Data Storage

During the previous assignement, our data storage/key-value store was referencing everything in a .txt file.

Each line used to represent a row, where the key and the value are separated by one space" "

The file was stored locally and the "FileHandler" had to lock on an object whenever a read/write operation is about to be done.

The issue being, especially with the way data is separated and where is it accessible is not that scalable. (What if we wanna store items that contain spaces ?)

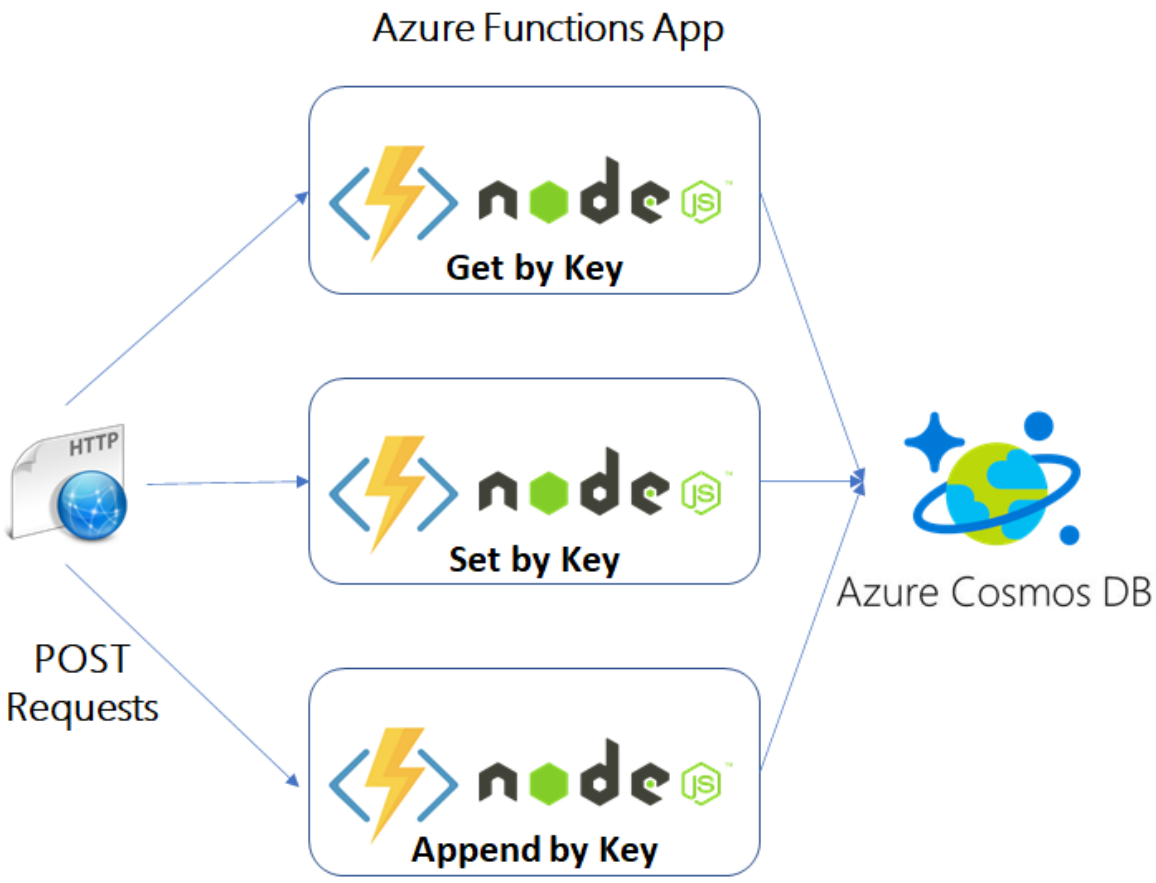
For this assignement, we switched to the usage of Azure CosmosDB on a serverless Tier.

Each item stored had a key, and a value that could be of any time of object.

Image Source : https://miro.medium.com/max/1300/0*XlYpzKXzSV61ltFV.jpg

The operations being executed on CosmosDB where managed through the CosmosDb Node.js SDK.

As for our "FileHandler", it is no longer a class being instatiated in our "Master", but



Our serverless code is based on Node.js version 12. They accept POST requests with a "key" field in the body.

Current Limitation: CosmosDB does not have a lock feature yet. But the Azure Blob Storage service offers that feature. We can use it from our functions, to use it as a lease before attempting any operations on CosmosDb. (This hasn't been implemented yet for this assignement)

Execution Pipeline

At the start of the program, the user is prompted to pass the **configuration file path**.

The configuration file is a .json containing:

- The Reduce Function to be invoked (An enum where 0 means WordCount and 1 for InvertedIndex)
- The endpoint that holds the Serverless functions API. This can be configured to a remote already deployed app on Azure, or something local
- The path of the file used for the WordCount execution
- The array of the files to be used for the InvertedIndex execution

The console app reads that config and executes the proper program with the passed params.

It starts by creating a log file, then deletes the cache/values from the Key-Value store

WordCount Execution Pipeline

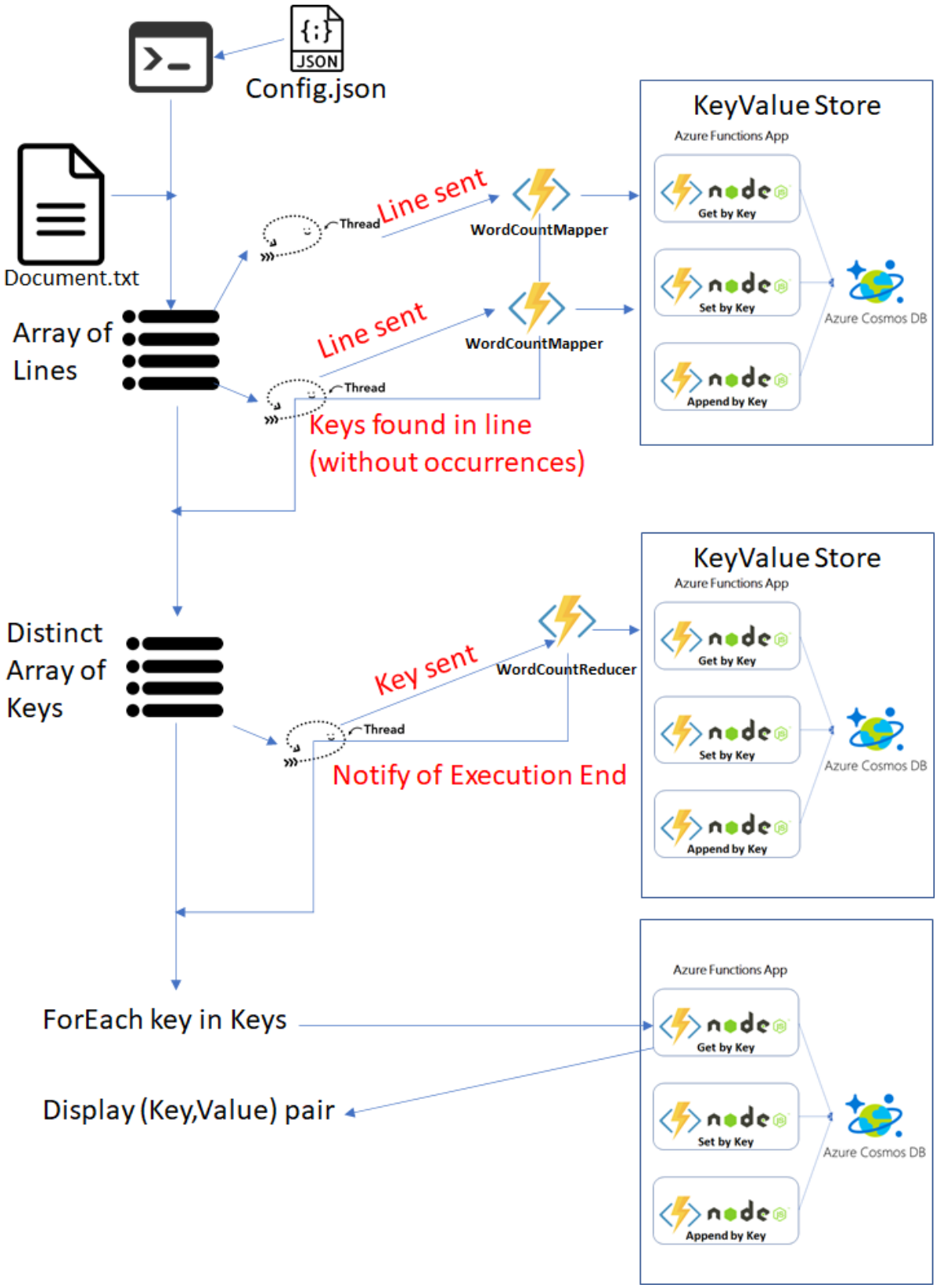
The **master** node splits the file into several lines. Each line is a "key" to the WordCountMapper.

The **master** creates a thread that calls an Azure Function to map that line into several pairs of ['Key', Occurrence]. These pairs are then passed to the Key-value store. The mapper then only returns the key to the master.

The **master** then creates a list of **distinct** keys, but does **NOT** make any aggregation nor merges data.

Then, for each key, the **master** creates a thread that calls an Azure Function to reduce that **key**. **ONLY** the key is sent to the reducer/AzureFunction which in turn, gets the data it needs from the KeyValue Store. Once reduced, it returns the same key to the **master**.

In order to display the results, the **master** calls the KeyValue store in order to receive the values of all the returned keys. (The reducer does **NOT** return the value to the master. It saves it into the store, and it gets retrieved by the master later on)


JSON
Config.json
Document.txt
Array of Lines
Thread
Line sent
WordCountMapper
Line sent
WordCountMapper
Thread
Keys found in line
(without occurrences)
KeyValue Store
Azure Functions App
Get by Key
Set by Key
Append by Key
Azure Cosmos DB
Distinct Array of Keys
Key sent
WordCountReducer
Thread
Notify of Execution End
KeyValue Store
Azure Functions App
Get by Key
Set by Key
Append by Key
Azure Cosmos DB
ForEach key in Keys
Display (Key,Value) pair
Azure Functions App
Get by Key
Set by Key
Append by Key
Azure Cosmos DB

Fault Tolerance

- Splitting a file into several lines is done locally for this operation, so no need to re-execute.
- When the WordCountMapper's done with mapping the values, it saves the data in the key-value-store. Whenever an exception is caught, it keeps iterating until it is saved.
- The Thread that communicates remotely with the WordCountMapper also relaunches the invocation if network communication is lost along the way.
- The same goes for the WordCountReducer function code, and the thread that communicates with it. Whenever an exception is caught on the Azure Function code, it gets relaunched. If the thread that sent the request lost connection, it recalls again the Azure Function until the Master is notified
-

InvertedDocument Execution Pipeline

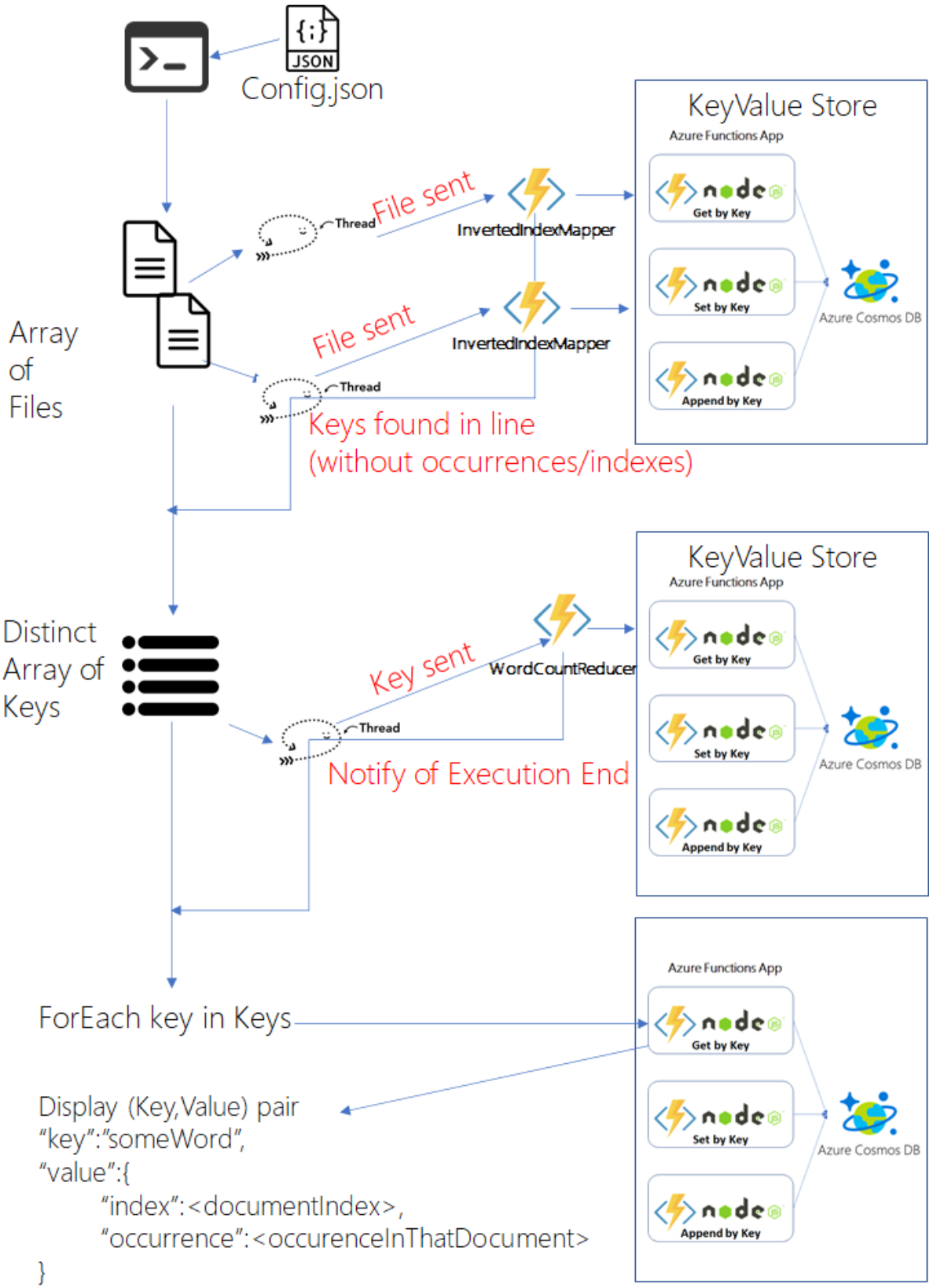
For each document, The **master** creates a thread that calls an Azure Function to map that document into several pairs of ['Key:Word',{documentIndex, occurrence}]. These pairs are then passed to the Key-value store. The mapper then only returns the key to the master.

The **master** then creates a list of **distinct** keys, but does **NOT** make any aggregation nor merges data.

Then, for each key, the **master** creates a thread that calls an Azure Function to reduce that **key**. **ONLY** the key is sent to the reducer/AzureFunction which in turn, gets the data it needs from the KeyValue Store.

The reducer receives the occurrences in each document index for that word and creates the sum. Once reduced, it returns the same key to the **master**.

In order to display the results, the **master** calls the KeyValue store in order to receive the values of all the returned keys. (The reducer does **NOT** return the value to the master. It saves it into the store, and it gets retrieved by the master later on)


JSON
Config.json
KeyValue Store
Azure Functions App
Get by Key
Set by Key
Append by Key
Azure Cosmos DB
Thread
File sent
InvertedIndexMapper
File sent
InvertedIndexMapper
Thread
Array
of
Files
Keys found in line
(without occurrences/indexes)
KeyValue Store
Azure Functions App
Get by Key
Set by Key
Append by Key
Azure Cosmos DB
Distinct
Array of
Keys
Key sent
WordCountReducer
Thread
Notify of Execution End
Azure Functions App
Get by Key
Set by Key
Append by Key
Azure Cosmos DB
ForEach key in Keys
Display (Key,Value) pair
"key":"someWord",
"value":{
"index":<documentIndex>,
"occurrence":<occurenceInThatDocument>
}

Setup

You have to setup the

- Frontend console application
- Backend

Let's start by building the frontend application.

1) For that, go to the ./MapReduce.Client/ folder
2) Execute dotnet build –configuration release
3) Go to /bin/release/netcoreapp3.1
4) Launch ./MapReduce.Client.exe

You will be prompted to input the Configuration file. You need to set:

- `ReduceFunction = 0 (for WordCount) | = 1 (for InvertedIndex)`
- `(For WordCount program) WordCountFilePath pointing to the file that will be parsed. Path should be absolute (or relative to the .exe)`
- `(For InvertedIndex) InvertedIndexFilePaths = [Array of FilePaths to be used as documents]`
- `AzureFunctionsEndpoint (For both programs), is the root url of the api the program will use. If you're going to configure the backend locally, put it to http://localhost:<port>/api (Do not put / after the api). You can also use the already deployed functions app (although it has some limitations). https://engr-func-app.azurewebsites.net/api`

Configuring the backend:

If we're going for the local approach:

- Navigate to MapReducer.Functions
- Run: **yarn install** (or **npm install**) **You need to have node.js v12 installed**
- Install the Azure-functions-core-tools to allow you to run these functions locally, you can do it through: **npm i -g azure-functions-core-tools@2**
- Run the app through: **npm function** or **yarn function**
- Use that locallink in the AzureFunctionsEndpoint field in your config.json

If we're going for the remote approach:

- Set `https://engr-func-app.azurewebsites.net/api` as the AzureFunctionsEndpointField in your config.json
- ???????????
- Profit

Limitations of each Setup

- When executing locally, the Azure-Functions-core-tools doesn't allow to execute several requests simultaneously. Basically, even with the multiple threads on the frontend posting several requests at the same time, they will all be executed sequentially which will slow down.
- When executing remotely on Azure, certain limitations arise especially when deleting the keystore initially and when dealing with large documents. Weirdly enough, the requests remotely are being throtteled or reach timeout. (There's a retry/fault tolerance, but it will have to be executed **several** times.

Comparison with the Research Paper

In the Fault Tolerance part, they are discussing 2x scenarios:

- On worker Failure
    - Detect via periodic heartbeats. In our case, since we're sending HTTP requests to the Azure Functions, as soon as we have a timeout or we receive a 5xx, we know something's wrong.
    - Re-execute completed and in-progress map tasks. (This is done as we wrap the code in try/catches and re-execute, whether it be on the Azure Function end, or the Thread that calls that function
    - Re-execute in progress reduce tasks. (This is done the same way as wrap the code in try/catches and re-execute Serverless-Side or frontend-wise on the Thread. (whether it be communication/network issues or other types of exceptions caught on execution)
- On Master Failure
    - "Could handle, but don't yet (master failure unlikely)". We're somehow assuming everything's fine. FilePaths provided are valid. Their content is actually UTF8- textual data that can be parsed.

We also provide parallelization of the tasks and their distributions across different resources through the network. (Other contains running our serverless code)

We also provide a status/logging feature as we showcase, what endpoint are we invoking, with which key ? What are we attempting to do ? Did an exception get caught ? Are we done with the reducers ? Did we get an exception ? Are we re-attempting ? You may check the log-output on both the console and the generated file

We do not provide monitoring and I/O scheduling yet.

Samples

3 Sample folders have been provided in the /MapReduceSamples folders

Each folder contains some .txt files, along with a Config.json file (you have to edit the Absolute Path. This was done intetionnally to avoid any assumptions about relativeness)

A log output of the execution was also provided with each one.