**MAP REDUCE SCRIPT ASSIGNMNET**

**1. Explain the Stages/Entry Points in Map-Reduce Script and Flow of the Stages.**

There are four entry points in NetSuite. Ie.

1. **getInputData**() - It provides us with the input data that we have to process in this stage. We can get data from CSV format or can create a saved search and load it here. Also, we can create a new record as well. Then we return the processed data to the next stage.
2. **map** () - It processes induvial data passed from getInputData and converts it to JS object to make it more readable. Then it makes {key: value} pair and processes to the next function by .write() API.
3. **shuffle** () - We don't have access to this stage as it's internally handled by the SuiteScript site. But here in this stage, the repeated values of a key are stored in an array.
4. **reduce**() - It processes the grouped data based on the key and performs some operation batch wise parallelly. Here we can log the output got from the reduce stage.
5. **summarize**() - It executes after the all the previous stage are complete. It provides a summary of the script execution. It shows the overall outcome of the entire process.

**2. Write Down any Two Scenario where Map-Reduce script is Used.**

**First Scenario:**

***Use Case:*** Bulk updating of all the items after certain period of time.

***Scenario:*** At the start of every quarter, the business needs to update the pricing for thousands of items in its inventory based on market conditions or inflation rate.

***How it Works:*** A Map/Reduce Script is created to handle this large-scale data processing. The script retrieves all items that need to be updated, breaks them down into smaller batches to do execute parallelly, which helps to operate this situation smoothly with less time.

**Second Scenario:**

***Use Case***: Bulk Email Notification for Pending Invoice Bills

***Scenario:*** At the end of each month, the finance department needs to send reminders to customers about their pending invoice bills. This will notify thousands of customers at once.

***How it Works:*** An automated email system is set up to handle this large-scale communication. The system retrieves all pending invoices, groups them by customer, and sends personalized email reminders in batches and will parallelly send to each customer.

**3. Create a Map-Reduce Script which is used to group the invoice by customer and send the email to the customer.**

Here I have used '**N/search', 'N/record', 'N/email**'.

* I have first created a saved search with our needed matching criteria.
* Then by **.getRange()** API, I filter out some invoices from the saved search and return it to the map function.
* In the map function, I convert the JSON strings to Java Script Object to get the path of needed data correctly.
* Here I got the customer id and the internal id of invoices. I just mapped them by {key: value} pairs and returned it to the next reduce function by **.write()** API.
* In the reduce function I got the customerId and the id of invoice as an array.
* Then I load the customer record of the particular customer that I got in the reduce function and extract their email from the record.
* If the customer has a mail, then I send a mail from employee Kathryn Glass else show an error message in the log.

***Code:***

/\*\*

\* @NApiVersion 2.1

\* @NScriptType MapReduceScript

\*/

define(['N/search', 'N/record', 'N/email'],

/\*\*

\* @param{search} search

\*/

(search, record, email) => {

const getInputData = (inputContext) => {

var invObjSavedSearch = search.load({

id: 'customsearch1770'

})

var resArr = invObjSavedSearch.run().getRange({start: 0, end:10}) // run is used for <= 4000 records, if it exceeds then use runPaged()

//log.debug('Resultant invoice : ', resArr) // it will give me 10 invoice record in the JSON format

return resArr

}

const map = (mapContext) => {

var newValue = JSON.parse(mapContext.value)

//log.debug('JSON parsed values : ', newValue)

var keyCustomerId = newValue.values.entity[0].value

var valueInvoiceId = newValue.id

// log.debug('id of cusstomer', keyCustomerId)

// log.debug('id of invoice', valueInvoiceId

mapContext.write({key: keyCustomerId, value: valueInvoiceId})

}

const reduce = (reduceContext) => {

var customerId = reduceContext.key

var invArr = reduceContext.values

//log.debug('Array of internal id of invoice : ', invArr)

//log.debug('Array of customer id : ', customerId)

//log.debug('customer having id '+ customerId + ', contains invoice : ' + invArr)

invArr.forEach((invId) => {

record.submitFields({

type: record.Type.INVOICE,

id: invId,

values: {

memo: 'Customer ID : ' + customerId

}

})

})

var custRecord = record.load({

type: record.Type.CUSTOMER,

id: customerId

})

var custEmail = custRecord.getValue({

fieldId: 'email'

})

if(!custEmail){

log.debug({

title: 'Error occured',

details: `The customer having id ${customerId} doesn't have a mail`

})

}else{

email.send({

author: '-5',

body: 'This is email body',

subject: 'This is my email subject',

recipients: custEmail

})

log.debug({

title: 'Email Sent',

details: `Email sent to customer ${customerId} for invoices: ${invArr}`

});

}

}

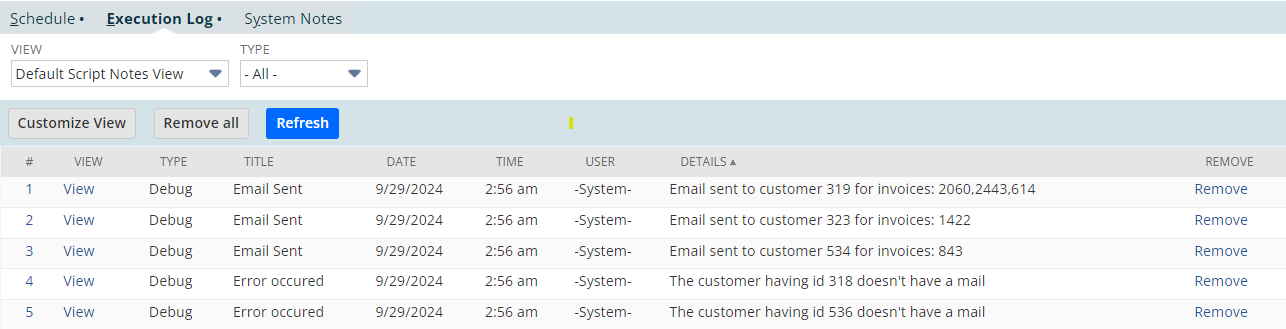
const summarize = (summaryContext) => {

};

return {getInputData, map, reduce, summarize}

});

***Output:***



**5. Create a Map-Reduce script that sends a single email to every customer with at least one past due invoice. The email will contain a list of those invoices with some basic information about them.**

Here I have used **'N/search', 'N/record', 'N/email'**.

* I have first created a saved search with our needed matching criteria.
* Then by **.getRange()** API, I filter out some invoices from the saved search and return it to the map function.
* In the map function, I convert the JSON strings to Java Script Object to get the path of needed data correctly.
* Here I got the customer id and the internal id of invoices. Then I created another object that contains the invoice internal id and bills of the invoice and finally passed it to the reduce function.
* In the reduce function I converted the JSON string to JS object and retrieved the bill amount of the customer.
* Then I send the mail to the customer with their pending bill amount.

***Code:***

/\*\*

\* @NApiVersion 2.1

\* @NScriptType MapReduceScript

\*/

define(['N/log', 'N/record', 'N/search', 'N/email'],

/\*\*

\* @param{log} log

\* @param{record} record

\* @param{search} search

\*/

(log, record, search, email) => {

const getInputData = (inputContext) => {

var savedSearchInvoice = search.load({

id: 'customsearch\_search\_mail\_send\_2'

})

var resArr = savedSearchInvoice.run().getRange(0, 10)

return resArr

}

const map = (mapContext) => {

var newValue = JSON.parse(mapContext.value)

var keyCustId = newValue.values.entity[0].value

var valueInvId = newValue.id

var pendingBill = newValue.values.amount

var valueObject = {

value: valueInvId,

bills: pendingBill

}

mapContext.write({

key: keyCustId,

value: valueObject

})

}

const reduce = (reduceContext) => {

var custId = reduceContext.key;

var invId = reduceContext.values;

var bill;

for (let i = 0; i < invId.length; i++) {

bill = JSON.parse(invId[i]).bills

}

var custRecord = record.load({

type: record.Type.CUSTOMER,

id: custId

});

var custEmail = custRecord.getValue({

fieldId: 'email'

});

if (custEmail) {

email.send({

author: -5,

body: `Hey ${custId}, You have a pending invoice due of ${bill}`,

subject: 'Please pay the bills.',

recipients: custEmail

});

log.debug({

title: 'Email Sent',

details: `Email sent to customer ${custId} for bill: ${bill}`

});

} else {

log.debug({

title: 'No email found!',

details: `Customer having id ${custId} has no email`

});

}

};

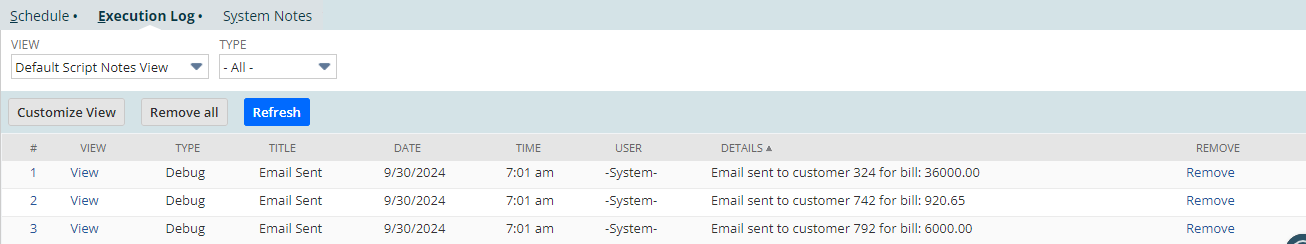
const summarize = (summaryContext) => {

}

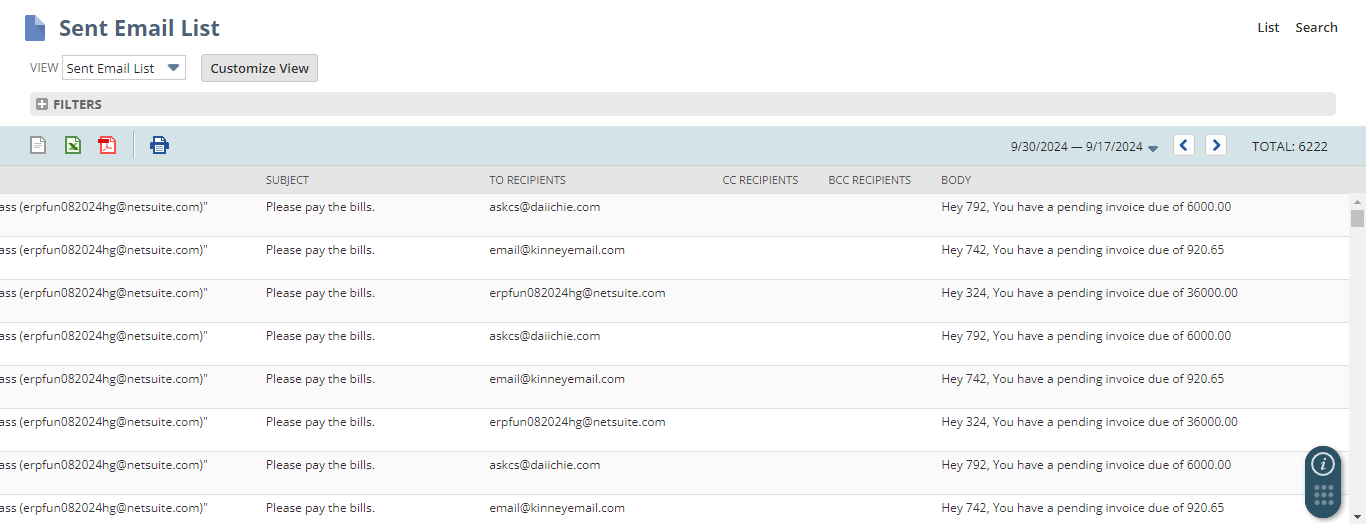
return { getInputData, map, reduce, summarize }

});

***Output:***



***Proof:***



**4. Schedule a Map-Reduce to run once after 1 hour on for each day and attach Screenshot.**

/\*\*

\* @NApiVersion 2.1

\* @NScriptType MapReduceScript

\*/

define(['N/record', 'N/search', 'N/log'],

/\*\*

\* @param{record} record

\* @param{search} search

\*/

(record, search, log) => {

/\*\*

\* Defines the function that is executed at the beginning of the map/reduce process and generates the input data.

\* @param {Object} inputContext

\* @param {boolean} inputContext.isRestarted - Indicates whether the current invocation of this function is the first

\* invocation (if true, the current invocation is not the first invocation and this function has been restarted)

\* @param {Object} inputContext.ObjectRef - Object that references the input data

\* @typedef {Object} ObjectRef

\* @property {string|number} ObjectRef.id - Internal ID of the record instance that contains the input data

\* @property {string} ObjectRef.type - Type of the record instance that contains the input data

\* @returns {Array|Object|Search|ObjectRef|File|Query} The input data to use in the map/reduce process

\* @since 2015.2

\*/

const getInputData = (inputContext) => {

log.debug('My map reduce script is running')

}

/\*\*

\* Defines the function that is executed when the map entry point is triggered. This entry point is triggered automatically

\* when the associated getInputData stage is complete. This function is applied to each key-value pair in the provided

\* context.

\* @param {Object} mapContext - Data collection containing the key-value pairs to process in the map stage. This parameter

\* is provided automatically based on the results of the getInputData stage.

\* @param {Iterator} mapContext.errors - Serialized errors that were thrown during previous attempts to execute the map

\* function on the current key-value pair

\* @param {number} mapContext.executionNo - Number of times the map function has been executed on the current key-value

\* pair

\* @param {boolean} mapContext.isRestarted - Indicates whether the current invocation of this function is the first

\* invocation (if true, the current invocation is not the first invocation and this function has been restarted)

\* @param {string} mapContext.key - Key to be processed during the map stage

\* @param {string} mapContext.value - Value to be processed during the map stage

\* @since 2015.2

\*/

const map = (mapContext) => {

}

/\*\*

\* Defines the function that is executed when the reduce entry point is triggered. This entry point is triggered

\* automatically when the associated map stage is complete. This function is applied to each group in the provided context.

\* @param {Object} reduceContext - Data collection containing the groups to process in the reduce stage. This parameter is

\* provided automatically based on the results of the map stage.

\* @param {Iterator} reduceContext.errors - Serialized errors that were thrown during previous attempts to execute the

\* reduce function on the current group

\* @param {number} reduceContext.executionNo - Number of times the reduce function has been executed on the current group

\* @param {boolean} reduceContext.isRestarted - Indicates whether the current invocation of this function is the first

\* invocation (if true, the current invocation is not the first invocation and this function has been restarted)

\* @param {string} reduceContext.key - Key to be processed during the reduce stage

\* @param {List<String>} reduceContext.values - All values associated with a unique key that was passed to the reduce stage

\* for processing

\* @since 2015.2

\*/

const reduce = (reduceContext) => {

}

/\*\*

\* Defines the function that is executed when the summarize entry point is triggered. This entry point is triggered

\* automatically when the associated reduce stage is complete. This function is applied to the entire result set.

\* @param {Object} summaryContext - Statistics about the execution of a map/reduce script

\* @param {number} summaryContext.concurrency - Maximum concurrency number when executing parallel tasks for the map/reduce

\* script

\* @param {Date} summaryContext.dateCreated - The date and time when the map/reduce script began running

\* @param {boolean} summaryContext.isRestarted - Indicates whether the current invocation of this function is the first

\* invocation (if true, the current invocation is not the first invocation and this function has been restarted)

\* @param {Iterator} summaryContext.output - Serialized keys and values that were saved as output during the reduce stage

\* @param {number} summaryContext.seconds - Total seconds elapsed when running the map/reduce script

\* @param {number} summaryContext.usage - Total number of governance usage units consumed when running the map/reduce

\* script

\* @param {number} summaryContext.yields - Total number of yields when running the map/reduce script

\* @param {Object} summaryContext.inputSummary - Statistics about the input stage

\* @param {Object} summaryContext.mapSummary - Statistics about the map stage

\* @param {Object} summaryContext.reduceSummary - Statistics about the reduce stage

\* @since 2015.2

\*/

const summarize = (summaryContext) => {

}

return {getInputData, map, reduce, summarize}

});

