Every function and operator in Flink can be **stateful** .Stateful functions store data across the processing of individual elements/events, making state a critical building block for any type of more elaborate operation.

In order to make state fault tolerant, Flink needs to **checkpoint** the state. Checkpoints allow Flink to recover state and positions in the streams to give the application the same semantics as a failure-free execution.

### Related Config Options

Some more parameters and/or defaults may be set via conf/flink-conf.yaml (see [configuration](https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/stream/config.html) for a full guide):

* state.backend: The backend that will be used to store operator state checkpoints if checkpointing is enabled. Supported backends:
  + jobmanager: In-memory state, backup to JobManager’s/ZooKeeper’s memory. Should be used only for minimal state (Kafka offsets) or testing and local debugging.
  + filesystem: State is in-memory on the TaskManagers, and state snapshots are stored in a file system. Supported are all filesystems supported by Flink, for example HDFS, S3, …
* state.backend.fs.checkpointdir: Directory for storing checkpoints in a Flink supported filesystem. Note: State backend must be accessible from the JobManager, use file:// only for local setups.
* state.backend.rocksdb.checkpointdir: The local directory for storing RocksDB files, or a list of directories separated by the systems directory delimiter (for example ‘:’ (colon) on Linux/Unix). (DEFAULT value is taskmanager.tmp.dirs)
* state.checkpoints.dir: The target directory for meta data of [externalized checkpoints](https://ci.apache.org/projects/flink/flink-docs-release-1.3/setup/checkpoints.html#externalized-checkpoints).
* state.checkpoints.num-retained: The number of completed checkpoint instances to retain. Having more than one allows recovery fallback to an earlier checkpoints if the latest checkpoint is corrupt. (Default: 1)

By default, checkpointing is disabled. To enable checkpointing, call enableCheckpointing(n) on the StreamExecutionEnvironment, where n is the checkpoint interval in milliseconds.

Other parameters for checkpointing include:

* exactly-once vs. at-least-once: You can optionally pass a mode to the enableCheckpointing(n) method to choose between the two guarantee levels. Exactly-once is preferrable for most applications. At-least-once may be relevant for certain super-low-latency (consistently few milliseconds) applications.
* checkpoint timeout: The time after which a checkpoint-in-progress is aborted, if it did not complete by then.
* minimum time between checkpoints: To make sure that the streaming application makes a certain amount of progress between checkpoints, one can define how much time needs to pass between checkpoints. If this value is set for example to 5000, the next checkpoint will be started no sooner than 5 seconds after the previous checkpoint completed, regardless of the checkpoint duration and the checkpoint interval. Note that this implies that the checkpoint interval will never be smaller than this parameter.

It is often easier to configure applications by defining the “time between checkpoints” then the checkpoint interval, because the “time between checkpoints” is not susceptible to the fact that checkpoints may sometimes take longer than on average (for example if the target storage system is temporarily slow).

Note that this value also implies that the number of concurrent checkpoints is one.

* number of concurrent checkpoints: By default, the system will not trigger another checkpoint while one is still in progress. This ensures that the topology does not spend too much time on checkpoints and not make progress with processing the streams. It is possible to allow for multiple overlapping checkpoints, which is interesting for pipelines that have a certain processing delay (for example because the functions call external services that need some time to respond) but that still want to do very frequent checkpoints (100s of milliseconds) to re-process very little upon failures.

This option cannot be used when a minimum time between checkpoints is defined.

* externalized checkpoints: You can configure periodic checkpoints to be persisted externally. Externalized checkpoints write their meta data out to persistent storage and are not automatically cleaned up when the job fails. This way, you will have a checkpoint around to resume from if your job fails. There are more details in the [deployment notes on externalized checkpoints](https://ci.apache.org/projects/flink/flink-docs-release-1.3/setup/checkpoints.html#externalized-checkpoints).

1. StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
3. // start a checkpoint every 1000 ms
4. env.enableCheckpointing(1000);
6. // advanced options:
8. // set mode to exactly-once (this is the default)
9. env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY\_ONCE);
11. // checkpoints have to complete within one minute, or are discarded
12. env.getCheckpointConfig().setCheckpointTimeout(60000);
14. // make sure 500 ms of progress happen between checkpoints
15. env.getCheckpointConfig().setMinPauseBetweenCheckpoints(500);
17. // allow only one checkpoint to be in progress at the same time
18. env.getCheckpointConfig().setMaxConcurrentCheckpoints(1);
20. // enable externalized checkpoints which are retained after job cancellation
21. env.getCheckpointConfig().enableExternalizedCheckpoints(ExternalizedCheckpointCleanup.RETAIN\_ON\_CANCELLATION);
23. // This determines if a task will be failed if an error occurs in the execution of the task’s checkpoint procedure.
24. env.getCheckpointConfig().setFailOnCheckpointingErrors(true);

* Use the StreamExecutionEnvironment.enableCheckpointing method to set the checkpoint to be enabled. You can use enableCheckpointing(long interval) or enableCheckpointing(long interval, CheckpointingMode mode);interval to specify the trigger interval of the checkpoint.Unit milliseconds), while CheckpointingMode defaults to CheckpointingMode.EXACTLY\_ONCE, or can be specified as CheckpointingMode.AT\_LEAST\_ONCE
* CheckpointingMode can also be set via StreamExecutionEnvironment.getCheckpointConfig().setCheckpointingMode, generally for ultra-low latency applications (About a few milliseconds) You can use CheckpointingMode.AT\_LEAST\_ONCE, most other applications use CheckpointingMode.EXACTLY\_ONCE
* checkpointTimeout is used to specify the timeout for checkpoint execution (Unit milliseconds), if the timeout is not completed, it will be aborted
* minPauseBetweenCheckpoints is used to specify the checkpoint coordinator after a checkpoint is completed, how long it takes to start another checkpoint. When this parameter is specified, the value of maxConcurrentCheckpoints is 1.
* maxConcurrentCheckpoints is used to specify the maximum number of checkpoints in the run. It does not take too much time to checkpoints for packaging topology. If minPauseBetweenCheckpoints is set, the maxConcurrentCheckpoints parameter will not work.Values ​​greater than 1 do not work)
* enableExternalizedCheckpoints is used to enable external persistence of checkpoints, but it does not automatically clean up when the job fails. You need to manually clean the state. ExternalizedCheckpointCleanup is used to specify how to clean up the externalized checkpoint when the job is canceled. DELETE\_ON\_CANCELLATION, when the job is canceled The externalized state is automatically deleted, but if it is the state of FAILED, it will be retained; RETAIN\_ON\_CANCELLATION will retain the externalized checkpoint state when the job is canceled.
* failOnCheckpointingErrors is used to specify whether the task should be failed when the checkpoint is abnormal. The default is true. If set to false, the task will reject the checkpoint and continue to run.

**Flink-conf.yaml related configuration**

1. *#==============================================================================*
2. *# Fault tolerance and checkpointing*
3. *#==============================================================================*
5. *# The backend that will be used to store operator state checkpoints if*
6. *# checkpointing is enabled.*
7. *#*
8. *# Supported backends are 'jobmanager', 'filesystem', 'rocksdb', or the*
9. *# <class-name-of-factory>.*
10. *#*
11. *# state.backend: filesystem*
13. *# Directory for checkpoints filesystem, when using any of the default bundled*
14. *# state backends.*
15. *#*
16. *# state.checkpoints.dir: hdfs://namenode-host:port/flink-checkpoints*
18. *# Default target directory for savepoints, optional.*
19. *#*
20. *# state.savepoints.dir: hdfs://namenode-host:port/flink-checkpoints*
22. *# Flag to enable/disable incremental checkpoints for backends that*
23. *# support incremental checkpoints (like the RocksDB state backend).*
24. *#*
25. *# state.backend.incremental: false*
26. Copy code

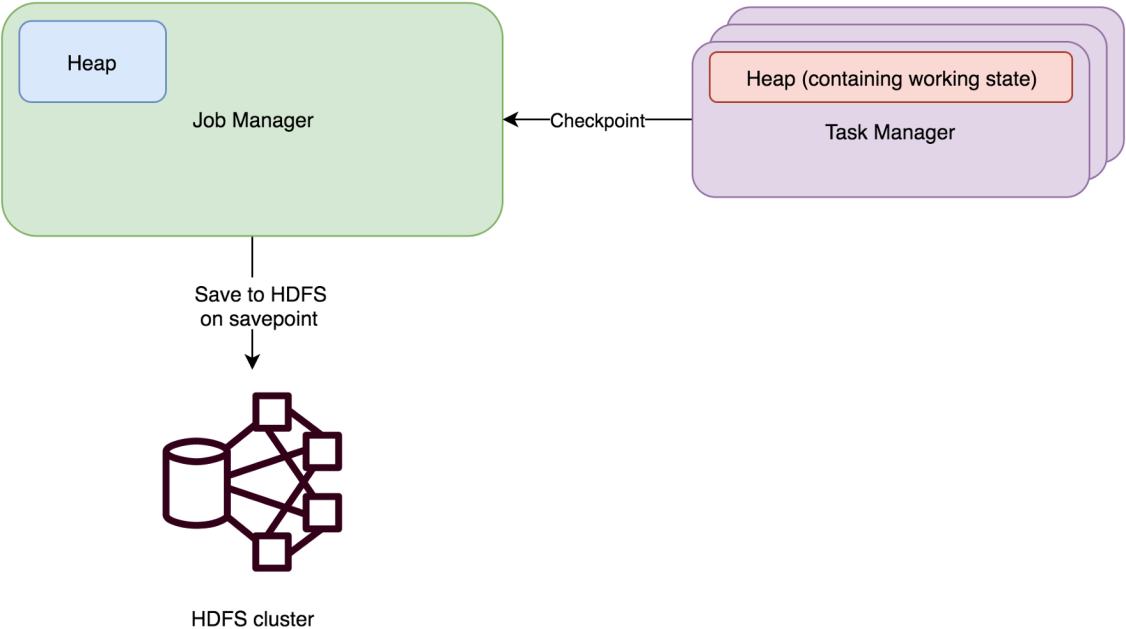
* State.backend is used to specify the backend stored in the checkpoint state. The default is none.
* State.backend.async is used to specify whether backend uses asynchronous snapshots (Default is true), some state backend that does not support async or only supports async may ignore this parameter
* State.backend.fs.memory-threshold, defaults to 1024, which is used to specify the state size threshold stored in files. If it is less than this value, it will be stored in root checkpoint metadata file.
* State.backend.incremental, the default is false, used to specify whether to use incremental checkpoint, some backend that does not support incremental checkpoint will ignore the configuration
* State.backend.local-recovery, defaults to false
* State.checkpoints.dir, defaults to none, specifies the directory of the checkpoint's data files and meta data, which must be visible to all participating TaskManagers and JobManagers.
* State.checkpoints.num-retained, defaults to 1 and specifies the number of completed checkpoints that are reserved.
* State.savepoints.dir, defaults to none, specifies the default directory for savepoints
* Taskmanager.state.local.root-dirs, defaults to none

## Memory State Backend

This storage persists the data in the memory of each task manager’s Heap. Hence, this makes it extremely fast in access. In spite of this performance, this state should never be used in production jobs. That’s because the state creates a backup of the data (also known as checkpointing) in the job manager memory which puts unnecessary pressure on the job manager's operational stability.

Another limitation of this backend that the total state size of a task can’t exceed 10MB. It can be configured to a higher limit but is not advised by the authors due to performance consideration.

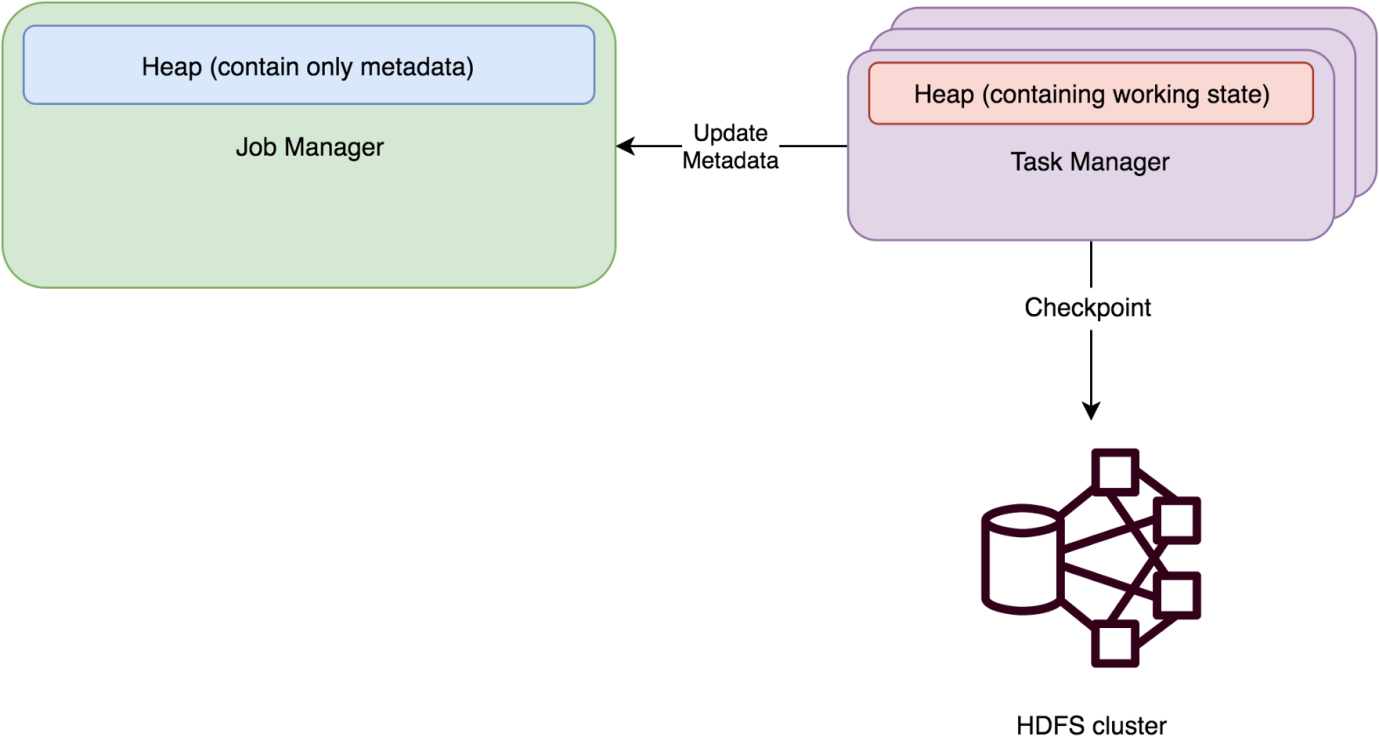
This is the default backend used by Flink in case nothing is configured.



## File System Backend

This backend is similar to Memory state backend except for the fact that it stores the backup on the filesystem rather than job manager memory. The filesystem can be task manager's local filesystem or a durable store such as HDFS/S3.

This state is also limited by the heap memory and hence should be used for cases when you have fewer data and require high performance.

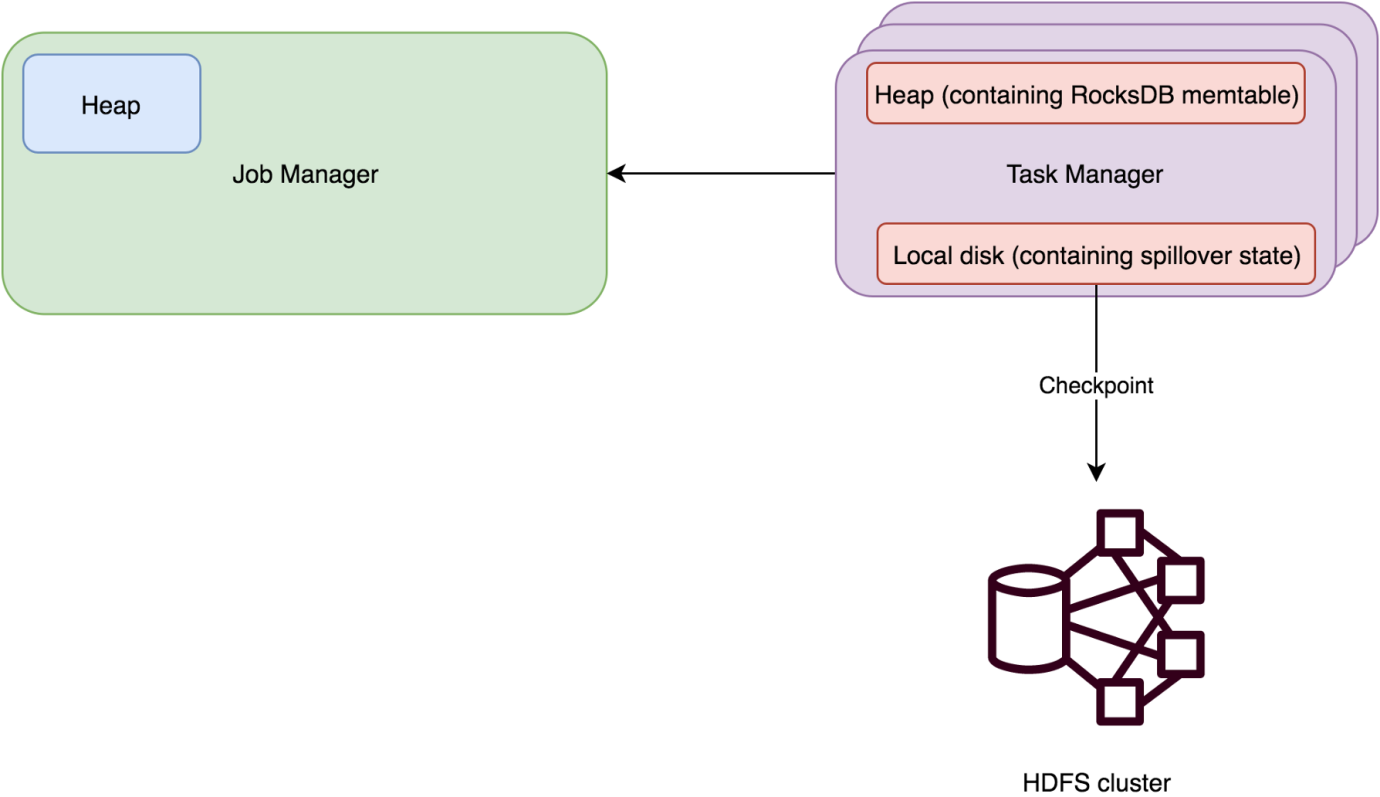


## RocksDB backend

This backend uses [RocksDB by Facebook](https://rocksdb.org/) to store the data. If you are not aware of RocksDB, it’s an embeddable key-value store which offers ACID guarantees. It is based on [LevelDB by Google](https://github.com/google/leveldb) but offers much better write performance.

Flink chose to use RocksDB instead of some of the most popular embeddable storage such as SQLlite because of its high write performance which comes from the LSM architecture based design.  
Since RocksDB also maintains an in-memory table (also known as mem-table) along with bloom filters, reading recent data also is extremely fast.

Each task manager maintains its own Rocks DB file and the backup of this state is checkpointed to a durable store such as HDFS.



### Kafka Consumers Offset Committing Behaviour Configuration

The Flink Kafka Consumer allows configuring the behaviour of how offsets are committed back to Kafka brokers. Note that the Flink Kafka Consumer does not rely on the committed offsets for fault tolerance guarantees. The committed offsets are only a means to expose the consumer’s progress for monitoring purposes.

The way to configure offset commit behaviour is different, depending on whether checkpointing is enabled for the job.

* Checkpointing disabled: if checkpointing is disabled, the Flink Kafka Consumer relies on the automatic periodic offset committing capability of the internally used Kafka clients. Therefore, to disable or enable offset committing, simply set the enable.auto.commit / auto.commit.interval.ms keys to appropriate values in the provided Properties configuration.
* Checkpointing enabled: if checkpointing is enabled, the Flink Kafka Consumer will commit the offsets stored in the checkpointed states when the checkpoints are completed. This ensures that the committed offsets in Kafka brokers is consistent with the offsets in the checkpointed states. Users can choose to disable or enable offset committing by calling the setCommitOffsetsOnCheckpoints(boolean) method on the consumer (by default, the behaviour is true). Note that in this scenario, the automatic periodic offset committing settings in Properties is completely ignored.

### Kafka Consumers and Timestamp Extraction/Watermark Emission

In many scenarios, the timestamp of a record is embedded in the record itself, or the metadata of the ConsumerRecord. In addition, users may want to emit watermarks either periodically, or irregularly, e.g. based on special records in the Kafka stream that contain the current event-time watermark. For these cases, the Flink Kafka Consumer allows the specification of a [watermark strategy](https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/event_time.html).

You can specify your custom strategy as described [here](https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/event_timestamps_watermarks.html), or use one from the [predefined ones](https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/event_timestamp_extractors.html).

Properties properties = new Properties();

properties.setProperty("bootstrap.servers", "localhost:9092");

properties.setProperty("group.id", "test");

FlinkKafkaConsumer<String> myConsumer =

new FlinkKafkaConsumer<>("topic", new SimpleStringSchema(), properties);

myConsumer.assignTimestampsAndWatermarks(

WatermarkStrategy.

.forBoundedOutOfOrderness(Duration.ofSeconds(20)));

DataStream<String> stream = env.addSource(myConsumer);

**Note**: If a watermark assigner depends on records read from Kafka to advance its watermarks (which is commonly the case), all topics and partitions need to have a continuous stream of records. Otherwise, the watermarks of the whole application cannot advance and all time-based operations, such as time windows or functions with timers, cannot make progress. A single idle Kafka partition causes this behavior. Consider setting appropriate [idelness timeouts](https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/event_timestamps_watermarks.html" \l "dealing-with-idle-sources) to mitigate this issue.

## Kafka Connector Metrics

Flink’s Kafka connectors provide some metrics through Flink’s [metrics system](https://ci.apache.org/projects/flink/flink-docs-release-1.12/ops/metrics.html) to analyze the behavior of the connector. The producers export Kafka’s internal metrics through Flink’s metric system for all supported versions. The Kafka documentation lists all exported metrics in its [documentation](http://kafka.apache.org/documentation/#selector_monitoring).

In addition to these metrics, all consumers expose the current-offsets and committed-offsets for each topic partition. The current-offsets refers to the current offset in the partition. This refers to the offset of the last element that we retrieved and emitted successfully. The committed-offsets is the last committed offset.

The Kafka Consumers in Flink commit the offsets back to the Kafka brokers. If checkpointing is disabled, offsets are committed periodically. With checkpointing, the commit happens once all operators in the streaming topology have confirmed that they’ve created a checkpoint of their state. This provides users with at-least-once semantics for the offsets committed to Zookeeper or the broker. For offsets checkpointed to Flink, the system provides exactly once guarantees.

The offsets committed to ZK or the broker can also be used to track the read progress of the Kafka consumer. The difference between the committed offset and the most recent offset in each partition is called the consumer lag. If the Flink topology is consuming the data slower from the topic than new data is added, the lag will increase and the consumer will fall behind. For large production deployments we recommend monitoring that metric to avoid increasing latency.

* Do not upgrade Flink and the Kafka Connector version at the same time.
* Make sure you have a group.id configured for your Consumer.
* Set setCommitOffsetsOnCheckpoints(true) on the consumer so that read offsets are committed to Kafka. It’s important to do this before stopping and taking the savepoint. You might have to do a stop/restart cycle on the old connector version to enable this setting.
* Set setStartFromGroupOffsets(true) on the consumer so that we get read offsets from Kafka. This will only take effect when there is no read offset in Flink state, which is why the next step is very important.
* Change the assigned uid of your source/sink. This makes sure the new source/sink doesn’t read state from the old source/sink operators.
* Start the new job with --allow-non-restored-state because we still have the state of the previous connector version in the savepoint.

### Kafka Producers and Fault Tolerance

With Flink’s checkpointing enabled, the FlinkKafkaProducer can provide exactly-once delivery guarantees.

Besides enabling Flink’s checkpointing, you can also choose three different modes of operating chosen by passing appropriate semantic parameter to the FlinkKafkaProducer:

* Semantic.NONE: Flink will not guarantee anything. Produced records can be lost or they can be duplicated.
* Semantic.AT\_LEAST\_ONCE (default setting): This guarantees that no records will be lost (although they can be duplicated).
* Semantic.EXACTLY\_ONCE: Kafka transactions will be used to provide exactly-once semantic. Whenever you write to Kafka using transactions, do not forget about setting desired isolation.level (read\_committed or read\_uncommitted - the latter one is the default value) for any application consuming records from Kafka.