

# Advanced Searching with SuiteScript 2.1

written by [Eric T Grubaugh](#)

*part of the ["SuiteScript by Example" ↗](#) series*

published by [Stoic Software, LLC](#)

# Advanced Searching with SuiteScript 2.1

by [Eric T Grubaugh](#)

Copyright (c) 2017- [Stoic Software, LLC](#). All rights reserved.

Published by Stoic Software, LLC, PO Box 129, Wellington, CO 80549.

NetSuite and SuiteScript are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Neither the author nor the publisher have any affiliation with Oracle Corporation or NetSuite, Inc. This product is neither endorsed nor sponsored by Oracle Corporation or NetSuite, Inc.

## Using Code Samples

This book is here to help you learn. In general, you may use the code presented herein in your own code. You do not need to contact me unless you are reproducing or redistributing large portions of the code.

I appreciate, but do not require, attribution. An attribution usually includes the title, author, and publisher:

*"Advanced Searching with SuiteScript 2.1, by Eric T Grubaugh (Stoic Software, LLC).  
Copyright 2017 Stoic Software, LLC."*

# Introduction

This SuiteScript cookbook is intended to provide you with practical examples for creating complex searches with the SuiteScript API.

In *Advanced Searching with SuiteScript 2.1*, you'll see examples of:


- How to process very large (4,000+) result sets using Paging
- How many Results does this Search return? (Without having to run the Search twice)
- Which Employees have logged overtime this week? (How to use Summary Filters)
- How do I compare values between Search Columns?
- Which Time Entries were entered late? (How to use Formula Filters and Columns)
- Which Employees have zero time entries? (How to search for absence of something)

## Conventions in this Book

All code examples in this book use the ``require`` function for defining modules. This allows you to copy and paste the snippets directly into the debugger or your browser's developer console and run them.

The ``N/search`` module is always imported as ``s``.

``console.log`` is used for writing output to the browser console. If desired, you can replace these with calls to the [`N/log` module ↗](#) for writing to the Execution Log for the debugger.

For more on how to test SuiteScript in your browser's console, watch my  [How-To video](#).

# What if I have more than 4000 results?

SuiteScript's various [Search APIs](#) are limited in the number of Results they will retrieve:

- The `each` iterator will iterate through at most `4,000` Results
- `getRange` only allows retrieval of `1,000` Results at a time

What do you do when you need to process more than those limits?

## Option 1: Repeated Calls to `getRange`

While `getRange` is limited to `1,000` Results at a time, those `1,000` can be selected from any slice of the Result set. We can use this to progressively grab slices of `1,000` Results and concatenating them into a single Result set:

```

/**
 * Retrieves all active Customers, even if there are more than 4,000,
 * by successively concatenating chunks of 1,000 at a time
 *
 * Uses the `getRange` method repeatedly to retrieve all Results, regardless of limit
 *
 * @author Eric T Grubaugh <eric@stoic.software> (https://stoic.software/)
 */
require(['N/search'], (s) => {
  const customerSearch = s.create({
    type: s.Type.CUSTOMER,
    filters: [
      ['isinactive', s.Operator.IS, 'F']
    ],
    columns: ['entityid', 'email']
  }).run()

  const getAllResults = (search) => {
    let all = []
    let results = []

    const pageSize = 1000
    let start = 0
    let end = 1000

    do {
      results = search.getRange({ start, end })

      all = [...all, ...results]

      start += pageSize
      end += pageSize
    } while (results.length === pageSize)

    return all
  }

  const customers = getAllResults(customerSearch)
  console.log(customers.length)
})

```

We start by creating our Search object and executing it with `run`. In this example, we're retrieving the email address (`email`) and name (`entityid`) of all active Customers:

```

const customerSearch = s.create({
  type: s.Type.CUSTOMER,
  filters: [
    ['isinactive', s.Operator.IS, 'F']
  ],
  columns: ['entityid', 'email']
}).run()

```

Let's investigate the `getAllResults` function closely. It accepts a generic [search.ResultSet](#) object (the output of `run()`):

```
const getAllResults = (search) => {
```

It instantiates two Arrays:

1. `all` accumulates all of the results with each loop.
2. `results` holds only the results from a single search execution at a time.

```
let all = []  
let results = []
```

Next, it defines some variables which control how we repeat our search executions without retrieving the same results:

1. `pageSize` dictates how many results we'll try to retrieve with each execution. We set this to the maximum possible size allowed by `getRange()` so that we execute the fewest number of calls to the NetSuite database.
2. `start` will track the beginning index (inclusive) of the current "page" of results.
3. `end` will track the final index (exclusive) of the current "page" of results.

```
const pageSize = 1000  
let start = 0  
let end = 1000
```

We retrieve all [Results](#) for the Search by invoking `getRange`, combining the current page of results to any previous results, then advancing the `start` and `end` indices by one page. We stop when the page we retrieve is not a full page:

```
do {  
  // Retrieve one chunk of 1,000 Results  
  results = search.getRange({ start, end })  
  
  // Add this 1,000 to the end of the full list of Results  
  // The order is important to maintain any sorting on our Columns  
  all = [...all, ...results]  
  
  // Move to the next page  
  start += pageSize  
  end += pageSize  
  
  // Stop after we no longer receive a full page  
} while (results.length === pageSize)
```

Once the loop finishes, the `all` Array will contain all of our Search Results in a single data set, and we can return it as the output of our function.

```
return all
```

Once we have all Results in a single Array, we can process that Array however we choose in order to accomplish our business task.

We execute our Search by passing it into a custom function, `getAllResults` and then printing out the total number of results we've collected:

```
const customers = getAllResults(customerSearch)  
console.log(customers.length)
```

## Option 2: Paging API

In option 1, we were essentially building our own paging system. Thankfully, NetSuite has already done that for us, so there is no need to do it ourselves.

The `N/search` module also provides us with a [Paging API](#) for processing large `Result` sets, giving us fine-grained control over what constitutes a "Page" of data, and how we want to process it.

```

/**
 * Retrieves all active Customers, even if there are more than 4,000,
 * by successively concatenating Pages of 1,000 at a time
 *
 * Uses the Paging API to retrieve all Results, regardless of limits
 *
 * @author Eric T Grubaugh <eric@stoic.software> (https://stoic.software/)
 */
require(['N/search'], (s) => {
  const customerSearch = s.create({
    type: s.Type.CUSTOMER,
    filters: [
      ['isinactive', s.Operator.IS, 'F']
    ],
    columns: ['entityid', 'email']
  }).runPaged({ pageSize: 1000 })

  const getAllResults = (search) => {
    const all = []

    search.pageRanges.forEach((pageRange) => {
      let page = search.fetch({ index: pageRange.index })
      all = [...all, ...page.data]
    })

    return all
  }

  console.log(`Expected result count: ${customerSearch.count}`)

  const customers = getAllResults(customerSearch)
  console.log(`Actual result count: ${customers.length}`)
})

```

We're using the same code structure as we used in Option 1; we're executing the search, then collecting all the results with a custom `getAllResults` function.

To use the Paging API, we use the Search's `runPaged` method instead of `run`:

```

const customerSearch = s.create({
  type: s.Type.CUSTOMER,
  filters: [
    ['isinactive', s.Operator.IS, 'F']
  ],
  columns: ['entityid', 'email']
}).runPaged({ pageSize: 1000 })

```

Notice that we can control the number of Results per Page using the `pageSize` option of `runPaged`.

- minimum allowed `pageSize` is `5`



- maximum allowed `pageSize` is `1,000`
- default `pageSize` is `50`

`runPaged` returns a [search.PagedData](#) instance. From the `PagedData`, we iterate through its [PageRanges](#):

```
search.pageRanges.forEach((pageRange) => {  
  // ...  
})
```

Each `PageRange` contains a [Page](#) (fetched by its index), and each `Page` subsequently contains the `search.Result` data we can retrieve:

```
let page = search.fetch({ index: pageRange.index })  
all = [...all, ...page.data]
```

This is slightly more concise than our previous `do...while` attempt.

At this point we have a single Array containing all of our Results, so we are free to process that Array however we choose:

```
const customers = getAllResults(customerSearch)  
console.log(`Actual result count: ${customers.length}`)
```

## How many results are there?

Notice that a nice side effect of using `runPaged` is the `count` property on the `PagedData` instance it returns, which does not exist when we use `run`:

```
console.log(`Expected result count: ${customerSearch.count}`)
```

This is a much more performant way of getting the number of total Results from a Search, without having to use any Summary `COUNT` Columns or execute the Search twice.

# Which Employees have logged more than 40 hours this week?

There are many times we'll want to filter Search Results based on an aggregate value, like a `SUM`, `COUNT`, or `MAX`. To accomplish this, we can leverage *Summary Filters*.

Let's look at an example where we find Employees who have entered more than 40 hours of time this week.

```

/**
 * Finds all Employees who have entered more than 40 hours of time during
 * the current business week.
 *
 * Uses a Summary Filter to find Time Duration greater than 40
 *
 * @author Eric T Grubaugh <eric@stoic.software> (https://stoic.software/)
 */
require(['N/search'], (s) => {
  const overtimeSearch = s.create({
    type: s.Type.TIME_BILL,
    filters: [
      ['type', s.Operator.ANYOF, 'A'], 'and', // 'A' => 'Actual Time'
      ['date', s.Operator.WITHIN, 'thisBusinessWeek'], 'and',
      ['SUM(durationdecimal)', s.Operator.GREATERTHAN, 40]
    ],
    columns: [
      {
        name: 'employee',
        summary: s.Summary.GROUP
      }, {
        name: 'durationdecimal',
        summary: s.Summary.SUM
      }
    ]
  })

  const printEmployee = (result) => {
    const employeeName = result.getText({
      name: 'employee',
      summary: s.Summary.GROUP
    })
    const employeeHours = result.getValue({
      name: 'durationdecimal',
      summary: s.Summary.SUM
    })

    console.log(employeeName + ': ' + employeeHours)

    return true
  }

  console.log(overtimeSearch.runPaged().count)
  overtimeSearch.run().each(printEmployee)
})

```

We create our Time Bill search to find all Time Entries where the Type is *Actual Time*, the Date is within the current business week, and the Sum of the Duration is greater than `40`. In our Results, we want the Sum of the Duration grouped by Employee.

```
const overtimeSearch = s.create({
  type: s.Type.TIME_BILL,
  filters: [
    ['type', s.Operator.ANYOF, 'A'], 'and', // 'A' => 'Actual Time'
    ['date', s.Operator.WITHIN, 'thisBusinessWeek'], 'and',
    ['SUM(durationdecimal)', s.Operator.GREATERTHAN, 40]
  ],
  columns: [
    {
      name: 'employee',
      summary: s.Summary.GROUP
    }, {
      name: 'durationdecimal',
      summary: s.Summary.SUM
    }
  ]
})
```

Focus primarily on the ``durationdecimal`` Search Filter:

```
['SUM(durationdecimal)', s.Operator.GREATERTHAN, 40]
```

To summarize a Filter in a [Filter Expression](#), we wrap the name of our Filter field in the Summary function we want:

- ``SUM()`` for a summation
- ``MAX()`` for a maximum
- ``MIN()`` for a minimum
- ``COUNT()`` for a count
- ``AVG()`` for an average

If we are using ``Filter`` Objects instead, we could express this same Filter as:

```
{
  name: 'durationdecimal',
  summary
:
  s.Summary.SUM,
  operator
:
  s.Operator.GREATERTHAN,
  values
:
  40
}
```

After that, we can execute and process our Search like any other.

In this example, we use `runPaged()` to first record the number of search results, then we process the results by invoking a custom `printEmployee` function on each one:

```
console.log(overtimeSearch.runPaged().count)
overtimeSearch.run().each(printEmployee)
```

Note that calling `runPaged()` does consume an additional `5` units of governance, so don't automatically do this on any search that you execute unless you are already calling `runPaged()` anyway.

# Which Time Entries were created more than 7 days after the work was completed?

In order to perform calculations or comparisons on fields in our Search Results, we need to leverage NetSuite's Formula capabilities. When searching, we can leverage formulae in both Filters and Columns.

Let's look at an example to find all Time Entries where the entry was created more than 7 days after the work was actually completed.

```
/**
 * Retrieves the Time Entries that were created more than one week after
 * the work was completed by comparing the Date field to the Date Created
 * field on the Time Entry
 *
 * @author Eric T Grubaugh <eric@stoic.software> (https://stoic.software/)
 */
require(['N/search'], (s) => {
  // Formula for calculating the difference (in Days) between the Date
  // and the Date Created, rounded up with CEIL
  const daysElapsedFormula = 'CEIL({date}-{datecreated})'
  const lateEntriesSearch = s.create({
    type: s.Type.TIME_BILL,
    filters: [
      [`formulanumeric: ${daysElapsedFormula}`, s.Operator.GREATER_THAN, 7]
    ],
    columns: [
      'employee',
      'date',
      'datecreated',
      {
        name: 'formulanumeric',
        formula: daysElapsedFormula
      }
    ]
  })

  const resultToObject = (result) => ({
    employeeName: result.getText({ name: 'employee' }),
    daysElapsed: result.getValue({ name: 'formulanumeric' })
  })

  console.log(`# Late Entries = ${lateEntriesSearch.runPaged().count}`)
  const results = lateEntriesSearch.run().getRange({ start: 0, end: 1000 })
  const lateEntries = results.map(resultToObject)
  console.table(lateEntries)
})
```

We start by defining the formula itself:

```
// Formula for calculating the difference (in Days) between the Date
// and the Date Created, rounded up with CEIL
const daysElapsedFormula = 'CEIL({date}-{datecreated})'
```

While it's not necessary to put this into its own variable like this, I've done so to avoid repeating the same formula in both the Filter and the Column. This way when I need to change it, I can do so in one spot, and the change will be reflected everywhere it's necessary.

With the formula defined, we create our Search, specifying a Formula Filter and a Formula Column for showing the number of days elapsed from Date Created to Date:

```
const lateEntriesSearch = s.create({
  type: s.Type.TIME_BILL,
  filters: [
    [`formulanumeric: ${daysElapsedFormula}`, s.Operator.GREATER_THAN, 7]
  ],
  columns: [
    'employee',
    'date',
    'datecreated',
    {
      name: 'formulanumeric',
      formula: daysElapsedFormula
    }
  ]
})
```

Note that when you subtract two Date fields in a Formula, NetSuite will give you the number of Days between those two Dates.

Because the subtraction of two Dates results in a Number, we use `formulanumeric` rather than `formuladate`. The formula type you must choose depends on the *output* of your formula, *not on the inputs*.

Once again, we execute our Search and retrieve results like any other Search:

```
console.log(`# Late Entries = ${lateEntriesSearch.runPaged().count}`)
const results = lateEntriesSearch.run().getRange({ start: 0, end: 1000 })
```

In this case I want to `map` over all my Search Results and turn each one into a flat Object so that they are nicely printable by `console.table`.

The `resultToObject` function turns a single `Result` into a plain object:

```
const resultToObject = (result) => ({
  employeeName: result.getText({ name: 'employee' }),
  daysElapsed: result.getValue({ name: 'formulanumeric' })
})
```

Then we pass `resultToObject` as the iterator function for `map` so it will translate all the elements of the `results` Array:

```
const lateEntries = results.map(resultToObject)
console.table(lateEntries)
```

Notice how we read the value of the Formula Column by specifying `formulanumeric` as the `name`:

```
daysElapsed: result.getValue({ name: 'formulanumeric' })
```

## What happens if I have multiple Formula Columns?

Since we retrieve the value of a Formula Column by specifying `formulanumeric`, we also need a way to distinguish between multiple Formula Columns of the same type.

Let's add a non-rounded version of the same Formula to our Search Columns:



```

require(['N/search'], (s) => {
  // Formula for calculating the difference (in Days) between the Date
  // and the Date Created, rounded up with CEIL
  const daysElapsedFormula = 'CEIL({date}-{datecreated})'
  const lateEntriesSearch = s.create({
    type: s.Type.TIME_BILL,
    filters: [
      ['formulanumeric: ' + daysElapsedFormula, s.Operator.GREATER_THAN, 7]
    ],
    columns: [
      'employee',
      'date',
      'datecreated',
      {
        name: 'formulanumeric',
        formula: daysElapsedFormula
      },
      {
        name: 'formulanumeric',
        formula: '{date}-{datecreated}'
      }
    ]
  })

  const resultToObject = (result) => {
    const res = result.toJSON()
    console.log(res)
    return {
      employeeName: result.getText({ name: 'employee' }),
      daysElapsed: res.values.formulanumeric,
      daysElapsedNoRound: res.values.formulanumeric_1
    }
  }

  console.log(`# Late Entries = ${lateEntriesSearch.runPaged().count}`)
  const results = lateEntriesSearch.run().getRange({ start: 0, end: 1000 })
  const lateEntries = results.map(resultToObject)
  console.table(lateEntries)
})

```

First we've added our new non-rounded Formula Column:

```

columns: [
  // ...
  {
    name: "formulanumeric",
    formula: "{date}-{datecreated}"
  }
]

```

However, since both columns are technically named `formulanumeric`, `getValue` is unable to distinguish between them and would retrieve the last one defined.

In order to use multiple Formula Columns of the same type, we have to get a little creative here and turn the Search Result into a plain JavaScript Object using the Result's `toJSON()` method:

```
const res = result.toJSON()
console.log(res)
```

We log out the object to inspect its structure, and I highly recommend you study it closely for yourself.

From here, each subsequent `formulanumeric` gets a number appended to it, like `formulanumeric_1`; we use this knowledge to distinguish between our Formulae of the same type:

```
return {
  employeeName: result.getText({ name: 'employee' }),
  daysElapsed: res.values.formulanumeric,
  daysElapsedNoRound: res.values.formulanumeric_1
}
```

## Column Comparisons

When we're building Searches, it's common that we'll need to return the data for individual Columns, and also need to compare values between those Columns.

Formula Columns are the best way to accomplish these comparisons, like we've done in this example, comparing the Date to the Date Created on the Time Entry:

```
columns: [
  "employee",
  "date",
  "datecreated",
  {
    name: "formulanumeric",
    formula: "{date}-{datecreated}"
  }
]
```

In this instance, we happen to be subtracting the two, but you could use any operator or available [SQL formula](#) you choose to compare the values.

# Which Employees have no Time Entries in the past week?

The majority of the time that we're building searches, we're searching for the *existence* of Records that meet our criteria. Once in a while, however, we actually need to search for the *absence* of something, and this can often be trickier.

For example, which Employees have *no* Time Entries for this week?

Because we are looking for the *absence* of a Record, we'll actually need to combine the Results from two different searches. If the Records don't exist, we can't find them directly, so we take a slightly different approach:

```

/**
 * Finds Employees that have no Time Entries for the last week.
 *
 * Combines the Results from two different searches in order to search
 * for the absence of Records.
 *
 * @author Eric T Grubaugh <eric@stoic.software> (https://stoic.software/)
 */
require(['N/search'], (s) => {
  const employeeSearch = s.create({
    type: s.Type.EMPLOYEE,
    filters: [
      ['isinactive', s.Operator.IS, 'F']
    ],
    columns: [
      { name: 'formulatext', formula: "{firstname} || ' ' || {lastname}" }
    ]
  })

  const timeSearch = s.create({
    type: s.Type.TIME_BILL,
    filters: [
      ['date', s.Operator.WITHIN, 'thisBusinessWeek'], 'and',
      ['type', s.Operator.ANYOF, 'A'] // 'A' => 'Actual Time'
    ],
    columns: [
      { name: 'employee', summary: s.Summary.GROUP }
    ]
  })

  // Display result counts from each search
  console.log(`# Employees = ${employeeSearch.runPaged().count}`)
  console.log(`# Employees with Entries = ${timeSearch.runPaged().count}`)

  // Assume there are less than 1,000 Employees
  const employees = employeeSearch.run().getRange({ start: 0, end: 1000 })
  const timeEntries = timeSearch.run().getRange({ start: 0, end: 1000 })

  // Does the given employee have at least one Time Entry in entries Array?
  const employeeHasEntry = (entries, employee) =>
    entries.some((entry) => {
      let entryEmployee = entry.getValue({
        name: 'employee',
        summary: s.Summary.GROUP
      })
      return (entryEmployee == employee.id)
    })

  // Returns a copy of employees Array but removes any Employees
  // that *do* have Time Entries in entries Array
  const findEmployeesWithNoEntries = (employees, entries) =>
    employees.filter((employee) => !employeeHasEntry(entries, employee))

  const printEmployeeName = (result) =>
    console.log(result.getValue({ name: 'formulatext' }))

```

```

const employeesWithNoEntries = findEmployeesWithNoEntries(
  employees,
  timeEntries
)

console.log(`# Employees With No Entries = ${employeesWithNoEntries.length}`)
employeesWithNoEntries.forEach(printEmployeeName)
})

```

We first get the list of all active Employees *and* the list of all Employees who *have* created Time Entries for this week:

```

const employeeSearch = s.create({
  type: s.Type.EMPLOYEE,
  filters: [
    ['isinactive', s.Operator.IS, 'F']
  ],
  columns: [
    { name: 'formulatext', formula: "{firstname} || ' ' || {lastname}" }
  ]
})

const timeSearch = s.create({
  type: s.Type.TIME_BILL,
  filters: [
    ['date', s.Operator.WITHIN, 'thisBusinessWeek'], 'and',
    ['type', s.Operator.ANYOF, 'A'] // 'A' => 'Actual Time'
  ],
  columns: [
    { name: 'employee', summary: s.Summary.GROUP }
  ]
})

console.log(`# Employees = ${employeeSearch.runPaged().count}`)
console.log(`# Employees with Entries = ${timeSearch.runPaged().count}`)

const employees = employeeSearch.run().getRange({ start: 0, end: 1000 })
const timeEntries = timeSearch.run().getRange({ start: 0, end: 1000 })

```

So far there's nothing new here; we're running two separate Searches.

Now in order to determine which Employees don't have any Time Entries, we can "subtract" the Employees who *do* have Time Entries from the list of *all* Employees. This will leave us only with Employees who *do not* have Time Entries.

To accomplish this, we can leverage the JavaScript Array's [filter`](#) and [some`](#) methods.

First, we write the `employeeHasEntry`` function that accepts an Array of Time Entry Search Results and a single Employee Search Result. Its job is to determine whether a specific

Employee has a Time Entry:

```
const employeeHasEntry = (entries, employee) =>
  entries.some((entry) => {
    let entryEmployee = entry.getValue({
      name: 'employee',
      summary: s.Summary.GROUP
    })
    return (entryEmployee == employee.id)
  })
```

We use `some` to determine if the given `employee` exists within the `entries` Array.

Now that we can detect whether a single Employee has any Time Entries, we need to extend that over the full list of Employees. This falls to `findEmployeesWithNoEntries`:

```
const findEmployeesWithNoEntries = (employees, entries) =>
  employees.filter((employee) => !employeeHasEntry(entries, employee))
```

We use `employeeHasEntry` as the [predicate](#) of a `filter` to remove the Employees that *do* have a Time Entry from `employees`.

We can now pass in our two separate Result Arrays and list the Employees that have no Time Entries this week:

```
const employeesWithNoEntries = findEmployeesWithNoEntries(
  employees,
  timeEntries
)

console.log(`# Employees With No Entries = ${employeesWithNoEntries.length}`)
employeesWithNoEntries.forEach(printEmployeeName)
```

This will print out the number of Employees with no Time Entries, followed by the list of their names.

# What's the Internal ID for this Search Result?

Every Search Result in SuiteScript is fundamentally a reference to a Record in NetSuite. It's common to run a Search and then want to use the Internal ID for the Record that the Result represents.

In fact, it's so common that every Search Result in SuiteScript has an `id` property that contains the corresponding Record's internal ID; there is rarely a need to explicitly add `internalid` as a Search Column in your Search.

```
require(['N/search'], function (s) {
  const plainSearch = s.create({
    type: s.Type.EMPLOYEE,
    filters: [
      ['isinactive', s.Operator.IS, 'F']
    ],
    columns: [
      { name: 'formulatext', formula: "{firstname} || ' ' || {lastname}" }
    ]
  })

  const printEmployeeId = (result) => {
    console.log(result.id)
    return false // only process first result
  }

  console.log('IDs for Plain Search:')
  plainSearch.run().each(printEmployeeId)
})
```

This will print the internal ID of every active Employee by accessing `result.id`; no `internalid` Column needed.

## Except...

As with all things NetSuite, however, there is an exception to this: Summary Columns.

*Summarized* Search Results no longer reference a single Record, but rather an aggregate of Records; for summarized Search Results, the `id` property will be `undefined`, and thus not very helpful. However, it's common to summarize your Search Results, and also need to drill down into the data for the individual Records that make up the summary.

Let's modify our example to group the Employees by their Hire Date:

```
s.create({
  type: s.Type.EMPLOYEE,
  filters: [
    ['isinactive', s.Operator.IS, 'F']
  ],
  columns: [
    { name: 'hiredate', summary: s.Summary.GROUP },
    {
      name: 'formulatext',
      formula: "{firstname} || ' ' || {lastname}",
      summary: s.Summary.GROUP
    }
  ]
})
```

How do we access the Internal ID for Search Results within a Summary?

We first add an `internalid` Search Column:

```
columns: [
  { name: 'hiredate', summary: s.Summary.GROUP },
  {
    name: 'formulatext',
    formula: "{firstname} || ' ' || {lastname}",
    summary: s.Summary.GROUP
  },
  // Add the internalid Column
  { name: 'internalid', summary: s.Summary.GROUP }
]
```

Then we need to access it using `getValue` instead of `id`:

```
const printEmployeeId = (result) => {
  // Utilize getValue instead of .id
  console.log(result.getValue({ name: 'internalid', summary: s.Summary.GROUP }))
  return false // only process first result
}
```



# Frequently Asked Questions

## How do I find the details on NetSuite's SQL formulas?

The Help page titled [SQL Expressions ↗](#) contains all the reference material for the supported SQL functions you can utilize.

## How do I find the name/ID for a specific Filter?

Find your Record Type in the [Records Browser ↗](#), and explore the "Search Filters" section. The value in the *Internal ID* column is what you'll use as your Filter name.

## How do I find the name/ID for a specific Column?

Find your Record Type in the [Records Browser ↗](#), and explore the "Search Columns" section. The value in the *Internal ID* column is what you'll use as your Column name.

## How do I find the name/ID for a specific Join?

Find your Record Type in the [Records Browser ↗](#), and explore the "Search Joins" section. The value in the *Join ID* column is what you'll use as your Join name.

## There's a Records Browser, a Schema Browser, and a Connect Browser. What's the difference?

- *Records Browser* - Used for accessing Record data via *SuiteScript*
- *Schema Browser* - Used for accessing Record data via *SuiteTalk*
- *Connect Browser* - Used for accessing Record data via *ODBC*

When you're writing SuiteScript, you can safely focus only on the *Records Browser*.

# Recommendations and Resources

## NetSuite Help

NetSuite Help is the most definitive reference for the `N/search`` module and all of its capabilities. I recommend studying the following articles and any related sub-articles:

- [N/search Module ↗](#)
- [N/search Module Script Samples ↗](#)
- [search.Type ↗](#)
- [search.Summary ↗](#)
- [SuiteScript 2.x Search Operators ↗](#)
- [search.Filter ↗](#)
- [search.filterExpression ↗](#)
- [search.Column ↗](#)
- [search.Operator ↗](#)
- [Summary Type Descriptions ↗](#)
- [Search Date Filters ↗](#)
- [Search.filters ↗](#)
- [search.filterExpression ↗](#)
- [search.Column ↗](#)
- [SQL Expressions ↗](#)

## Use the Search UI

A great way to both learn about and verify your SuiteScript searches is to actually build the search in the UI first, then translate it into SuiteScript.

By doing this, you can quickly verify that the Filters and Columns you're specifying actually give you the correct results before you even start writing code.

## NetSuite Search Export Chrome Plugin

There is a helpful Chrome Plugin called "NetSuite Search Export" built by [in David Smith](#). The Plugin will automatically generate SuiteScript for any Saved Search in your account. You can find it [on the Chrome Plugin Store ↗](#)

To use this plugin:

1. Create a Saved Search in the UI
2. Save the search
3. Click the "Export to Script" link near the top right

## Searching with SuiteScript Playlist

I have a [📺 playlist on YouTube](#) containing several videos and examples of searching in SuiteScript.

## The Records Browser

The [Records Browser ↗](#) is an absolutely crucial tool for creating effective searches. There is a new version of the Records Browser for every version of NetSuite. The 2023.2 version can be found [in NetSuite Help ↗](#).

If you are unfamiliar with the Records Browser, see [SuiteScript Records Browser ↗](#) in the Help documentation and [my tutorial](#).

## Mozilla Developer Network

SuiteScript is a library on top of JavaScript, and the best JavaScript reference manual is the [Mozilla Developer Network ↗](#).

While not related specifically to NetSuite, this site is an excellent source of JavaScript reference material, examples, and tutorials.

# About the Author



My name is [in Eric T Grubaugh](#). I run the *Sustainable SuiteScript* community for NetSuite developers. I founded Stoic Software in 2016 to help others lead successful, sustainable careers as NetSuite developers.

## The "Sustainable SuiteScript" Community

We are a small community of NetSuite developers who want to deepen their technical skills, expand their professional network, and raise the bar for SuiteScript development. [Join us](#) today.

## Questions, Comments, Corrections

If you have any questions, comments, or corrections on this document, please email them to me at [eric+cookbooks@stoic.software](mailto:eric+cookbooks@stoic.software).

## Get in Touch

The best way to keep in regular contact with me is to join the [Sustainable SuiteScript](#) mailing list. I read and respond to all emails I receive there.

I create SuiteScript videos on [YouTube](#).

You can also connect with me on [LinkedIn](#).

