

Basic Optimization with SuiteScript 2.1

written by [Eric T Grubaugh](#)

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

published by [Stoic Software, LLC](#)

Basic Optimization 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:

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

Introduction

`“`Error: SCRIPT_EXECUTION_USAGE_LIMIT_EXCEEDED`”`

How many times has this error ruined your day?

Basic Optimization with SuiteScript 2.1 is intended to provide you with practical examples for reducing governance usage, reducing execution time, and increasing performance in your SuiteScript.

In this cookbook, you'll see advice on:

- Minimizing Log Entries
- Leveraging Inline Editing
- Consolidating Lookups and Inline Edits
- Eliminating Loads or Lookups on Associated Records
- Eliminating Repeated Lookups of Same Type
- Avoiding Searches or Queries from Loops
- Reducing Repetition in Common SuiteScript Tasks

Conventions in this Book

All code examples are written in *SuiteScript 2.1*.

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

Minimize the Number of Log Entries

Logging is an extremely useful tool for tracing and debugging the execution of a script, but it can be detrimental to the performance of your scripts.

Years ago, I was working on a team building a SuiteApp, and we had a Scheduled Script responsible for bulk processing records. This script had a significant amount of logic in it, and could go through several different paths, so we had a correspondingly large amount of logging in the script. The average execution time for this script in Production was consistently around ``45`` minutes, but we had always attributed that to the amount of records being processed and the various logic paths the script needed to follow.

One day, as an experiment, I suggested we change the Log Level of the script from ``DEBUG`` to ``ERROR``. This had absolutely no effect on performance. Later, one of our developers commented out *all* logging done by the script. We *immediately* saw the execution time drop to around *7 minutes*.

There are several patterns to watch out for that can help you minimize the impact of logging on your performance while maintaining its usefulness to your debugging efforts. While you may not necessarily see the drastic 85% reduction in execution time that we happened to see, it is a good practice to use logging efficiently, rather than pervasively.

Condense multiple similar log entries into a single entry:

Each log entry your scripts write creates an instance of a Script Execution Log record, so while logging doesn't use up any governance units, it does carry about the same performance penalty of creating an instance of a custom record.

Changing the Log Level on a Script Deployment has no effect on this penalty because the request is *always* sent to the server *before* the Log Level is checked. This request forms the majority of the overhead in the log functions.

Because of these factors, we want to minimize the amount of calls we make to the ``log`` module. We can do this by taking advantage of the fact that NetSuite will automatically invoke ``JSON.stringify()`` on any Object passed in to the ``details`` parameter.

Turn this:

```

log.debug({
  title: 'Name',
  details: record.getValue({ 'fieldId': 'name' })
})
log.debug({
  title: 'Sales Rep',
  details: record.getValue({ 'fieldId': 'salesrep' })
})
log.debug({
  title: 'Customer',
  details: record.getValue({ 'fieldId': 'entity' })
})
log.debug({
  title: 'Location',
  details: record.getValue({ 'fieldId': 'location' })
})

```

into this:

```

const data = {
  name: record.getValue({ 'fieldId': 'name' }),
  salesRep: record.getValue({ 'fieldId': 'salesrep' }),
  customer: record.getValue({ 'fieldId': 'entity' }),
  location: record.getValue({ 'fieldId': 'location' })
}
log.debug({ title: 'Record Data:', details: data })

```

In Client Scripts, use Developer Console logs instead of NetSuite Execution Logs

Writing your Client Script's logs to the Developer Console is much faster than writing NetSuite Execution Logs as there is no server request to make and no log record to create.

No matter what the Log Level set on your Script Deployment, NetSuite will send an HTTP request for each NetSuite log you try to write from a Client Script. To avoid the overhead of all these requests, it's best to leverage console logging for your Client Scripts.

Turn this:

```

log.debug({
  title: 'Sales Rep:',
  details: record.getValue({ fieldId: 'salesrep' })
})

```

into this:

```
console.log(`Sales Rep: ${record.getValue({ fieldId: 'salesrep' })}`)
```

“⚠ Be conscious not to expose any sensitive data in your console logs. Any user who knows how to open the browser console will be able to see your logs. Never log secure information like passwords or tokens to the console.”

Prefer Inline Editing over Loading and Saving

Loading and submitting a record is one of the most expensive tasks - in terms of both governance and time - that you can perform in SuiteScript. It is crucial to understand that this is not your only option for editing existing records.

When you only need to edit the *body* fields of a record, you can leverage [inline editing](#) using the [`record.submitFields` method](#) from the [`N/record` module](#).

This has several advantages:

- ``submitFields`` uses fewer governance units than ``load() + save()``.
- ``submitFields`` will generally execute faster because there are likely to be fewer User Event Scripts and Workflows responding to the ``XEDIT`` event it fires.
- ``submitFields`` is much more concise than repeated ``setValue()`` calls.

Turn this:

```
const rec = record.load({
  type: record.Type.EMPLOYEE,
  id: '123'
})
rec.setValue({
  fieldId: 'firstname',
  value: 'Eric'
})
rec.setValue({
  fieldId: 'middlename',
  value: 'T'
})
rec.setValue({
  fieldId: 'lastname',
  value: 'Grubaugh'
})
rec.setValue({
  fieldId: 'title',
  value: 'SuiteScript Strategist'
})
rec.save()
```

into this:

```
record.submitFields({
  type: record.Type.EMPLOYEE,
  id: '123',
  values: {
    firstname: 'Eric',
    middlename: 'T',
    lastname: 'Grubaugh',
    title: 'NetSuite Developer Advisor'
  }
})
```

`submitFields` also contains some options that enable it to run faster as well:

- `enablesourcing` determines whether [sourcing](#) will be performed on the updated record. This defaults to `true`, so sourcing is on by default.
- `ignoreMandatory` determines whether mandatory field validation is performed on the updated record. This defaults to `false`, so mandatory field validation is performed by default.

You can speed up the submission process further by disabling both of these options when it makes sense to do so. You pass them in an `options` parameter, like so:

```
record.submitFields({
  type: record.Type.EMPLOYEE,
  id: '123',
  values: {
    firstname: 'Eric',
    middlename: 'T',
    lastname: 'Grubaugh',
    title: 'NetSuite Developer Advisor'
  },
  options: {
    enablesourcing: false, // disable sourcing
    ignoreMandatory: true // disable mandatory validation
  }
})
```

There are a few limitations with `record.submitFields` to be aware of:

- You cannot use `submitFields` to modify fields in a sublist.
- You cannot use `submitFields` to modify fields on a [subrecord](#).
- Not all body fields support inline editing. See the `nlapiSubmitField` column of the [Records Browser](#) to know whether a specific field supports inline editing. If you use inline editing on a field that doesn't support it, NetSuite will still execute the function properly, but behind the scenes it will load and submit the entire record, thus utilizing more governance and time.

Note that `submitFields` will fire an `XEDIT` [Event type ↗](#), whereas `load` and `save` will fire a `VIEW` Event followed by an `EDIT` Event. Keep this in mind if you need other Scripts and Workflows to respond to your `submitFields` call.

Consolidate Lookups or Inline Edits of the Same Record

Both the Lookup and Inline Edit functionality of SuiteScript support working with multiple fields in a single call, but I often see this feature overlooked. If a script is running multiple Lookups or Inline Edits of the same record, then it is utilizing far more governance and execution time than it should.

Turn this:

```
const orderCustomer = search.lookupFields({
  type: search.Type.SALES_ORDER,
  id: '123',
  columns: 'entity'
})
const orderRep = search.lookupFields({
  type: search.Type.SALES_ORDER,
  id: '123',
  columns: 'salesrep'
})
const orderLocation = search.lookupFields({
  type: search.Type.SALES_ORDER,
  id: '123',
  columns: 'location'
})
```

into this:

```
const orderData = search.lookupFields({
  type: search.Type.SALES_ORDER,
  id: '123',
  columns: ['entity', 'salesrep', 'location']
})
```

Eliminate Multiple Lookups of Associated Records

Another oft-overlooked feature of Lookups is their support of joined fields. I will often see the results of Lookups from one record used to lookup data on a related record. This is not necessary. When you need body-field data from a related record, you can utilize the join syntax supported by the Lookup functionality.

Turn this:

```
const salesRepId = search.lookupFields({
  type: search.Type.SALES_ORDER,
  id: '123',
  columns: 'salesrep'
}).salesrep[0].value

const salesRepEmail = search.lookupFields({
  type: search.Type.EMPLOYEE,
  id: salesRepId,
  columns: 'email'
}).email
```

into this:

```
const salesRepEmail = search.lookupFields({
  type: search.Type.SALES_ORDER,
  id: '123',
  columns: 'salesrep.email'
})['salesrep.email']
```

Eliminate Repeated Lookups of the Same Record Type

SuiteScript's Lookup functionality provides us with a quick way to retrieve body field data from a specific record. Often, you will need to look up the same fields from different records of the same type; for instance, you may need to retrieve the web store description for all items on a Sales Order.

Behind the scenes, a Lookup is a Search that filters on record type and a specific internal ID. If you are repeatedly performing a Lookup on records of the same type, you can instead run a single Search to get all the data at once.

With any more than a handful of repeated Lookups, the Search will quickly break even, using less governance and taking less time than the repeated Lookups.

Turn this:

```
const itemCount = rec.getLineCount({ sublistId: 'item' })
for (let i = 0; i < itemCount; i++) {
  const itemDescription = search.lookupFields({
    type: search.Type.ITEM,
    id: rec.getSublistValue({
      sublistId: 'item',
      fieldId: 'item',
      line: i
    }),
    columns: 'storedetaileddescription'
  }).storedetaileddescription

  // Do something with itemDescription
  // ...
}
```

into this:

```
const itemCount = rec.getLineCount({ sublistId: 'item' })
const itemIds = [...Array(itemCount)].map((n, line) => rec.getSublistValue({
  sublistId: 'item',
  fieldId: 'item',
  line
}))

const itemResults = search.create({
  type: search.Type.ITEM,
  filters: [['internalid', 'anyof', itemIds]],
  columns: ['storedetaileddescription']
}).run()

itemResults.each( /* Do something with each Item Description */)
```

By replacing the repeated Lookups with a single Search, we collect all the necessary data in one place, and we are utilizing less governance and time if there are more than a few items. These savings in governance and time will more than make up for the penalty we incur from an additional loop to process the search results.

Avoid Searching or Querying from a Loop

Turn this:

```
const lineCount = salesOrder.getLineCount(/* ... */)
for (let i = 0; i < lineCount; i++) {
  let itemId = salesOrder.getSublistValue(/* ... */)
  let results = search.create({
    // ...
    filters: [
      // ...
      ['internalid', search.Operator.ANYOF, itemId], 'AND',
      // ...
    ]
    // ...
  }).run().each(/* Do something with results */)
}
```

into this:

```
const itemIds = []
const lineCount = salesOrder.getLineCount({ sublistId: 'item' })
// Collect all the necessary input values in the loop
for (let i = 0; i < lineCount; i++) {
  let itemId = salesOrder.getSublistValue({
    sublistId: 'item',
    fieldId: 'item',
    line: i
  })
  itemIds.push(itemId)
}
// Move the search outside the loop and use all the input values at once
let results = search.create({
  type: 'item',
  filters: [
    // ...
    ['internalid', search.Operator.ANYOF, itemIds], 'AND'
    // ...
  ]
  // ...
}).run().each(/* Do something with results */)
```

The same approach applies if it's a Query inside your loop instead of a Search.

Reduce Repetition

While they won't improve the performance of your script, there are also more concise ways to express many of the common SuiteScript tasks we do. Being more concise without sacrificing legibility of your code will often improve the developer experience and reduce the cognitive overhead of reading and maintaining it.

Retrieval of Sublist Values

The retrieval of sublist values from every line is one common task, as we do for the ``itemIds`` in the previous example.

A pattern I like to follow for populating data structures from sublists goes something like:

```
const lineCount = salesOrder.getLineCount({ sublistId: 'item' })
// Create an Array the same size as the sublist, empty values become undefined
const itemIds = [...Array(lineCount)]
// Transform the undefined elements into values from the sublist
.map((n, line) => salesOrder.getSublistValue({
  sublistId: 'item',
  fieldId: 'item',
  line
}))
```

``Array(lineCount)`` creates an Array filled with ``lineCount`` number of [empty slots](#). [Spreading](#) the Array converts the *empty* slots into ``undefined`` elements.

```
>> Array(5)
< > Array(5) [ <5 empty slots> ]
>> [...Array(5)]
< > Array(5) [ undefined, undefined, undefined, undefined, undefined ]
```

There is a [significant difference](#) in how different Array methods handle *empty* values versus how they handle ``undefined`` or ``null`` values, and it's worth a read to understand.

For our purposes here, suffice it to say that ``map()`` never visits an *empty* element, but does visit an ``undefined`` one, so we use ``...`` to convert *empty* to ``undefined`` first, then ``map`` over the result.

You could use the same approach to retrieve multiple values from each sublist line instead rather than only one.

Reusing Repeated Function Parameters

Many SuiteScript API methods share multiple common parameter names that get repeated over and over, especially when reading or populating records.

If you do not like the repetition of parameters like ``sublistId`` and ``line``, you can store those as objects and take advantage of the spread (``...``) operator:

```
// Reuse the sublistId property for getLineCount and getSublistValue
const itemSublist = { sublistId: 'item' }
const lineCount = salesOrder.getLineCount({ ...itemSublist })

const itemData = [...Array(lineCount)]
  .map((n, line) => {
    // Reuse the sublistId and line properties for getSublistValue
    const itemLine = { ...itemSublist, line }
    return {
      itemId: salesOrder.getSublistValue({ ...itemLine, fieldId: 'item' }),
      itemQty: salesOrder.getSublistValue({ ...itemLine, fieldId: 'quantity' }),
      itemAmt: salesOrder.getSublistValue({ ...itemLine, fieldId: 'amount' })
    }
  })
/* =>
itemData = [
  { itemId: 123, itemQty: 75, itemAmt: 25000 },
  { itemId: 456, itemQty: 28, itemAmt: 467.33 },
  { itemId: 789, itemQty: 13, itemAmt: 1285.27 }
]
*/
```


Recommendations and Resources

NetSuite Help

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

- [Optimizing SuiteScript Performance ↗](#)
- [Setting Script Execution Log Levels ↗](#)
- [Governance on Script Logging ↗](#)
- [search.lookupFields\(options\) ↗](#)
- [record.submitFields\(options\) ↗](#)

SuiteScript Videos and Articles

- [Working with Records in SuiteScript](#)
- [XEDIT Events](#)
- [📺 Field Lookups in SuiteScript 2.0](#)
- [📺 Inline Editing in SuiteScript 2.0](#) video
- [Inline Editing in SuiteScript 2.0](#) article
- [Inline Editing Dropdown Fields](#)
- [📺 Logging in SuiteScript 2.0](#)
- [📺 NetSuite's Governance System](#)

Browser Developer Console References

- [Chrome ↗](#)
- [Firefox ↗](#)
- [Safari ↗](#)

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.

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](#).

