto-recovery are used to scale out key-value stores
    - Speed
        - Simple operational commands like get, put, and delete are used in key-value stores making them extremely adept at processing continuous streams of read/write operations
        - In comparison to other database models, the path requests they employ are shorter and more direct, meaning that more operations may be executed in a given amount of time
    - Flexibility
        - Because the value can be anything from a number to a text or even JSON, key-value storage are highly flexible
    - Improved User Experience
        - In key-value stores, storing data for client personalization is fairly common
        - Once the customer's data, as well as their behaviors and preferences, have been gathered, the user experience will be customized
- Disadvantages
    - Key-value stores aren't recommended for applications that need to be updated often or for sophisticated queries involving specific data values or many unique keys with relationships between them
    - There is no query language. Queries from one database may not be transferable to another key-value database without the usage of a common query language

- MongoDB, Amazon DynamoDB, Aerospike, Redis, and Couchbase are some of the popular KV databases