**Apache Spark** is a unified analytics engine for large-scale data processing.

① The main abstraction Spark provides is a resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel.

a) One important parameter for parallel collections is the number of partitions to cut the dataset into. Spark will run one task for each partition of the cluster.

b) RDDs support two types of operations: transformations, which create a new dataset from an existing one, and actions, which return a value to the driver program after running a computation on the dataset. For example, map is a transformation that passes each dataset element through a function and returns a new RDD representing the results. On the other hand, reduce is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program. All transformations in Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently. On a single machine, this will generate the expected output and print all the RDD’s elements. However, in cluster mode, the output to stdout being called by the executors is now writing to the executor’s stdout instead, not the one on the driver, so stdout on the driver won’t show these! To print all elements on the driver, one can use the collect() method to first bring the RDD to the driver node thus: rdd.collect().foreach(println).

② A second abstraction in Spark is shared variables that can be used in parallel operations. Spark supports two types of shared variables: broadcast variables, which can be used to cache a value in memory on all nodes, and accumulators, which are variables that are only “added” to, such as counters and sums.

——————————————————————————

counter = 0

rdd = sc.parallelize(data)

# Wrong: Don't do this!!

**def** increment\_counter(x):

**global** counter

counter += x

rdd.foreach(increment\_counter)

**print**("Counter value: ", counter)

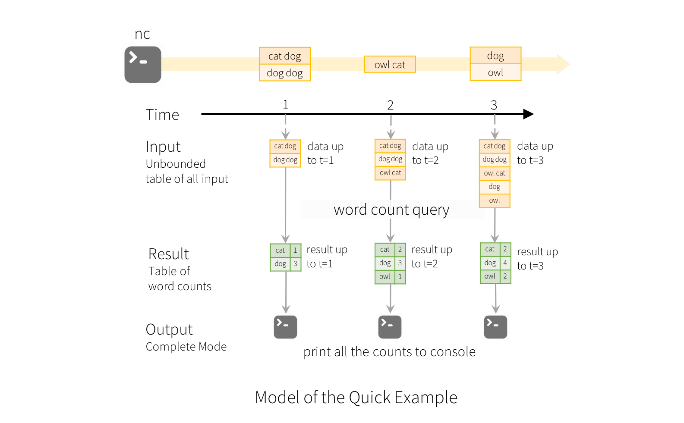
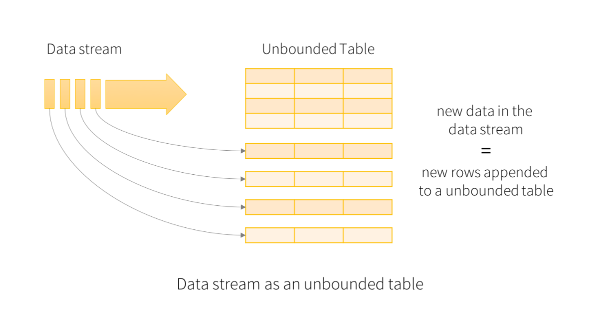
——————————————————————————

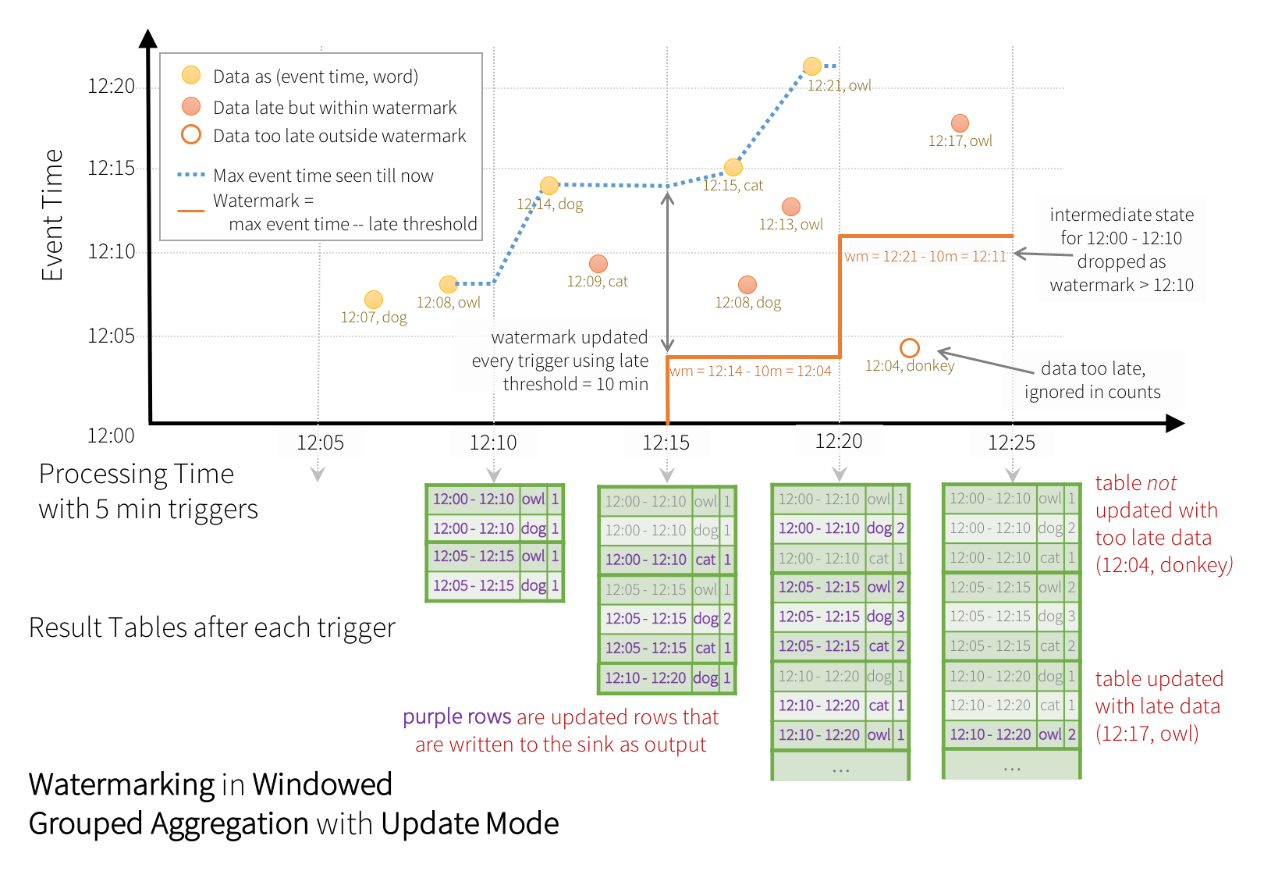
The variables within the closure sent to each executor are now copies and thus, when **counter** is referenced within the foreach function, it’s no longer the **counter** on the driver node. There is still a **counter** in the memory of the driver node but this is no longer visible to the executors!

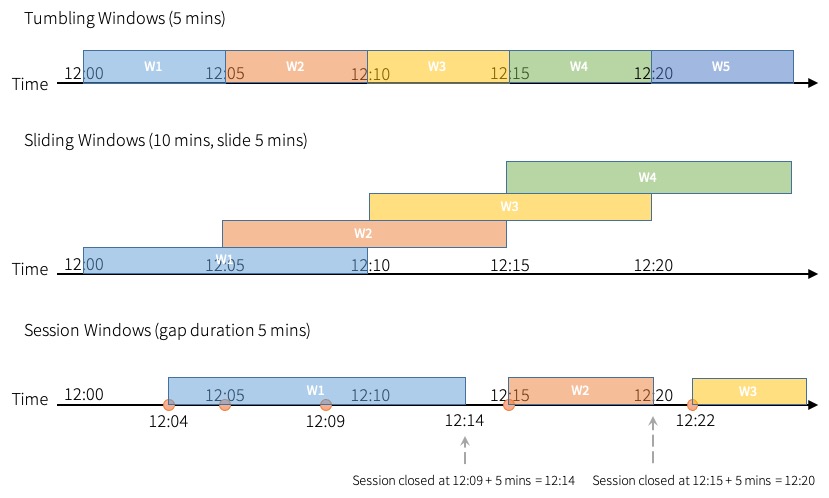
③ The shuffle is Spark’s mechanism for re-distributing data so that it’s grouped differently across partitions. This typically involves copying data across executors and machines, making the shuffle a complex and costly operation.

④ When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it). Spark’s cache is fault-tolerant, if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it. Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (LRU) fashion. If you would like to manually remove an RDD instead of waiting for it to fall out of the cache, use the RDD.unpersist() method.

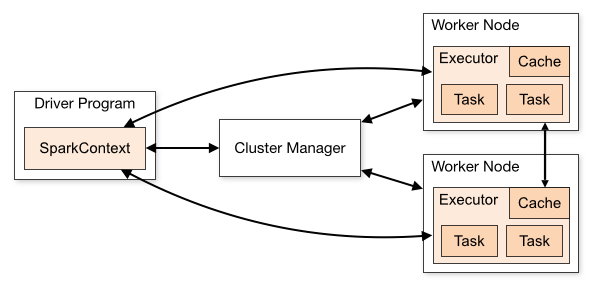
⑤ The key idea in Structured Streaming is to treat a live data stream as a unbounded table that is being continuously appended. This leads to a new stream processing model that is very similar to a batch processing model. A query on the input will generate the “Result Table”. It reads the latest available data from the streaming data source, processes it incrementally to update the result, and then discards the source data. It only keeps around the minimal intermediate state data as required to update the result.







**Cluster Mode：**Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in your main program (called the driver program).



There are several useful things to note about this architecture:

1. Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads. This has the benefit of isolating applications from each other, on both the scheduling side (each driver schedules its own tasks) and executor side (tasks from different applications run in different JVMs). However, it also means that data cannot be shared across different Spark applications (instances of SparkContext) without writing it to an external storage system.
2. Spark is agnostic to the underlying cluster manager. As long as it can acquire executor processes, and these communicate with each other, it is relatively easy to run it even on a cluster manager that also supports other applications (e.g. Mesos/YARN/Kubernetes).
3. The driver program must listen for and accept incoming connections from its executors throughout its lifetime (e.g. see spark.driver.port in the network config section). As such, the driver program must be network addressable from the worker nodes.
4. Because the driver schedules tasks on the cluster, it should be run close to the worker nodes, preferably on the same local area network. If you’d like to send requests to the cluster remotely, it’s better to open an RPC to the driver and have it submit operations from nearby than to run a driver far away from the worker nodes.

**Performance Tuning：**

1. data serialization
2. memory tuning