**Configuration modifications to be made:**

**Zookeeper:**

1. Go to zookeeper config directory. For me its C:\zookeeper-3.6.6\conf
2. Rename file “zoo\_sample.cfg” to “zoo.cfg”
3. Change dataDir=/tmp/zookeeper to dataDir=C:\zookeeper-3.6.6\data
4. Add entry in System Environment Variables for Zookeeper
5. Edit System Variable named “Path” add ;%ZOOKEEPER\_HOME%\bin

**Kafka:**

1. Go to Kafka config directory: C:\kafka\_2.11-0.11.0.1\config
2. Edit file “server.properties”
3. Change line “log.dirs=/tmp/kafka-logs” to “log.dir= C:\kafka\_2.11-0.11.0.1\kafka-logs”.
4. To change zookeeper IP address and port “zookeeper.connect=localhost:2181” (DEFAULT)
5. Kafka will run on default port 9092 & connect to zookeeper’s default port which is 2181.

**To start zookeeper server:**

*.\bin\zkserver*

**To start Kafka server:**

*.\bin\windows\kafka-server-start.bat .\config\server.properties*

**To create Kafka topic:**

*kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test*

**To create Kafka producer:**

*kafka-console-producer.bat --broker-list localhost:9092 --topic test*

**To create Kafka consumer:**

*kafka-console-consumer.bat --zookeeper localhost:2181 --topic test*

CRON JOB:

WINDOWS SCHEDULER:

|  |  |
| --- | --- |
|  | You can do it in the command line as follows:  schtasks /Create /SC HOURLY /TN PythonTask /TR "PATH\_TO\_PYTHON\_EXE PATH\_TO\_PYTHON\_SCRIPT"  That will create an hourly task called 'PythonTask'. You can replace HOURLY with DAILY, WEEKLY etc. PATH\_TO\_PYTHON\_EXE will be something like: C:\python27\python.exe. Check out more examples by writing this in the command line:  schtasks /?  **Installation:**  Pyspark setup: (Jupyter notebook)  https://nerdsrule.co/2016/06/15/ipython-notebook-and-spark-setup-for-windows-10/  <https://github.com/steveloughran/winutils/blob/master/hadoop-2.6.0/bin/winutils.exe?raw=true>  http://spark.apache.org/docs/2.0.1/streaming-programming-guide.html#linking  From which, we could extract information regarding kafka-0-8 and version=2.0.1, remember that for later use. Then we navigate to [Spark Streaming Programming Guide - Linking Part](http://spark.apache.org/docs/2.0.1/streaming-programming-guide.html#linking), change the URL(http://spark.apache.org/docs/**2.0.1**/streaming-programming-guide.html#linking) to the corresponding Spark's version and click [Maven Repository](http://search.maven.org/#search%7Cga%7C1%7Cg%3A%22org.apache.spark%22%20AND%20v%3A%222.0.1%22) link in Linking Part so it will automatically search packages corresponding to your Spark's version. After that, finding ArtifactId in format 'spark-streaming-kafka-**0-8**-assembly\_**2.11**' and version column equals to the previous version(=2.0.1), where 0-8 is retrieved from above and 2.11 is your Scala version. Log groupId, artifactId as well as version from this row.  Edit $SPARK\_HOME/conf/spark-defaults.conf (origined from spark-defaults.conf.template file), append 'spark.jars.packages org.apache.spark:spark-streaming-kafka-0-8-assembly\_2.11:2.0.1', which is in format 'spark.jars.packages groupId:artifactId:version'.  import os  os.environ["GOOGLE\_APPLICATION\_CREDENTIALS"] = "path\_to\_your\_.json\_credential\_file"  **Points to remember:**   * Topic partitions in Kafka does not correlate to partitions of RDDs generated in Spark Streaming. So increasing the number of topic-specific partitions in the KafkaUtils.createStream() only increases the number of threads using which topics that are consumed within a single receiver. It does not increase the parallelism of Spark in processing the data. Refer to the main document for more information on that. * Multiple Kafka input DStreams can be created with different groups and topics for parallel receiving of data using multiple receivers. * If you have enabled Write Ahead Logs with a replicated file system like HDFS, the received data is already being replicated in the log. Hence, the storage level in storage level for the input stream to StorageLevel.MEMORY\_AND\_DISK\_SER (that is, use   Why kafka?  Kafka is simply a collection of topics split into one or more partitions.   A Kafka partition is a linearly ordered sequence of messages, where each message is identified by their index (called as offset).  Kafka is designed for distributed high throughput systems. Kafka tends to work very well as a replacement for a more traditional message broker. In comparison to other messaging systems, Kafka has better throughput, built-in partitioning, replication and inherent fault-tolerance, which makes it a good fit for large-scale message processing applications  has the ability to handle a large number of diverse consumers. Kafka is very fast, performs 2 million writes/sec.  supports low latency message delivery and gives guarantee for fault tolerance in the presence of machine failures.  Which database?  In summary, both Amazon DynamoDB and Apache HBase define data models that allow efficient storage of data to optimize query performance. Amazon DynamoDB imposes a restriction on its item size to allow efficient processing and reduce costs.  Apache HBase uses the concept of column families to provide data locality for more efficient read operations.  Amazon DynamoDB supports both scalar and multi-valued sets to accommodate a wide range of unstructured datasets. Similarly, Apache HBase stores its key/value pairs as arbitrary arrays of bytes, giving it the flexibility to store any data type.  Amazon DynamoDB supports built-in secondary indexes and automatically updates and synchronizes all indexes with their parent tables. With Apache HBase, you can implement and manage custom secondary indexes yourself.  From a data model perspective, you can choose Amazon DynamoDB if your item size is relatively small. Although Amazon DynamoDB provides a number of options to overcome row size restrictions, Apache HBase is better equipped to handle large complex payloads with minimal restrictions.  **BigQuery:**   * BigQuery does not guarantee data consistency for external data sources. * Query performance for external data sources may not be as high as querying data in a native BigQuery table since it depends on the external storage type. * You cannot run a BigQuery job that exports data from an external data source. |
|  | **Cloud Bigtable storage model**   * Cloud Bigtable stores data in massively scalable tables, each of which is a sorted key/value map. The table is composed of rows, each of which typically describes a single entity, and columns, which contain individual values for each row. Each row is indexed by a single row key, and columns that are related to one another are typically grouped together into a column family. Each column is identified by a combination of the column family and a column qualifier, which is a unique name within the column family. * Each row/column intersection can contain multiple cells at different timestamps, providing a record of how the stored data has been altered over time. Cloud Bigtable tables are sparse; if a cell does not contain any data, it does not take up any space.   All client requests go through a front-end server before they are sent to a Cloud Bigtable node.  (In the [original Bigtable whitepaper](http://static.googleusercontent.com/media/research.google.com/en/us/archive/bigtable-osdi06.pdf), these nodes are called "tablet servers.")  The nodes are organized into a Cloud Bigtable cluster, which belongs to a Cloud Bigtable instance, a container for the cluster. Each node in the cluster handles a subset of the requests to the cluster. By adding nodes to a cluster, you can increase the number of simultaneous requests that the cluster can handle, as well as the maximum throughput for the entire cluster.  A Cloud Bigtable table is sharded into blocks of contiguous rows, called *tablets*, to help balance the workload of queries.  Keep each partition for each location  Columns timings  Time interval  Column 1 date |