Gouvernance SuiteScript 2.x : record.load vs lookupFields

Publié le 9 mai 2026 23 min de lecture


Gouvernance SuiteScript 2.x : record.load vs lookupFields

Résumé analytique

SuiteScript 2.x est l'API JavaScript moderne de NetSuite destinée à la personnalisation et à l'automatisation des applications NetSuite. Elle fonctionne selon un modèle d'utilisation strict : chaque appel d'API consomme une *unité de gouvernance*, et chaque type de script dispose d'un quota fixe par exécution (Source: docs.oracle.com) (Source: www.houseblend.io). Notamment, le chargement complet d'un enregistrement via `record.load` est relativement coûteux (par exemple, 10 unités pour un enregistrement de transaction) (Source: docs.oracle.com) (Source: www.houseblend.io), tandis que la récupération de champs spécifiques par ID via `search.lookupFields` ne coûte qu'une seule unité (Source: docs.oracle.com) (Source: www.houseblend.io). Cette disparité importante a des conséquences majeures sur l'optimisation des performances. Nous constatons que **minimiser les opérations coûteuses** (par exemple, remplacer les chargements d'enregistrements par des recherches de champs, utiliser des recherches enregistrées, etc.) est crucial pour éviter de dépasser les limites de gouvernance (Source: www.houseblend.io) (Source: www.houseblend.io).

Dans ce rapport, nous analysons en profondeur l'impact sur les unités de gouvernance de `record.load` par rapport à `search.lookupFields` dans SuiteScript 2.x. Nous commençons par examiner le modèle de gouvernance de NetSuite et les limites d'utilisation pour divers types de scripts (Source: docs.oracle.com) (Source: docs.oracle.com), incluant une comparaison des quotas au niveau des scripts dans le Tableau 1. Nous examinons ensuite le comportement et les coûts d'utilisation des appels d'API clés. Comme détaillé dans la documentation officielle de SuiteScript, `record.load(options)` consomme **10 unités** pour les enregistrements de transaction standard, **2 unités** pour les enregistrements personnalisés et **5 unités** pour les autres enregistrements (Source: docs.oracle.com) (Source: www.houseblend.io). En revanche, `search.lookupFields(options)` consomme toujours **1 unité**, quel que soit le type d'enregistrement (Source: docs.oracle.com) (Source: www.houseblend.io). Notre analyse souligne comment l'exploitation de `search.lookupFields` (et d'API légères similaires) peut réduire considérablement l'utilisation. Par exemple, un blog de développeur observe qu'une recherche de champ coûte toujours 1 unité, contre 10 unités pour une itération complète sur un `resultSet` de recherche (Source: hutada.home.blog).

Nous complétons ces conclusions par des données empiriques et des cas réels. Par exemple, une étude a révélé qu'une recherche de champs simples prenait en moyenne ~0,03 seconde, ce qui est comparable ou plus rapide qu'une requête SuiteQL (Source: www.nzrsolutions.com). Dans les scripts réels, le remplacement des appels répétés à `record.load` par `search.lookupFields` ou des recherches enregistrées a permis aux scripts de rester dans les quotas d'utilisation (Source: www.houseblend.io) (Source: www.houseblend.io). Nous incluons des comparaisons détaillées (voir Tableau 2) et des études de cas : par exemple, un Suitelet d'importation en volume qui exécutait initialement 30 unités par itération (via `record.load` + `record.save`) a été refactorisé pour traiter des lots de 200 enregistrements et récupérer les données associées via des recherches, évitant ainsi les régulateurs★ (Source: www.houseblend.io). De même, Anchor Group rapporte que le chargement de 200 enregistrements d'articles coûte 1 000 unités (limite d'événement utilisateur), tandis qu'une recherche d'article unique avec des colonnes limitées ne coûte que 10 unités, quel que soit le nombre (Source: www.anchorgroup.tech) (Source: www.anchorgroup.tech).

Notre examen approfondi souligne qu'une **sélection minutieuse des méthodes et une conception de requête adaptée** sont essentielles. Nous fournissons des conseils basés sur les données et des bonnes pratiques (mise en cache, traitement par lots, filtres indexés, etc.) pour maintenir les performances et éviter les dépassements (Source: www.houseblend.io) (Source: www.houseblend.io). Enfin, nous discutons des approches émergentes (par exemple, SuiteQL, N/query) et de leur intégration dans le modèle de gouvernance.

Introduction et contexte

La plateforme **SuiteScript** de NetSuite permet aux développeurs d'étendre et d'automatiser NetSuite (une plateforme ERP/CRM cloud) en utilisant JavaScript. L'API SuiteScript 2.x (la version majeure actuelle, incluant 2.0 et 2.1) fournit des primitives de script modulaires, asynchrones, côté serveur ou côté client (Source: docs.oracle.com) (Source: docs.oracle.com). Le code SuiteScript peut être déployé en tant que **scripts d'événement utilisateur** (déclenchés lors d'opérations sur les enregistrements), **scripts planifiés**, **Suitelets**, **RESTlets**, **scripts Map/Reduce**, et plus encore. SuiteScript 2.x est une API moderne publiée après SuiteScript 1.0, avec une modularité améliorée et la prise en charge des promesses (Source: docs.oracle.com) (Source: docs.oracle.com).

Un aspect critique du développement SuiteScript est la **gouvernance des performances**. NetSuite impose des **unités de gouvernance** (également appelées *unités d'utilisation*) pour mesurer la consommation des ressources par les scripts (Source: docs.oracle.com). Chaque invocation d'une méthode d'API SuiteScript consomme un nombre fixe d'unités d'utilisation, reflétant le coût de traitement sous-jacent. Si un script dépasse ses unités allouées, NetSuite le termine (en générant des erreurs telles que `SSS_USAGE_LIMIT_EXCEEDED`) (Source: docs.oracle.com). Par conséquent, une utilisation efficace des méthodes d'API est essentielle pour éviter l'échec des scripts, en particulier avec de grands volumes de données.

Les unités d'utilisation sont suivies à deux niveaux : par **type de script** et par **appel d'API** (Source: docs.oracle.com). Chaque exécution de script a un quota maximum ; le Tableau 1 (ci-dessous) résume les types de scripts SuiteScript 2.x courants et leurs limites d'unités. Par exemple, les scripts **User Event** et **Client** disposent de 1 000 unités par exécution (Source: docs.oracle.com) (Source: docs.oracle.com), tandis que les scripts **Scheduled** et **RESTlet** disposent respectivement de 10 000 et 5 000 unités (Source: docs.oracle.com) (Source: docs.oracle.com). Les scripts Map/Reduce sont uniques : le traitement global n'est pas plafonné, mais chaque étape individuelle **getInputData**, **map**, **reduce** ou **summarize** a sa propre limite (souvent 10 000 par étape) (Source: docs.oracle.com) (Source: docs.oracle.com).

Tableau 1. Limites d'unités d'utilisation par type de script (SuiteScript 2.x)

TYPE DE SCRIPT	UNITÉS D'UTILISATION MAX. PAR EXÉCUTION	NOTES ET RÉFÉRENCES
User Event Script (2.x)	1 000	Par exécution sur les événements d'enregistrement (Source: docs.oracle.com). Ces scripts s'exécutent de manière synchrone lors de l'enregistrement/soumission.
Client Script (2.x)	1 000	S'exécute dans le contexte du navigateur ; 1 000 unités par script (niveau formulaire et niveau enregistrement) (Source: docs.oracle.com).
Suitelet (2.x)	1 000	S'exécute à la demande via une URL ; limites similaires aux autres scripts serveur [(Source: docs.oracle.com)].
Scheduled Script (2.x)	10 000	Tâches ponctuelles ou récurrentes ; limite de 10 000 unités par exécution de script (Source: docs.oracle.com).
Map/Reduce (2.x)	*Aucune limite fixe*	Intensif en données ; pas de plafond total de script, mais chaque étape map/reduce est limitée (par ex. 10k par étape) (Source: docs.oracle.com) (Source: www.houseblend.io).
RESTlet Script (2.x)	5 000	Exposé via l'API REST ; 5 000 unités par requête REST (Source: docs.oracle.com).
Mass Update Script (2.x)	1 000	Déclenché pour la mise à jour en masse d'enregistrements ; 1 000 unités par enregistrement ou exécution (Source: docs.oracle.com).
Portlet Script (2.x)	1 000	S'exécute dans les tableaux de bord classiques ou améliorés ; 1 000 unités par invocation (Source: docs.oracle.com).

Comme illustré, les scripts légers (Client, User Event, Suitelet) ne disposent que de 1 000 unités (Source: docs.oracle.com) (Source: docs.oracle.com), ce qui rend vital l'efficacité de chaque ligne de code. Les processus plus lourds (Scheduled, RESTlets) ont plus de marge (5–10k). Néanmoins, tout script à longue exécution divise souvent les tâches en Map/Reduce ou en plusieurs étapes pour gérer l'utilisation (par exemple, traiter un million d'enregistrements via 50 invocations de script planifié, chacune traitant 20 000 enregistrements).

Dans ces quotas, chaque appel d'API SuiteScript consomme des unités. Le Tableau 2 ci-dessous (et les sections qui suivent) détaille les coûts d'utilisation des méthodes pertinentes. L'aide officielle de NetSuite indique que les opérations complexes comme le chargement d'un enregistrement ou l'exécution d'une recherche sont délibérément plus coûteuses, reflétant la charge du serveur (Source: docs.oracle.com) (Source: www.houseblend.io). À l'inverse, les accesseurs simples en mémoire ne coûtent rien. Comprendre ces coûts est essentiel pour concevoir des scripts qui n'épuisent pas leurs quotas.

Par exemple, **E/S d'enregistrement vs recherche de champ** est un thème récurrent. La méthode `record.load` (chargement d'un enregistrement complet par ID) coûte jusqu'à 10 unités, tandis que `record.getValue` sur un enregistrement chargé est gratuit (Source: docs.oracle.com) (Source: www.houseblend.io). En revanche, `search.lookupFields` (récupération de champs spécifiques pour un ID d'enregistrement) ne coûte qu'une unité (Source: docs.oracle.com) (Source: www.houseblend.io). Ces différences ont des conséquences pratiques sur la conception et les performances des scripts, comme nous l'explorons ci-dessous.

Modèle de gouvernance de SuiteScript 2.x

Le modèle de gouvernance de NetSuite garantit une utilisation équitable des ressources. Tous les appels d'API SuiteScript sont pré-mesurés ; la dotation de l'environnement du script est réduite en conséquence. En cas de surconsommation, NetSuite arrête le script (Source: docs.oracle.com). Il est important de noter que la gouvernance s'applique également aux **résultats de recherche** : par exemple, par défaut, NetSuite limite une recherche renvoyée à 4 000 lignes (sauf si une recherche paginée est utilisée) (Source: docs.oracle.com), ce qui affecte les scripts traitant de grands ensembles de données.

Utilisation des appels d'API

La documentation officielle de l'API SuiteScript 2.x fournit un **tableau de gouvernance** complet. Par exemple, les modules *N/record* et *N/search* listent les pénalités d'unités pour chaque appel (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: www.houseblend.io). Quelques entrées clés pertinentes pour cette discussion sont :

- **record.load(options)** – Charge un enregistrement par ID. Coût de gouvernance : **10 unités** pour les enregistrements de transaction standard, **2 unités** pour les enregistrements personnalisés et **5 unités** pour les autres enregistrements (Source: docs.oracle.com) (Source: www.houseblend.io).
- **record.save(options)** – Enregistre les modifications apportées à un enregistrement. Coût : **20** (transaction), **4** (personnalisé), **10** (autre) (Source: docs.oracle.com) (Source: www.houseblend.io).
- **record.submitFields(options)** – Met à jour les champs spécifiés sans charger l'enregistrement complet. Coût : **10** (transaction), **2** (personnalisé), **5** (autre) (Source: docs.oracle.com) (Source: www.houseblend.io).
- **search.lookupFields(options)** – Renvoie un ou plusieurs champs pour un ID d'enregistrement donné. Coût : **1 unité** (fixe) (Source: docs.oracle.com) (Source: www.houseblend.io).
- **search.load(options)** et **search.run(options)** – La création ou le chargement d'une recherche enregistrée coûte 5 unités (Source: docs.oracle.com), et l'exécution d'une recherche coûte actuellement 5 unités par exécution (Source: www.houseblend.io) (confirmé par plusieurs sources).
- **ResultSet.each(callback)** – L'itération sur les résultats de recherche coûte **10** unités par appel (Source: docs.oracle.com) (ce qui implique que chaque tranche de 1 000 lignes peut coûter 10 unités).
- **record.getValue(options)** / **setValue** – Les getters/setters de champs en mémoire ont un coût de **0** (Source: docs.oracle.com) (Source: www.houseblend.io).
- **email.send(options)** – L'envoi d'un e-mail coûte **20** unités (Source: www.houseblend.io).
- **file.load(options)** – Le chargement d'un fichier depuis le File Cabinet coûte **10** unités (Source: www.houseblend.io).

Le tableau 2 résume l'utilisation pour une sélection d'appels :

Tableau 2. Utilisation des méthodes SuiteScript (SuiteScript 2.x)

MÉTHODE (SUITESCRIPT 2.X)	UNITÉS D'UTILISATION (TRANSACTION)	PERSONNALISÉ	AUTRES ENREGISTREMENTS	DESCRIPTION
`record.load(options)`	10	2	5	Charge un enregistrement par ID (données complètes et sous-listes) (Source: docs.oracle.com) (Source: www.houseblend.io).
`record.save(options)`	20	4	10	Enregistre les modifications d'un enregistrement (sauvegarde complète) (Source: docs.oracle.com) (Source: www.houseblend.io).
`record.submitFields(options)`	10	2	5	Met à jour des champs spécifiques sans charger tout l'enregistrement (Source: docs.oracle.com) (Source: www.houseblend.io).
`search.lookupFields(options)`	**1**	1	1	Récupère les champs sélectionnés pour un ID d'enregistrement (Source: docs.oracle.com) (Source: www.houseblend.io).
`search.load(options)`	5	5	5	Charge une définition de recherche enregistrée (Source: docs.oracle.com).
`search.run(options)`	~5	~5	~5	Exécute une recherche enregistrée ou ad-hoc (renvoie un jeu de résultats) (Source: www.houseblend.io).
`ResultSet.getRange(options)`	10 (par appel)	–	–	Récupère une plage de résultats de recherche (Source: docs.oracle.com).
`record.getValue(options)`	0 (aucun)	0	0	Lit une valeur depuis un enregistrement chargé (aucun coût) (Source: docs.oracle.com).
`search.lookupFields.promise()`	1	1	1	Version asynchrone de lookupFields (même coût).

MÉTHODE (SUITESCRIPT 2.X)	UNITÉS D'UTILISATION (TRANSACTION)	PERSONNALISÉ	AUTRES ENREGISTREMENTS	DESCRIPTION
Autres méthodes de sous-liste/enregistrement	*Variable (souvent aucun)*			De nombreux getters simples (`getValue`, `getText`) ont un coût de 0 (Source: docs.oracle.com).

Ces chiffres illustrent l'importance de la charge liée au chargement des enregistrements par rapport aux simples recherches. Par exemple, charger un seul enregistrement de commande client (une transaction) coûte 10 unités, alors qu'appeler `search.lookupFields` pour les champs de cet enregistrement ne coûte que 1 unité. De même, enregistrer cette commande (via `record.save`) coûte 20 unités. Au total, un aller-retour chargement-sauvegarde sur une transaction coûte 30 unités (Source: www.houseblend.io).

Surveillance et bonnes pratiques

Les développeurs peuvent suivre l'utilisation via `runtime.getCurrentScript().getRemainingUsage()` (SuiteScript 2.x) et concevoir des scripts pour gérer correctement les conditions proches de la limite (Source: docs.oracle.com). Les conseils officiels mettent l'accent sur la surveillance et les points de rendement : par exemple, les scripts Map/Reduce peuvent « céder » (yield) lors du prétraitement pour réinitialiser les gouverneurs. Comme le note [Huỳnh T. Đạt], « Évitez de charger un enregistrement si vous n'en avez pas besoin », car cela consomme entre 2 et 10 unités plus les frais généraux (Source: hutada.home.blog). En effet, l'analyse récente de Houseblend souligne le principe : « *Minimisez les opérations coûteuses : utilisez l'API la plus légère possible. Par exemple, au lieu de `record.load`, utilisez `search.lookupFields` ou `search.run` lors de la récupération de quelques champs* » (Source: www.houseblend.io).

`record.load` dans SuiteScript 2.x

La fonction `record.load(options)` du module `N/record` est l'un des appels SuiteScript les plus fréquemment utilisés. Elle instancie un objet d'enregistrement complet à partir de son ID interne et de son type. Par exemple :

```
const record = require('N/record');
const so = record.load({
    type: record.Type.SALES_ORDER,
    id: 1234
});
```

Cela charge tous les champs d'en-tête, les sous-listes, les sous-enregistrements et les métadonnées de l'enregistrement. Il est courant de faire suivre cela par des appels comme `so.getValue({ fieldId: 'entity' })` pour inspecter les valeurs. Comme elle récupère l'intégralité de l'enregistrement depuis la base de données, `record.load` est relativement coûteuse en unités d'utilisation et en latence (Source: hutada.home.blog) (Source: www.anchorgroup.tech).

Comme documenté dans les tableaux de gouvernance, le coût de `record.load` dépend de la catégorie de l'enregistrement (Source: docs.oracle.com). L'aide SuiteScript précise explicitement :

> ***record.load(options)*** *– Utilise **10 unités d'utilisation** pour les enregistrements de transaction, **2 unités** pour les enregistrements personnalisés et **5 unités** pour les enregistrements standard non transactionnels (Source: docs.oracle.com) (Source: www.houseblend.io). Ce coût échelonné reflète la complexité du traitement. Les enregistrements de transaction (commandes client, factures, etc.) contiennent généralement plus de données (sous-listes, lignes) et coûtent donc 10 unités. Les entités standard (clients, articles) coûtent 5, tandis que les enregistrements personnalisés plus simples ne coûtent que 2.*

Au-delà de la gouvernance documentée, `record.load` comporte des considérations pratiques :

- **Surcharge de performance :** Le chargement d'un enregistrement récupère également les *métadonnées* de l'enregistrement (tous les champs et sous-listes), ce qui peut introduire de la latence. Comme le souligne un développeur, cela « charge les métadonnées de l'enregistrement, ce qui

rend le processus plus long » (Source: hutada.home.blog). Sur une transaction complexe, cela peut prendre beaucoup plus de temps qu'une simple recherche.

- **Considérations sur la mémoire :** Après le chargement, l'accès aux valeurs (via `getValue` ou `getText`) se fait en mémoire et est gratuit (0 unité) (Source: docs.oracle.com). Cependant, l'itération sur les sous-listes consomme toujours du temps d'exécution de script.
- **Utilisation pour l'écriture :** Pour les scripts qui modifient un enregistrement, le modèle `record.load` suivi de `save` est courant. Ce cycle « charger puis sauvegarder » écrit tous les champs et déclenche la logique métier. Cependant, en guise de bonne pratique, lorsque seuls quelques champs doivent être mis à jour, `record.submitFields` est préférable (abordé plus loin). La différence est frappante : un chargement + sauvegarde (par exemple, sur une transaction) coûte environ 30 unités (Source: www.houseblend.io), alors qu'un seul appel `submitFields` coûte au maximum 10.

Illustrons comment une utilisation intensive de `record.load` peut consommer les quotas. Anchor Group fournit un exemple : si l'on doit lire des champs à partir de plusieurs enregistrements d'articles, une boucle avec `record.load` pourrait ressembler à ceci :

```
for (let id of itemIds) {
    let itemRec = record.load({
        type: record.Type.INVENTORY_ITEM,
        id: id
    });
    let sku = itemRec.getValue({ fieldId: 'sku' });
    // process sku...
}
```

Puisque `Inventory Item` est un enregistrement standard non transactionnel, chaque `load` coûte **5 unités**. Si `itemIds` contient 200 entrées, le coût total est de 1 000 unités – exactement la limite d'un User Event (Source: www.anchorgroup.tech). Au-delà, le script échouera à moins d'être réarchitecturé (par exemple, déplacé vers un Map/Reduce ou divisé en lots). Comme le note Anchor Group :

> *« Chaque utilisation de `record.load` coûte 5 unités pour les enregistrements d'articles. S'il y a jusqu'à 200 ID d'articles... cela fonctionne bien dans un script User Event. S'il y a plus de 200 ID d'articles... ce script devra peut-être être déchargé vers un script Map/Reduce » (Source: www.anchorgroup.tech).*

Dans le tableau 2, nous avons résumé les unités d'utilisation pour les méthodes d'enregistrement clés. Nous constatons que `record.save` est encore plus coûteux : sauvegarder une transaction coûte 20 unités (Source: docs.oracle.com) (Source: www.houseblend.io). Un exemple de Houseblend a montré un script utilisant `record.load` + `record.save` par enregistrement (30 unités au total) et atteignant rapidement la limite (Source: www.houseblend.io).

Optimisation autour de `record.load`

Compte tenu de son coût, les développeurs recherchent souvent des alternatives aux `record.load` répétés. Certaines tactiques incluent :

- **Récupérer uniquement les champs nécessaires :** Si seuls quelques champs sont requis, charger l'enregistrement entier est un gaspillage d'unités. Cela motive l'utilisation de `search.lookupFields` (section suivante) ou de recherches enregistrées.
- **Utiliser `record.submitFields` pour les mises à jour :** Si vous écrivez dans un enregistrement, `submitFields` peut mettre à jour des champs sans chargement (5–10 unités) (Source: www.houseblend.io), évitant ainsi le coût total du chargement.
- **Mettre en cache les recherches :** Si vous référencez le même enregistrement plusieurs fois, mettez ses valeurs en cache dans une variable. Le recharger à nouveau entraînerait un coût en double.
- **Traitement par lots (Map/Reduce) :** Lorsque de nombreux enregistrements doivent être chargés, envisagez un script Map/Reduce pour diviser le traitement en tranches (chaque étape ayant des quotas distincts) (Source: www.houseblend.io) (Source: docs.oracle.com).

Comme le conseillent Houseblend et d'autres, le principe directeur est de « minimiser les opérations coûteuses » (Source: www.houseblend.io). En pratique, cela signifie souvent **éviter `record.load` lorsque vous pouvez utiliser un appel plus léger** tel que `search.lookupFields` ou `search.run`.

`search.lookupFields` dans SuiteScript 2.x

Le module `N/search` fournit des API liées à la recherche. Parmi celles-ci, `search.lookupFields(options)` est une méthode légère pour récupérer des champs spécifiques d'un seul enregistrement par son ID. Sa signature est :

```
const search = require('N/search');
let data = search.lookupFields({
    type: search.Type.CUSTOMER, // ou tout type d'enregistrement
    id: 1234,
    columns: ['entityid','email','phone'] // tableau d'ID de champs (ou champs joints)
});
```

Cela renvoie un objet contenant les valeurs de champ demandées (avec `value` et, le cas échéant, `text` pour les champs de sélection). Par exemple, la documentation officielle montre que lookupFields renvoie un JSON comme :

```
{
  internalid: 1234,
  firstname: 'Joe',
  my_multiselect: [
     {value: 1, text: 'US Sub'},
     {value: 2, text: 'Canada'}
  ]
}
```

où les champs multi-sélection renvoient un tableau de paires `{value,text}` (Source: docs.oracle.com).

La puissance de `lookupFields` réside dans le fait qu'**il ne coûte qu'une seule unité de gouvernance**, par appel, quel que soit le type d'enregistrement (Source: docs.oracle.com) (Source: www.houseblend.io). Dans le tableau de gouvernance, cela apparaît comme un « 1 » fixe sous le module N/search (Source: docs.oracle.com). L'analyse de Houseblend confirme :

> *« search.lookupFields(options) – 1 unité (récupère des champs spécifiques d'un enregistrement) » (Source: www.houseblend.io).*

Ce coût d'une unité rend les recherches de champs extrêmement efficaces en termes de gouvernance. Même si vous récupérez plusieurs champs à la fois, le coût reste de 1. En revanche, l'exécution d'une recherche enregistrée ou d'une recherche complète (`search.run`) coûte généralement 5 unités à chaque fois. Et `record.load` peut coûter 10.

Caractéristiques clés de `search.lookupFields` :

- **Récupération de données ciblée :** Il récupère uniquement les *champs d'en-tête (body fields)* de l'enregistrement (Source: docs.oracle.com). Il peut également récupérer des champs joints à partir d'enregistrements associés en utilisant la « syntaxe de jointure » – par exemple, `columns: ['entityid','created_from.orderstatus']`.
- **Focus sur un seul enregistrement :** Il fonctionne sur un enregistrement spécifique par ID. Il ne peut pas rechercher ou filtrer par critères ; pour plusieurs enregistrements, on utiliserait `search.run` ou une requête de recherche.
- **Renvoie des valeurs et du texte :** Pour les champs de sélection simple, le résultat produit un objet avec `{ value: xxx, text: 'ABCD' }`. Pour les champs de sélection multiple, comme indiqué ci-dessus, un tableau de paires `{value,text}` est renvoyé (Source: docs.oracle.com).
- **Appel léger :** Il effectue une recherche rapide dans la base de données. Comme il ne renvoie que les champs spécifiés, la charge utile de données est faible et l'appel est efficace. La documentation note : « La méthode search.lookupFields(options) inclut également une version promise... Renvoie les champs de sélection sous forme d'objet avec des propriétés de valeur et de texte » (Source: docs.oracle.com).
- **Exemple d'utilisation :** Un blog de développeur souligne la simplicité :

  > *« Si vous connaissez l'ID de l'enregistrement que vous ciblez, utilisez `search.lookupFields`. Cela coûte 1 unité de gouvernance... et c'est plus rapide » (Source: hutada.home.blog).*

En raison de ces attributs, `search.lookupFields` est fortement recommandé lorsque seules quelques valeurs de champ provenant d'un ID d'enregistrement connu sont nécessaires. Par exemple, si seuls le nom de l'entité et le statut d'une commande client sont requis, une recherche est bien plus efficace que le chargement de la commande entière. De nombreux experts citent ce modèle. Comme le déclare Houseblend : *« Utiliser `search.lookupFields` (1 unité) au lieu de charger un enregistrement entier (10 unités) chaque fois que seuls quelques champs sont nécessaires entraîne de grosses économies »* (Source: www.houseblend.io). En pratique, cela signifie que vous pourriez remplacer :

```
let rec = record.load({ type: record.Type.SALES_ORDER, id: soId });
let status = rec.getText({ fieldId: 'orderstatus' });
```

par :

```
let data = search.lookupFields({
    type: search.Type.SALES_ORDER,
    id: soId,
    columns: ['orderstatus']
});
let status = data.orderstatus.text;
```

économisant 9 unités d'utilisation dans cet exemple simple.

Cependant, `search.lookupFields` a des limites :

- **Pas d'accès aux sous-listes :** Il ne peut pas récupérer de données au niveau des lignes ou des sous-listes. Si vous avez besoin de valeurs provenant de sous-enregistrements (comme des articles sur une transaction), `record.load` ou un `search.run` avec des jointures de sous-listes doit être utilisé.
- **Champs uniquement, aucun filtre :** Vous devez connaître l'identifiant interne de l'enregistrement à l'avance ; cette méthode ne permet pas d'effectuer des recherches basées sur des critères.
- **Colonnes obligatoires :** Si vous omettez des colonnes ou spécifiez des champs inexistants, l'opération échouera.

Malgré ces limites, `lookupFields` est une bonne pratique pour réduire la consommation de gouvernance chaque fois que cela est possible. Cette méthode est fréquemment mentionnée dans les conseils de performance. Par exemple, le consultant NetSuite Huỳnh T. Đạt la recommande explicitement pour rechercher des données par ID : *« Si vous connaissez l'ID de l'enregistrement ciblé, utilisez `search.lookupFields`. Cela coûte 1 unité »* (Source: hutada.home.blog).

Utilisation de `submitFields` pour les mises à jour

Bien que ce ne soit pas l'objectif principal de ce rapport, il convient de noter que NetSuite propose une autre méthode légère pour les mises à jour d'enregistrements : `record.submitFields(options)`. Elle permet de mettre à jour un ou plusieurs champs sur un enregistrement sans avoir à le charger. Les coûts de gouvernance pour `submitFields` sont identiques à ceux de `record.load` : 10/2/5 pour les transactions/enregistrements personnalisés/autres (Source: docs.oracle.com) (Source: www.houseblend.io). Étant donné que `submitFields` ne charge pas toutes les sous-listes, il peut être plus rapide et moins coûteux qu'un cycle chargement+sauvegarde (qui coûte 10+20=30 unités sur une transaction, contre un maximum de 10 unités pour `submitFields`). Les experts en performance suggèrent régulièrement d'utiliser `submitFields` lorsque seuls quelques champs doivent être modifiés (Source: hutada.home.blog).

Performance et résultats des benchmarks

Bien que ce rapport mette l'accent sur les **unités** de gouvernance, le temps d'exécution réel (en millisecondes) est également important. En règle générale, une consommation d'unités moindre est souvent corrélée à une exécution plus rapide, mais ce n'est pas toujours le cas (la mise en cache et les effets réseau jouent un rôle). Une comparaison récente apporte un éclairage : NZR Solutions a testé la récupération d'un champ (ID du client parent) dans un script User Event en utilisant soit SuiteQL, soit `search.lookupFields` (Source: www.nzrsolutions.com). Les résultats (moyennés sur de nombreuses exécutions) étaient :

- **SuiteQL** : ~0,0341 secondes

- **search.lookupFields** : ~0,0316 secondes

Lors des exécutions individuelles, `lookupFields` s'est avéré nettement plus rapide (0,027s contre 1,007s pour SuiteQL lors de la première tentative) en raison des effets de mise en cache (Source: www.nzrsolutions.com) (Source: www.nzrsolutions.com). Dans l'ensemble, les deux approches avaient une vitesse similaire, mais la conclusion clé est que `lookupFields` était **rapide et peu gourmand en ressources**. (Il est important de noter que `lookupFields` n'a consommé qu'une seule unité par appel, alors que la construction et l'exécution d'une requête SuiteQL consomment également des unités, bien que non pas directement liées aux E/S d'enregistrement, mais au moteur de requête.)

Les observations des praticiens en matière de performance font écho aux conseils sur la gouvernance. Comme l'écrit The NetSuite Pro : *« Minimisez les chargements d'enregistrements : évitez de charger des enregistrements entiers lorsqu'un seul champ est nécessaire. Utilisez `lookupFields` ou `search.lookupFields` »* (Source: www.thenetsuitepro.com). En pratique, l'élimination de dizaines de chargements d'enregistrements a permis d'obtenir des scripts à la fois plus rapides et respectant la limite d'utilisation dans de nombreuses études de cas.

Cependant, la vitesse brute n'est pas le seul facteur. L'environnement d'exécution de NetSuite peut mettre en cache les recherches (comme le suggère l'exemple de préchauffage SuiteQL). Néanmoins, l'utilisation d'une recherche à appel unique par enregistrement est souvent plus évolutive que le chargement de dizaines d'enregistrements. Il faut également tenir compte de la concurrence : un script effectuant des recherches en parallèle paiera toujours 1 unité par enregistrement, alors qu'une recherche en masse peut amortir le coût. La décision dépend souvent du fait que vous disposiez des identifiants à l'avance (privilégiez la recherche par ID) ou que vous deviez découvrir des enregistrements via des critères (privilégiez la recherche).

Études de cas et exemples

Pour illustrer ces principes, nous examinons des exemples réels où des optimisations tenant compte de la gouvernance ont fait la différence.

Suitelet d'importation en masse (Dépassement d'unités)

Un article de Houseblend décrit une entreprise important des milliers de lignes de transaction via un Suitelet. Leur approche initiale était naïve : pour chaque ligne, ils effectuaient un `record.load` et un `record.save`. À 30 unités par itération, ils ont rapidement dépassé la limite et rencontré des erreurs (Source: www.houseblend.io). L'analyse a révélé :

- **Coût par ligne :** ~30 unités (`load` + `save`) (Source: www.houseblend.io).
- **Solution :** Passer à un script planifié (10 000 unités par exécution) et traiter des **lots de 200 enregistrements** à la fois.
- **Amélioration supplémentaire :** Au lieu de charger à plusieurs reprises les enregistrements de référence, ils ont utilisé `search.lookupFields` pour récupérer les données statiques une seule fois par ligne. Cela a permis d'économiser « des centaines d'unités par lot » (Source: www.houseblend.io).

Ce cas souligne l'importance du traitement par lots et des recherches. Découper une tâche en segments a permis de rester dans la limite des 10 000 unités. L'utilisation de `lookupFields` pour récupérer des données auxiliaires (au lieu de chargements d'enregistrements supplémentaires) a considérablement réduit le coût par élément.

Recherche enregistrée dans un User Event

Un autre exemple impliquait un script User Event se déclenchant à chaque sauvegarde de commande client. Il effectuait une recherche enregistrée coûteuse avec de nombreux résultats à chaque exécution. Cela a entraîné une utilisation anormalement élevée et une lenteur des performances. La solution a consisté à **mettre en cache** la recherche enregistrée (afin qu'elle ne soit pas réexécutée à chaque fois) et à **limiter les colonnes** aux seuls champs indexés. Après ces changements, l'utilisation a chuté de manière significative (Source: www.houseblend.io). Bien que cela ne concerne pas directement `record.load` ou `lookupFields`, cela démontre un thème commun : éviter le travail de recherche inutile. Le filtrage par champs indexés et la réutilisation des définitions de recherche ont réduit l'utilisation « de façon spectaculaire » (Source: www.houseblend.io). L'approche est parallèle aux conseils sur `lookupFields` : ne récupérez que ce dont vous avez besoin.

Boucle de chargements d'enregistrements d'articles (Anchor Group)

Considérons à nouveau l'exemple des articles d'inventaire d'Anchor Group (Source: www.anchorgroup.tech). Ils ont noté deux approches :

- **Boucle avec `record.load` :** 200 articles × 5 unités chacun = 1 000 unités.
- **Recherche unique :** Une recherche enregistrée ou ad hoc sur ces 200 identifiants avec uniquement les colonnes nécessaires coûte 10 unités au total. Cela illustre qu'une recherche peut être beaucoup moins coûteuse lorsqu'on traite de nombreux enregistrements. En effet, Anchor Group a écrit :

> *« Si aucune valeur ne doit être définie et que seuls les champs de corps sont nécessaires, le montant de la gouvernance peut être fixé à 10 unités, quel que soit le nombre d'identifiants d'articles nécessaires » (Source: www.anchorgroup.tech).*

En somme, les expériences des développeurs et la documentation officielle pointent vers la même bonne pratique : **Lorsque vous travaillez avec des identifiants d'enregistrement connus et que vous n'avez besoin que de quelques champs, utilisez `search.lookupFields` (1 unité). Lorsque vous travaillez avec de nombreux enregistrements, utilisez `N/search` pour les récupérer par lot (par exemple, un `search.run` à ~5 unités chacun)** (Source: hutada.home.blog) (Source: www.anchorgroup.tech). À l'inverse, `record.load` doit être réservé aux situations où les détails complets de l'enregistrement sont nécessaires ou lorsque des mises à jour des sous-listes sont requises.

Implications et orientations futures

L'analyse approfondie des unités de gouvernance qui précède a des implications pratiques :

- **Concevoir tôt pour l'échelle :** Les développeurs doivent estimer le nombre d'enregistrements attendus. Une boucle de 100 articles chargés un par un (500 unités) est acceptable pour un script de 1 000 unités, mais 1 000 articles (5 000 unités) ne le sont pas. Si l'utilisation approche du quota, envisagez Map/Reduce ou une décomposition planifiée (Source: www.houseblend.io).
- **Privilégier les API légères :** En guise de bonne pratique, demandez-vous toujours : « Puis-je utiliser un appel moins coûteux ? » La mise en œuvre de stratégies telles que `lookupFields`, `search.run`, les recherches enregistrées et `submitFields` plutôt que `record.load` portera ses fruits (Source: www.houseblend.io) (Source: hutada.home.blog).
- **Surveiller et tester :** Utilisez `getRemainingUsage()` pour vérifier la consommation pendant le développement. Les scripts dans la Sandbox doivent être testés avec des volumes de données représentatifs de la production pour détecter les cas limites dès le début.
- **Surveiller les nouvelles API :** NetSuite ajoute régulièrement des fonctionnalités. Par exemple, SuiteQL (requêtes de type SQL) et le module N/query (SuiteScript 2.1) offrent des capacités de requête SQL. Les premières preuves suggèrent qu'ils peuvent parfois rivaliser avec `lookupFields` en termes de vitesse, mais leur empreinte sur la gouvernance doit être comprise. Le test de NZR a révélé un temps moyen pour SuiteQL d'environ 0,034s contre 0,032s pour lookupFields (Source: www.nzrsolutions.com), mais n'a pas discuté spécifiquement des unités d'utilisation (SuiteQL entraîne probablement une consommation similaire à une recherche). À mesure que SuiteQL mûrit, il pourrait offrir une alternative, surtout pour les requêtes complexes. Pourtant, même avec de nouveaux outils, la leçon fondamentale demeure : ne demandez que ce dont vous avez besoin.

À long terme, le modèle de NetSuite continue de récompenser le codage efficace. Les intégrations à haut volume (comme les plateformes d'intégration ou les outils de synchronisation de données) doivent respecter la gouvernance ou utiliser les services Web REST avec des quotas distincts. Les administrateurs et les architectes doivent sensibiliser les équipes à ces limites.

Pour le futur développement SuiteScript, la tendance est claire : *plus léger, c'est plus rapide*. Les innovations à venir pourraient inclure des capacités de recherche plus granulaires (par exemple, le renvoi d'objets JSON) ou une mise en cache intégrée. En attendant, NetSuite a signalé qu'une automatisation excessive peut déclencher une mesure de la consommation en vertu de leurs conditions d'utilisation (Source: docs.oracle.com), rappelant aux utilisateurs de rester dans les modèles d'utilisation normatifs.

Conclusion

La comparaison axée sur la gouvernance entre `record.load` et `search.lookupFields` de SuiteScript révèle un fait crucial pour les développeurs : **choisissez le bon outil pour le bon travail**. Toutes choses égales par ailleurs, le chargement d'un enregistrement complet est beaucoup plus coûteux que la recherche de quelques champs. Nous avons montré, grâce à la documentation, aux conseils d'experts et à des scénarios réels, que le recours à `search.lookupFields` et à d'autres méthodes à faible coût peut réduire considérablement la consommation d'unités d'utilisation et améliorer les performances (Source: hutada.home.blog) (Source: www.houseblend.io). À l'inverse, les chargements et sélections d'enregistrements inutiles sont un piège courant menant aux erreurs « SSS_USAGE_LIMIT_EXCEEDED ».

Les tableaux 1 et 2 résument les informations quantitatives clés : les opérations de chargement/sauvegarde d'enregistrements entraînent jusqu'à 10–20 unités par appel (Source: docs.oracle.com) (Source: www.houseblend.io), tandis que `lookupFields` est à un tarif fixe de 1 unité (Source: docs.oracle.com) (Source: www.houseblend.io). La connaissance du schéma est importante : les développeurs doivent distinguer les transactions, les

enregistrements personnalisés et les non-transactions lors de la planification (car les coûts varient).

Notre analyse souligne la conception de scripts basée sur des preuves. Par exemple, une conversion Map/Reduce d'une importation en masse qui a remplacé les chargements par des recherches a permis d'éviter les dépassements de limites (Source: www.houseblend.io). De même, un script de traitement d'articles a réduit son utilisation en utilisant une recherche unique au lieu de 200 chargements individuels (Source: www.anchorgroup.tech) (Source: www.anchorgroup.tech). Ces exemples servent d'études de cas de bonnes pratiques pour d'autres.

À l'avenir, à mesure que NetSuite évoluera, des alternatives comme SuiteQL changeront le paysage, mais les unités de gouvernance resteront centrales. Ainsi, la connaissance du coût d'utilisation de chaque API est indispensable. Nous concluons qu'en internalisant ces coûts et en suivant le principe « minimiser les opérations coûteuses » (Source: www.houseblend.io), les développeurs peuvent garantir des applications SuiteScript robustes et évolutives qui restent bien dans les limites de gouvernance, aujourd'hui et à l'avenir.

Références : Toutes les unités de gouvernance et tous les comportements sont tirés de la documentation NetSuite la plus récente (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: www.houseblend.io) ainsi que d'analyses d'experts et d'études de cas (Source: hutada.home.blog) (Source: www.houseblend.io) (Source: www.houseblend.io) (Source: www.houseblend.io). (Les citations en ligne ci-dessus indiquent les sources spécifiques et les numéros de ligne.)

Étiquettes: suitescript-2x, gouvernance-netsuite, recordload, searchlookupfields, limites-api, optimisation-performance, developpement-suitescript, unites-usage

---