Kafka Topics Partition Count Recommender Application

The Kafka Cluster Topics Partition Count Recommender Application delivers data-driven precision to Kafka topic sizing. By analyzing historical consumption patterns, bytes and records per topic at specific points in time, it calculates daily averages of bytes-per-record and record counts, combining them to measure consumer throughput. Over a rolling 7–14 day window, the application pinpoints peak throughput, scales it by a factor X to project future demand, and then translates that requirement into an optimal partition count. The result: an intelligent, automated recommendation engine that ensures each Kafka topic has the correct number of partitions to sustain workload throughput and future growth reliably.

Table of Contents

1.0 Let's Run It!

1. Get your Confluent Cloud API key pair by executing the following Confluent CLI command to generate the Cloud API Key (click here to learn why you need it):

   ```
   confluent api-key create --resource "cloud"
   ```

   The API Key pair allows Terraform to provision, manage, and update Confluent Cloud resources as defined in your infrastructure code, maintaining a secure, automated deployment pipeline.

2. Clone the repo:

   ```
   git clone https://github.com/j3-signalroom/kafka-cluster-topics-partition_count_recommender-app.git
   ```

3. Create the `.env` file and add the following environment variables, filling them with your Confluent Cloud credentials and other required values:

```
BOOTSTRAP_SERVER_URI=<YOUR_BOOTSTRAP_SERVER_URI>
CONFLUENT_CLOUD_API_KEY=<YOUR_CONFLUENT_CLOUD_API_KEY>
```

```
CONFLUENT_CLOUD_API_SECRET=<YOUR_CONFLUENT_CLOUD_API_SECRET>
INCLUDE_INTERNAL_TOPICS=False
KAFKA_API_KEY=<YOUR_KAFKA_API_KEY>
KAFKA_API_SECRET=<YOUR_KAFKA_API_SECRET>
KAFKA_CLUSTER_ID=<YOUR_KAFKA_CLUSTER_ID>
SAMPLE_RECORDS=True
SAMPLE_SIZE=1000
TOPIC_FILTER=
```

4. Here you go, run the application:

```
uv run python src/app.py
```

1.1 Did you notice we prepended `uv run` to `python src/app.py`?

You maybe asking yourself why. Well, `uv` is an incredibly fast Python package installer and dependency resolver, written in **Rust**, and designed to seamlessly replace `pip`, `pipx`, `poetry`, `pyenv`, `twine`, `virtualenv`, and more in your workflows. By prefixing `uv run` to a command, you're ensuring that the command runs in an optimal Python environment.

Now, let's go a little deeper into the magic behind `uv run`:

- When you use it with a file ending in `.py` or an HTTP(S) URL, `uv` treats it as a script and runs it with a Python interpreter. In other words, `uv run file.py` is equivalent to `uv run python file.py`. If you're working with a URL, `uv` even downloads it temporarily to execute it. Any inline dependency metadata is installed into an isolated, temporary environment—meaning zero leftover mess! When used with `-`, the input will be read from `stdin`, and treated as a Python script.
- If used in a project directory, `uv` will automatically create or update the project environment before running the command.
- Outside of a project, if there's a virtual environment present in your current directory (or any parent directory), `uv` runs the command in that environment. If no environment is found, it uses the interpreter's environment.

So what does this mean when we put `uv run` before `python src/app.py`? It means `uv` takes care of all the setup—fast and seamless—right in your local environment. If you think AI/ML is magic, the work the folks at Astral have done with `uv` is pure wizardry!

Curious to learn more about Astral's `uv`? Check these out:

- Documentation: Learn about `uv`.
- Video: `uv` IS THE FUTURE OF PYTHON PACKING!.

If you have connectivity issues, you can verify connectivity using the following command:

1.2 Troubleshoot Connectivity Issues (if any)

To verify connectivity to your Kafka cluster, you can use the `kafka-topics.sh` command-line tool. First, create a `client.properties` file with your Kafka credentials:

```
# For SASL_SSL (most common for cloud services)
security.protocol=SASL_SSL
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule
required \
  username="<<YOUR_KAFKA_API_KEY>>" \
  password="<YOUR_KAFKA_API_SECRET>";

# Additional SSL settings if needed
ssl.endpoint.identification.algorithm=https
```

```
./kafka-topics.sh --list --bootstrap-server <YOUR_BOOTSTRAP_SERVER_URI> --
command-config ./client.properties
```

2.0 Manually determining the number of partitions needed for a Kafka Consumer

For example, you have a consumer that consumes at **25MB/s**, but the the consumer requirement is a throughput of **1GB/s**. How many partitions should you have?

To determine the number of partitions needed to support a throughput of **1GB/s** for a Kafka consumer that can only consume at **25MB/s**, you can calculate it as follows:

1. Convert the target throughput to the same units:
   - **1GB/s = 1024MB/s**
2. Divide the target throughput by the consumer's capacity:

$$\text{Number of partitions} = \frac{\text{Required throughput}}{\text{Consumer throughput}} = \frac{1024MB/s}{25MB/s} = 40.96$$

3. Since you can only have a whole number of partitions, you should round up to the nearest whole number:

$$\text{Number of partitions} = 41$$

The **41 partitions** ensure that the consumer can achieve the required throughput of **1GB/s** while consuming at a rate of **25MB/s** per partition. This will allow the workload to be distributed across partitions so that multiple consumers can work in parallel to meet the throughput requirement.

3.0 What is meant by the Kafka Consumer throughput?

The throughput of a **Kafka consumer** refers to the rate at which it can read data from Kafka topics, typically measured in terms of **megabytes per second (MB/s)** or **messages per second**. Consumer throughput depends on several factors, including the configuration of Kafka, the consumer application, and the underlying infrastructure.

3.1 Key Factors Affecting Kafka Consumer Throughput:

1. **Partitions**
   - Throughput scales with the number of partitions assigned to the consumer. A consumer can read from multiple partitions concurrently, but the total throughput is bounded by the number of partitions and their data production rates.
   - Increasing the number of partitions can improve parallelism and consumer throughput.
2. **Consumer Parallelism**
   - A single consumer instance reads from one or more partitions, but it can be overwhelmed if the data rate exceeds its capacity.
   - Adding more consumers in a consumer group increases parallelism, as Kafka reassigns partitions to balance the load.
3. **Fetch Configuration**
   - **`fetch.min.bytes`**: Minimum amount of data (in bytes) the broker returns for a fetch request. Larger values reduce fetch requests but may introduce latency.
   - **`fetch.max.bytes`**: Maximum amount of data returned in a single fetch response. A higher value allows fetching larger batches of messages, improving throughput.
   - **`fetch.max.wait.ms`**: Maximum time the broker waits before responding to a fetch request. A higher value can increase batch sizes and throughput but may increase latency.
4. **Batch Size**
   - Consumers process messages in batches for better efficiency. Larger batches reduce processing overhead but require sufficient memory.
   - Configuration: **`max.poll.records`** controls the number of records fetched in a single poll.
5. **Message Size**
   - Larger messages can reduce throughput if the network or storage systems are bottlenecks. Use compression (e.g., `gzip`, `snappy`) to optimize data transfer.
6. **Network Bandwidth**
   - Network speed between Kafka brokers and consumers is critical. A consumer running on a limited-bandwidth network will see reduced throughput.
7. **Deserialization Overhead**
   - The time required to deserialize records impacts throughput. Efficient deserialization methods (e.g., Avro, Protobuf with optimized schemas) can help.
8. **Broker Load**

- Broker performance and replication overhead impact the throughput seen by consumers. If brokers are under heavy load, consumer throughput may decrease.

9. **Consumer Poll Frequency**

   - Consumers must frequently call `poll()` to fetch messages. If the consumer spends too much time processing messages between polls, throughput can drop.

10. **System Resources**

- CPU, memory, and disk I/O on the consumer's machine affect how fast it can process data.

3.2 Typical Kafka Consumer Throughput:

- **Single Partition Throughput**: A single consumer reading from a single partition can typically achieve **10-50 MB/s** or higher, depending on message size, compression, and hardware.
- **Multi-Partition Throughput**: For a consumer group reading from multiple partitions, throughput can scale linearly with the number of partitions (subject to other system limits).

3.3 Strategies to Improve Consumer Throughput:

1. **Increase Partitions**: Scale partitions to allow more parallelism.
2. **Add Consumers**: Add more consumers in the consumer group to distribute the load.
3. **Optimize Fetch Settings**: Tune `fetch.min.bytes`, `fetch.max.bytes`, and `fetch.max.wait.ms`.
4. **Batch Processing**: Use `max.poll.records` to fetch and process larger batches.
5. **Compression**: Enable compression to reduce the amount of data transferred.
6. **Efficient Serialization**: Use optimized serializers and deserializers.
7. **Allocate Resources**: Ensure consumers run on high-performance hardware with sufficient network bandwidth.

By optimizing these factors, Kafka consumers can achieve higher throughput tailored to the specific use case and infrastructure.

4.0 Resources

- Confluent Kafka Python Client Documentation
- Optimize Confluent Cloud Clients for Throughput
- Choose and Change the Partition Count in Kafka