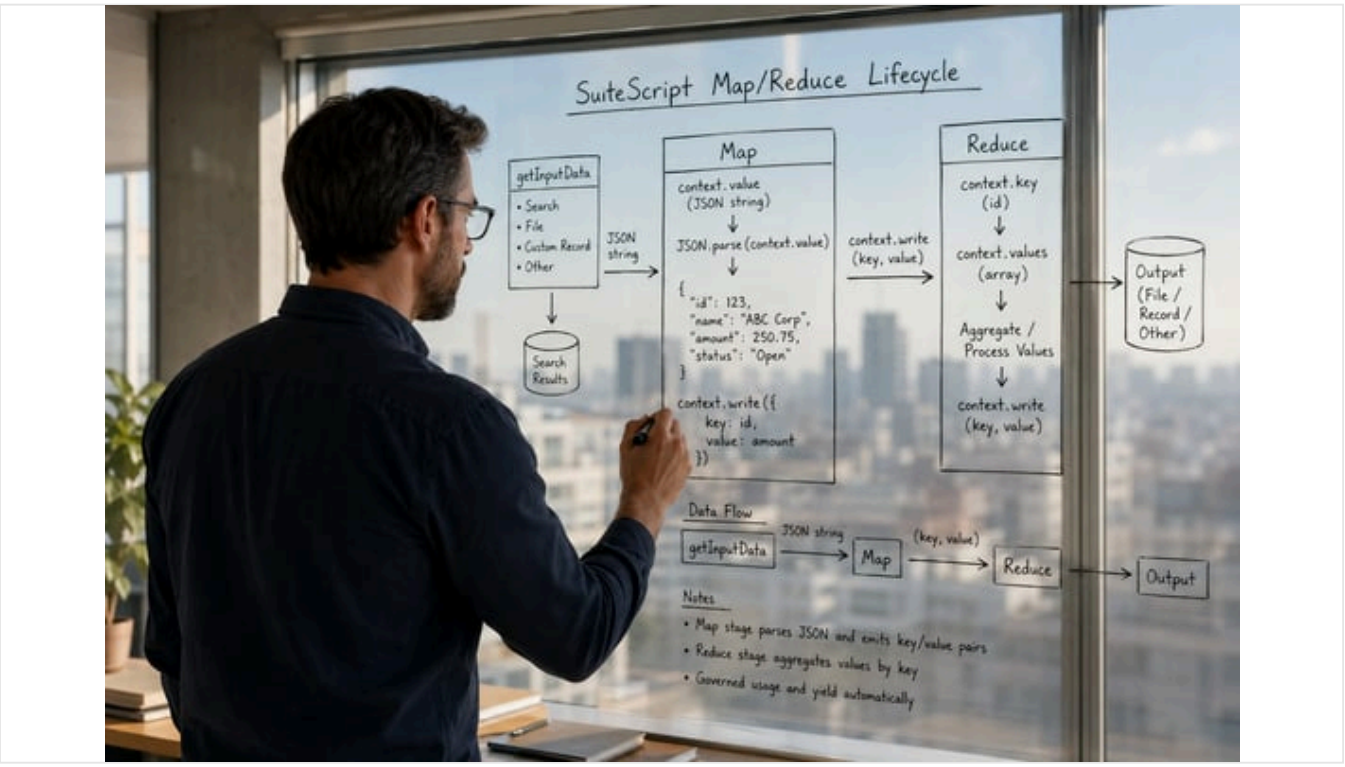
Guide d'analyse JSON de context.value dans SuiteScript Map/Reduce

Publié le 25 avril 2026 31 min de lecture



Résumé analytique

SuiteScript Map/Reduce est un framework puissant introduit par NetSuite pour permettre le traitement de données à haut volume au sein de l'environnement SuiteCloud. Ce rapport fournit une **analyse extrêmement approfondie** de la propriété `context.value` au sein du script Map/Reduce, en se concentrant spécifiquement sur son rôle en tant que **chaîne JSON de données de résultats de recherche**, et sur les divers *modèles d'analyse* (parsing) que les développeurs utilisent pour extraire des informations de ce JSON. Nous détaillons le **contexte historique** de SuiteScript et de Map/Reduce, l'**architecture et le comportement actuels** des scripts Map/Reduce, la *structure exacte* du JSON produit lorsque les résultats de recherche sont transmis via `context.value`, ainsi que les **meilleures pratiques** pour analyser ce JSON. Nous nous appuyons sur la documentation officielle de NetSuite, les connaissances de la communauté des développeurs, des articles de blog d'experts et des statistiques du secteur pour fournir une **discussion approfondie et fondée sur des preuves**.

Les principales conclusions incluent :

- **Paradigme Map/Reduce dans NetSuite :** Le type de script Map/Reduce de NetSuite s'inspire du concept MapReduce de Google (Source: docs.oracle.com). Il divise les données en paires clé-valeur pour un traitement parallèle. Dans SuiteScript Map/Reduce, si une recherche est renvoyée par l'étape `getInputData`, chaque résultat de recherche génère une paire clé-valeur : la *clé* est l'ID interne de l'enregistrement et la *valeur* est une chaîne JSON des données de l'enregistrement (Source: docs.oracle.com).
- **Structure JSON des résultats de recherche :** Comme documenté, `context.value` contient une chaîne JSON qui est essentiellement l'objet `search.Result` sérialisé par `JSON.stringify()` (Source: docs.oracle.com) (Source: suitescriptwithramesh.blogspot.com). Le JSON inclut des propriétés telles que `"recordType"` (le type d'enregistrement, par ex. `"customer"` (Source: docs.oracle.com), `"id"` (l'ID interne sous forme de chaîne (Source: docs.oracle.com), et `"values"` (un objet contenant toutes les colonnes de recherche demandées et leurs valeurs). Les clés de l'objet `"values"` correspondent aux ID de colonne (et aux noms de jointure, le cas échéant). Les valeurs des champs simples (dates, nombres, texte) sont données sous forme de chaînes brutes, tandis que les champs de liste/enregistrement apparaissent sous forme de sous-

objets avec les propriétés `"value"` et `"text"` (Source: stackoverflow.com) (Source: cloud.tencent.com). Par exemple, un champ joint tel que `"item.workOrder"` apparaît sous la forme `{"value":"1517","text":"Some Item Name"}` dans le JSON (Source: stackoverflow.com).

- **Modèles d'analyse (Parsing) :** Pour accéder à ces données, les développeurs appellent généralement `JSON.parse(context.value)` dans la fonction `map` ou `reduce` pour convertir la chaîne JSON en objet JavaScript (Source: houseblend.io) (Source: cloud.tencent.com). Ils lisent ensuite les champs via `result.id`, `result.values.fieldname`, ou en utilisant la notation entre crochets pour les clés avec des points (par ex. `data["item.workOrder"].value`) (Source: stackoverflow.com) (Source: cloud.tencent.com). La meilleure pratique consiste souvent à utiliser l'API `search.Result` (`getValue`, avec jointure optionnelle) lorsque cela est possible (pour éviter de manipuler le JSON), mais l'analyse du JSON est simple et courante dans les scripts personnalisés (Source: stackoverflow.com) (Source: suitescriptwithramesh.blogspot.com).
- **Comportement systémique et limites :** Map/Reduce sérialise intrinsèquement les données entre les étapes, de sorte que les clés et les valeurs sont transmises sous forme de chaînes JSON (Source: houseblend.io). NetSuite impose des limites de taille : les clés de plus de ~3000 caractères ou les valeurs de plus de 10 Mo (historiquement 1 Mo) échoueront (Source: houseblend.io) (Source: docs.oracle.com). Comprendre ces limites est crucial dans la conception des données.
- **Implications sur les performances :** Map/Reduce améliore considérablement le débit pour les grands ensembles de résultats de recherche. Par exemple, diviser une mise à jour de 10 000 enregistrements en cinq tâches de mappage parallèles peut se terminer *beaucoup plus rapidement* qu'un script planifié monothread (Source: houseblend.io). Map/Reduce cède également automatiquement la main pour éviter les limites de gouvernance, permettant un traitement total « illimité » sur de nombreuses invocations (Source: houseblend.io).
- **Orientations futures :** À mesure que NetSuite évolue (par ex. requêtes SuiteQL pour les données structurées), les développeurs pourraient utiliser de plus en plus SuiteQL dans les étapes `getInputData` ou `map` pour contourner l'analyse JSON manuelle (Source: timdietrich.me). L'adoption généralisée du JSON dans les API modernes (environ 97 % pour les API REST (Source: www.glyphwidgets.com) signifie que le format JSON de `context.value` s'aligne bien avec les tendances plus larges de l'industrie vers des intégrations centrées sur le JSON. Le modèle Map/Reduce et son flux de données JSON resteront probablement au cœur des tâches d'intégration et de personnalisation SuiteCloud à haut volume.

En résumé, ce rapport synthétise la documentation officielle et l'expertise de la communauté pour fournir une **compréhension complète** de la structure JSON de `context.value` dans Map/Reduce et de la manière de travailler avec celle-ci. Nous proposons des exemples de modèles, des mises en garde et des meilleures pratiques, étayés par des références tout au long du document.

Introduction

NetSuite SuiteScript est une API basée sur JavaScript qui permet aux développeurs de créer une logique personnalisée sur la plateforme NetSuite (un système ERP cloud intégré). Initialement introduit sous le nom de SuiteScript 1.0, l'API SuiteScript a été remaniée avec SuiteScript 2.x pour offrir une expérience de script moderne et modulaire. Une amélioration majeure de SuiteScript 2.x (vers 2016) a été le type de script **Map/Reduce**. Inspirés par le paradigme MapReduce dans l'informatique distribuée (Source: docs.oracle.com), les scripts Map/Reduce de NetSuite permettent aux développeurs de traiter des **opérations de données en masse** en parallèle sur plusieurs threads de traitement (appelés processeurs SuiteCloud) (Source: houseblend.io) (Source: houseblend.io). Ceci est particulièrement utile pour des tâches telles que les mises à jour massives d'enregistrements, les migrations de données et le traitement complexe de transactions, où des dizaines de milliers d'enregistrements peuvent être impliqués.

Traditionnellement, les développeurs SuiteScript utilisaient des scripts planifiés ou des Suitelets pour le travail par lots, parcourant manuellement les résultats de recherche et se souciant des limites de gouvernance (NetSuite impose des plafonds d'utilisation de l'exécution). En revanche, les **scripts Map/Reduce divisent automatiquement la charge de travail** en une série de tâches et gèrent en interne les problèmes tels que la gouvernance (cédant et replanifiant) (Source: houseblend.io). Cela en fait l'approche recommandée pour les processus personnalisés à grande échelle dans le développement NetSuite moderne.

Un aspect clé de l'écriture de scripts Map/Reduce est la compréhension du **flux de données entre les étapes**. Un script Map/Reduce comporte jusqu'à cinq étapes : `getInputData`, `map`, `shuffle` (interne, pas de code personnalisé), `reduce` et `summarize` (Source: docs.oracle.com). Dans l'étape **getInputData**, le développeur renvoie l'ensemble de données à traiter. Il peut s'agir d'un tableau, d'un objet ou (important) d'une recherche enregistrée ou d'un objet de recherche. Si une recherche est renvoyée (par exemple, `return search.create({...});` ou une recherche chargée), NetSuite exécutera cette recherche et produira les enregistrements d'entrée. Chaque enregistrement de la recherche devient une entrée de *paire clé-valeur* pour l'étape Map. Selon la documentation officielle, **la clé est généralement l'ID de l'enregistrement**, et **la valeur est une chaîne encodée en JSON des champs de l'enregistrement** (Source: docs.oracle.com) (Source: docs.oracle.com). Le mappage des résultats de recherche en paires clé/valeur est intégré au framework : « les paires clé-valeur seraient les résultats de la recherche où chaque clé serait l'ID interne d'un enregistrement

et chaque valeur serait une représentation JSON des ID de champ et des valeurs de l'enregistrement » (Source: docs.oracle.com). Ainsi, dans la fonction `map(context)` de l'étape Map, `context.key` sera l'ID de l'enregistrement (sous forme de chaîne) et `context.value` sera une chaîne JSON décrivant les données de cet enregistrement.

Ce rapport se concentre sur cette **chaîne JSON `context.value`** : sa **structure** exacte et les **modèles d'analyse** typiques utilisés pour extraire les données dans les scripts Map/Reduce. Nous couvrons :

- **Les objets de contexte Map/Reduce et le flux de données :** Comprendre comment les étapes `getInputData`, `map`, `reduce` et `summarize` utilisent `context.key` et `context.value`, et comment le système sérialise les données entre les étapes (Source: docs.oracle.com) (Source: suitescriptwithramesh.blogspot.com).
- **Format JSON des résultats de recherche :** Quels champs apparaissent dans le JSON (par ex. `recordType`, `id`, `values`), comment les champs de liste/enregistrement sont représentés (avec des sous-champs `value` et `text`) (Source: stackoverflow.com) (Source: cloud.tencent.com), et comment les colonnes jointes apparaissent.
- **Exemples de modèles :** Exemples d'extraits de code et approches d'analyse (utilisation de `JSON.parse(context.value)`, comment accéder aux champs de jointure, quand utiliser `getValue()` à la place) issus de questions/réponses de la communauté et d'exemples officiels (Source: stackoverflow.com) (Source: cloud.tencent.com).
- **Limites et meilleures pratiques :** Gestion des limites de taille de 1 Mo à 10 Mo sur les valeurs (Source: houseblend.io) (Source: docs.oracle.com), représentations numériques vs chaînes, et conseils pour les performances et la gouvernance (par ex. parallélisme, unités d'utilisation) (Source: houseblend.io) (Source: houseblend.io).
- **Implications futures :** Tendances émergentes telles que l'utilisation de SuiteQL pour une entrée structurée (réduisant les besoins d'analyse JSON (Source: timdietrich.me) et l'importance continue du JSON dans les API (Source: www.glyphwidgets.com).

Tout au long du document, nous étayons toutes les affirmations par des **sources canoniques** : documentation d'Oracle NetSuite (Source: docs.oracle.com) (Source: docs.oracle.com), blogs écrits par des experts (Source: houseblend.io) (Source: suitescriptwithramesh.blogspot.com), réponses de StackOverflow/communauté NetSuite (Source: stackoverflow.com) (Source: cloud.tencent.com), et analyses de données sur l'utilisation de JSON/API (Source: www.glyphwidgets.com) et l'adoption de NetSuite (Source: www.anchorgroup.tech). L'objectif est de fournir une ressource complète et détaillée pour les développeurs et les architectes travaillant avec SuiteScript Map/Reduce sur les résultats de recherche, en veillant à ce que chaque déclaration soit fondée sur des preuves et qu'aucun aspect significatif ne soit laissé inexploré.

SuiteScript Map/Reduce : Objets de contexte et flux de données

Avant d'aborder la structure JSON elle-même, nous passons en revue l'**architecture Map/Reduce** et le rôle des objets de contexte dans chaque étape. Cela fournit une base pour comprendre comment `context.value` est produit et consommé.

Étapes Map/Reduce et contexte

Un script SuiteScript Map/Reduce s'exécute par étapes discrètes, pour la plupart desquelles le développeur peut définir des fonctions de point d'entrée. Les étapes typiques sont : **getInputData**, **map**, **shuffle** (système uniquement), **reduce** et **summarize** (Source: docs.oracle.com) (Source: docs.oracle.com). Chaque étape est gérée par NetSuite et fonctionne sur les données en parallèle ou séquentiellement, selon le cas :

- **getInputData :** Cette étape exécute une fois le `getInputData(inputContext)` du développeur. Son travail est de *fournir* les données à traiter. La fonction renvoie généralement un tableau, un objet ou une recherche. Pour les grands ensembles de données, renvoyer une recherche enregistrée (par ex. `return search.load({ id: 'mysavedsearch' })`) est courant. NetSuite exécutera la recherche et utilisera ses résultats comme ensemble de données (Source: docs.oracle.com) (Source: docs.oracle.com). L'objet `inputContext` fournit des métadonnées (par ex. si le script est en cours de redémarrage) mais n'est généralement pas nécessaire pour récupérer les données elles-mêmes (Source: docs.oracle.com).

- **map :** S'il est fourni, `map(mapContext)` est appelé une fois pour **chaque** paire clé-valeur des données d'entrée. Dans notre scénario (où l'entrée est une recherche), chaque invocation gère **une ligne de résultat de recherche**. Ici, `mapContext.key` contient l'ID interne de l'enregistrement (sous forme de chaîne), et `mapContext.value` contient les données de l'enregistrement (sous forme de chaîne JSON) (Source: docs.oracle.com) (Source: suitescriptwithramesh.blogspot.com). L'étape Map peut traiter les enregistrements indépendamment en parallèle ; le code du développeur peut également émettre des paires clé-valeur supplémentaires via `mapContext.write({key, value})` pour alimenter l'étape reduce.

- **shuffle :** Cette étape est interne (pas de code utilisateur). NetSuite regroupe toutes les sorties de map par clé et les prépare pour reduce. Si aucune étape map n'a été utilisée (c'est-à-dire que `map` est omis et que seul `reduce` est défini), alors NetSuite mélange la sortie originale de getInputData en groupes comme si les clés étaient déjà assignées.
- **reduce :** S'il est fourni, `reduce(reduceContext)` est appelé **une fois par clé unique** après le mélange. Si chaque itération de map a écrit des paires `(key, value)`, alors dans `reduce`, chaque `reduceContext.key` est une clé unique et `reduceContext.values` est un tableau de toutes les valeurs associées à cette clé. (Si aucun map n'a été utilisé, alors chaque clé de ligne d'entrée originale sera regroupée avec des clés identiques.) Le code peut itérer sur `reduceContext.values` (généralement en appelant `JSON.parse` sur chacune), et appeler à nouveau `reduceContext.write()` pour produire des paires clé-valeur pour l'étape summarize.
- **summarize :** La phase finale `summarize(summaryContext)` s'exécute une fois après la fin de tous les travaux map/reduce. Elle a accès aux statistiques et peut récupérer l'ensemble de la sortie de la phase reduce via `summaryContext.output`. Notez qu'à la différence de `reduce`, le contexte de résumé ne possède **pas** de tableau `values` ; au lieu de cela, on itère sur `summaryContext.output.iterator()` pour voir tous les résultats finaux clé/valeur (Source: stackoverflow.com).

Le tableau suivant résume ces objets (paraphrasé de la documentation NetSuite) :

ÉTAPE	OBJET DE CONTEXTE	TYPE DE CLÉ	TYPE DE VALEUR	DESCRIPTION
getInputData	`inputContext` (obj)	N/A	recherche ou donnée	Fournit les données d'entrée (résultats de recherche, tableau, etc.)
map	`mapContext` (obj)	string (ID) (Source: docs.oracle.com)	string (JSON) (Source: docs.oracle.com) (Source: suitescriptwithramesh.blogspot.com)	Traite chaque ligne d'entrée. `mapContext.value` est une **chaîne JSON** des champs de l'enregistrement.
reduce	`reduceContext` (obj)	string	tableau de strings	Traite des groupes de valeurs par clé. `reduceContext.values` est un tableau de chaînes (souvent JSON) si map a écrit des valeurs.
summarize	`summaryContext` (obj)	N/A	N/A	Étape finale pour la journalisation/sortie. Utilisez `summaryContext.output` pour accéder aux paires clé/valeur finales.

Les comportements importants liés à `context.value` incluent :

- La **valeur de l'étape Map est toujours une chaîne**. Le système garantit que les clés et les valeurs transmises entre les étapes sont des chaînes pour éviter les références inter-contextes (Source: houseblend.io). En pratique, si l'entrée est une recherche, `mapContext.value` est une chaîne JSON représentant un objet `search.Result` (Source: docs.oracle.com) (Source: suitescriptwithramesh.blogspot.com).
- Chaque invocation de `map(context)` peut s'exécuter en parallèle sur différents enregistrements, l'état partagé doit donc être géré avec précaution. De même, l'étape `reduce` exécute une fonction par clé unique avec un tableau de ses valeurs.
- Si l'étape Map est omise, NetSuite mélangera l'entrée originale par clé et invoquera `reduce` avec potentiellement plusieurs valeurs par clé (chaque valeur étant une chaîne JSON d'un enregistrement). Inversement, si l'étape Reduce est omise, chaque sortie de map va directement vers summarize.

- Le framework appelle automatiquement `JSON.stringify` sur les objets que vous écrivez via `context.write`. Si vous passez des valeurs qui ne sont pas des chaînes, elles sont converties en chaînes JSON à la volée (Source: houseblend.io). Dans tout code personnalisé, il est courant d'appeler `JSON.parse(context.value)` au début d'une fonction `map` pour obtenir un objet utilisable. Par exemple, en pseudo-code :

```
function map(context) {
    var searchResultObj = JSON.parse(context.value);
    // Maintenant, searchResultObj.id, searchResultObj.values.fieldName sont accessibles
}
```

  Ce modèle apparaît dans de nombreux exemples et réponses StackOverflow (Source: houseblend.io) (Source: cloud.tencent.com).

Objets de contexte Map/Reduce : Détails officiels

La documentation SuiteCloud d'Oracle décrit en détail les objets de contexte Map/Reduce (Source: docs.oracle.com) (Source: docs.oracle.com). Les points clés pertinents pour `context.value` incluent :

- **Propriété `mapContext.value` :** Officiellement définie comme *« La valeur à traiter pendant l'étape map. »* (Source: docs.oracle.com). Le document indique : si l'entrée est un ensemble de résultats de recherche, alors `mapContext.value` est un objet `search.Result` *converti en chaîne JSON à l'aide de* `JSON.stringify()` (Source: docs.oracle.com). Cela confirme explicitement que, pour les entrées de recherche, l'étape Map reçoit du texte JSON, et non un objet actif. (Cette sérialisation est ce qui rend possibles l'exécution parallèle et la planification ultérieure.)
- **Propriété `mapContext.key` :** La clé à traiter pendant cette invocation de map. Lorsque l'entrée est une recherche, la clé est l'ID interne de l'enregistrement (sous forme de chaîne) (Source: docs.oracle.com).
- **Propriété `reduceContext.values` :** Contient les valeurs émises par les travaux map pour cette clé (sous forme de string[]). Si l'étape map a écrit des paires clé/valeur, elles arrivent ici. (Pas directement dans `context.value`.)

Au-delà des documents officiels, les sources communautaires confirment que mapContext.value est du JSON. Par exemple, un blog NetSuite récent note explicitement : *« mapContext.value : La valeur actuelle associée à* `mapContext.key`*. Il s'agit généralement d'une chaîne JSON dérivée d'un résultat de recherche ou d'une autre source de données. »* (Source: suitescriptwithramesh.blogspot.com). Cela s'aligne avec le document officiel et confirme que les développeurs doivent s'attendre à du JSON.

En interne, NetSuite garantit que chaque fois que des données circulent entre les étapes, elles sont sérialisées sous forme de chaînes JSON. Par exemple, le guide Map/Reduce de Houseblend explique : *« NetSuite garantit que les données sont transmises sous forme de chaînes sérialisées... Le système utilise automatiquement* `JSON.stringify()` *sur les clés/valeurs si elles ne sont pas déjà des chaînes... Dans votre code map, vous appelez généralement* `JSON.parse()` *sur* `context.value` *s'il contient des données JSON »* (Source: houseblend.io). En bref, **tout ce que vous placez dans `context.write` en tant qu'objet apparaîtra sous forme de chaîne JSON dans le `context.value` de l'étape suivante.** Inversement, si vous renvoyez une recherche ou un tableau d'objets dans getInputData, l'étape Map verra la forme de chaîne sérialisée.

Ces comportements ont plusieurs implications :

- **Indépendance vis-à-vis du langage :** En sérialisant les données sous forme JSON, Map/Reduce isole le contexte d'exécution de chaque travail. Aucune référence d'objet JavaScript ne traverse les threads ; tout circule sous forme de texte.
- **Types de données :** Toutes les données dans `context.value` seront du JSON textuel. Les champs numériques dans la recherche apparaîtront comme des nombres ou des chaînes JSON ; les dates apparaîtront comme des chaînes. Les développeurs doivent convertir les types si nécessaire. (Par exemple, l'API de recherche renvoie souvent les champs numériques ou de date sous forme de chaînes — les développeurs doivent utiliser `parseInt/parseFloat` ou les constructeurs `Date` appropriés.)
- **Limites de taille :** Chaque chaîne `context.value` est soumise aux limites de la plateforme. Le document Oracle avertit que « chaque valeur [dans mapContext] ne peut pas dépasser 1 mégaoctet » (Source: docs.oracle.com). Houseblend et d'autres sources notent que la limite actuelle semble être d'environ 10 Mo (Source: houseblend.io). Quoi qu'il en soit, les résultats de recherche extrêmement volumineux (milliers de colonnes ou texte énorme) doivent être réduits ou divisés. Connaître la surcharge JSON est important lors de la conception des requêtes.

Ensuite, nous décrivons **comment le JSON est structuré** pour un résultat de recherche typique. C'est le cœur de la compréhension de la manière de l'analyser efficacement.

Structure du JSON de résultat de recherche

Lorsqu'un résultat de recherche est transmis à l'étape Map, `context.value` contient une chaîne JSON qui représente l'objet `search.Result`. Bien que NetSuite ne publie pas le schéma exact de ce JSON, une observation et une documentation approfondies (y compris des exemples dans les forums et les réponses) révèlent sa forme typique. Le JSON possède généralement trois clés de niveau supérieur :

1. **`recordType`** – Une chaîne indiquant le type d'enregistrement NetSuite du résultat (par ex. `"customer"`, `"salesorder"`, `"item"`, etc.). Cela correspond à la propriété `search.Result.recordType` (Source: docs.oracle.com). Par exemple, si la recherche porte sur des factures, `recordType` pourrait être `"invoice"`. Le type est la valeur de l'énumération SuiteScript `search.Type` sous forme de chaîne.

2. **`id`** – Une chaîne contenant l'ID interne de cet enregistrement (Source: docs.oracle.com). (Bien qu'il soit numérique, il est sérialisé sous forme de chaîne. Les documents de NetSuite notent : *« L'ID interne est un nombre, mais il est stocké sous forme de chaîne »* (Source: docs.oracle.com).) Par exemple, `"12345"`.

3. **`values`** – Un objet dont les clés sont les ID (ou noms) des colonnes de recherche demandées, et dont les valeurs représentent les valeurs de colonne pour cette ligne. L'objet `values` encapsule tous les champs sélectionnés de la recherche enregistrée, les champs de formule, les champs joints, etc.

Au sein de l'objet `values` :

- Pour les **champs simples** (comme les colonnes de texte, les dates, les colonnes numériques ou les colonnes de formule qui renvoient une valeur unique), la valeur JSON est généralement une chaîne ou un nombre brut. Par exemple, une colonne nommée `"tranid"` (numéro de transaction) pourrait apparaître comme `"1567"`, ou une colonne de date comme `"2023-09-01"` (une chaîne). Quelques exemples : dans l'extrait JSON ci-dessous, `"enddate":"10/13/2017"` et `"formulanumeric":"65"` sont des valeurs simples (Source: stackoverflow.com).

- Pour les **champs de liste/enregistrement** (champs où la valeur est elle-même une référence d'enregistrement, comme une entité, un article, un client, un département, etc.), la valeur JSON est un objet avec deux propriétés : `"value"` et `"text"`. La `"value"` est généralement l'ID interne de l'enregistrement référencé (sous forme de chaîne ou de nombre), et `"text"` est la chaîne d'affichage. Par exemple, si une colonne est `customer` (un champ de liste), vous pourriez voir `"customer": {"value":"67","text":"Acme Corporation"}`. Dans le cas de champs joints, la clé dans `values` peut inclure le nom de la jointure (voir ci-dessous), mais le modèle est le même : un objet avec `.value` et `.text` (Source: stackoverflow.com) (Source: cloud.tencent.com).

- **Champs joints :** Si la recherche inclut des colonnes provenant d'enregistrements liés (jointures), ces colonnes sont indexées en combinant l'alias de jointure et le nom du champ, souvent séparés par un point. Par exemple, dans une recherche de commande client qui se joint à l'enregistrement `item`, une colonne pourrait être `"item.workOrder"`. Dans le JSON, on pourrait voir : `"item.workOrder": {"value":"1517","text":"Agent Orange AOP 1/2"}` (Source: stackoverflow.com). Notez le point dans le nom de la clé. Cela a été illustré dans un exemple StackOverflow où `data["item.workOrder"].value` était utilisé pour extraire l'ID de l'ordre de fabrication (Source: stackoverflow.com).

Voici un exemple d'un résultat de recherche unique converti en JSON (tiré de (Source: stackoverflow.com):

```
{
  "recordType": "manufacturingoperationtask",
  "id": "1974",
  "values": {
    "item.workOrder":   {"value": "1517", "text": "Agent Orange Pale Ale : AOP 1/2"},
    "enddate":         "10/13/2017",
    "formulanumeric":  "65"
  }
}
```

Dans cet extrait :

- `recordType` est `"manufacturingoperationtask"` (type d'enregistrement).
- `id` est `"1974"`.
- Sous `values`, nous avons trois champs :
  - `"item.workOrder": {value: "1517", text: "Agent Orange Pale Ale : AOP 1/2"}` – un champ joint (en supposant que `item.workOrder` soit un champ sur les tâches d'opération de fabrication). Nous extrayons sa valeur via `data["item.workOrder"].value`.
  - `"enddate": "10/13/2017"` – un champ de date affiché sous forme de chaîne.
  - `"formulanumeric": "65"` – une colonne de formule (résultat numérique) sous forme de chaîne.

Dans de nombreux cas, les clés dans `values` correspondent à l'**ID de script** ou à l'**étiquette de résumé** de la colonne. Si une colonne a une étiquette comme « End Date », sa clé peut être l'identifiant interne sans espaces, tel que `"enddate"`. Les formules personnalisées ou les ID de colonne de recherche enregistrée peuvent avoir des clés générées par le système.

Pour clarifier la structure JSON, considérez ce **tableau Markdown** résumant les composants typiques du JSON de résultat de recherche :

CLÉ OU CHEMIN JSON	DESCRIPTION	EXEMPLE DE VALEUR	NOTES/SOURCE
`recordType`	Le type d'enregistrement de la ligne de résultat (SuiteScript `search.Type`) (Source: docs.oracle.com). Chaîne.	`"customer"`, `"invoice"`, etc.	Par ex. `"salesorder"`, `"manufacturingoperationtask"`.
`id`	L'ID interne de l'enregistrement sous forme de chaîne (Source: docs.oracle.com).	`"12345"`	ID numérique stocké sous forme de chaîne.
`values`	**Objet** contenant les valeurs de colonne. Les clés sont des ID ou noms de colonne.	—	Chaque clé sous `values` contient les données de colonne.
`values.<fieldName>`	Valeur d'un champ *non-liste* ou d'une formule.	`"100.00"`, `"2023-09-05"`	Nombre ou chaîne selon le champ ; pas de sous-objet.
`values.<listField>.value`	Pour les **champs de liste/enregistrement**, la valeur interne (ID) de l'enregistrement référencé.	`"67"` (ID client)	Si le champ est une recherche (entité, article, etc.).
`values.<listField>.text`	Pour les champs de liste, le texte d'affichage (description) de l'enregistrement référencé.	`"Acme Corporation"`	Nom lisible par l'homme.
`values.<join>.<field>`	Pour les **champs joints**, les clés combinent jointure et champ (par ex. `"item.workOrder"`).	(sous-objet ou brut comme ci-dessus)	Accès avec notation entre crochets : `data["item.workOrder"]`.

Le tableau ci-dessus souligne que **l'analyse du JSON nécessite de connaître la structure des champs**. Pour les colonnes simples, on lit directement `result.values.fieldName`. Pour les champs de liste/enregistrement, on explore `.value`. Pour les champs joints, il peut être nécessaire d'utiliser les crochets `[...]` avec la clé complète (incluant le '`.`').

Les **types** de données renvoyés comptent également. Même si une colonne est numérique, le JSON peut la représenter sous forme de chaîne. Par exemple, dans [14], `"formulanumeric": "65"` est une donnée numérique stockée sous la forme de la chaîne `"65"`. Les développeurs doivent convertir les types si nécessaire. Les dates arriveront également sous forme de représentations textuelles (par ex. `"2023-09-07"`) et peuvent être analysées avec `Date.parse()` de JavaScript ou similaire.

Liste de modèles spécifiques :

- **Champs de valeur directe :** Si la colonne renvoie un scalaire unique (texte, nombre, date), `result.values.fieldId` ou `result.values["fieldId"]` donne la valeur directement (chaîne/nombre). Exemple tiré d'un script Map :

```
var searchResult = JSON.parse(context.value);
var invoiceId = searchResult.values.tranid; // par ex. "1001"
```

- **Champs de liste/enregistrement :** Ceux-ci apparaissent sous forme d'objets. Par exemple, si `entity` est un champ client, alors :

```
var customer = searchResult.values.entity;        // par ex. {value:"67", text:"Acme Corp"}
var customerId = customer.value;                 // "67"
var customerName = customer.text;                // "Acme Corp"
```

  Ou simplement : `var customerId = searchResult.values.entity.value;`. Ce modèle est clairement illustré dans un exemple de fonction Map où `entityId = searchResult.values.entity.value;` (Source: cloud.tencent.com).

- **Champs joints :** Si la recherche inclut, par exemple, l'ordre de fabrication d'un article, la clé JSON peut contenir un point : `result.values["item.workOrder"]`. Vous devez utiliser la notation entre crochets : `result.values["item.workOrder"].value` pour obtenir l'ID (Source: stackoverflow.com) (Source: stackoverflow.com). (L'utilisation d'un littéral d'objet avec un point ne fonctionnera pas dans les noms de variables JavaScript.)

- **Recherche agrégée/Résumé :** Dans le cas de recherches enregistrées avec résumé (regroupement, fonctions d'agrégation), les clés JSON peuvent ressembler à `"GROUP(vendor.entityid)"` ou similaire (la clé exacte est souvent la définition de la colonne de recherche). Celles-ci apparaissent également dans `values`. L'approche d'analyse est la même : JSON.parse suivi de la lecture de la clé, bien qu'il puisse être nécessaire de s'adapter aux noms de clés inhabituels.

Exemple de ligne de résumé : Lorsqu'une recherche comporte des colonnes de résumé, une clé Map peut représenter un groupe agrégé (avec une clé synthétique) et le JSON montre des valeurs agrégées. L'analyse est analogue, mais les valeurs peuvent représenter des nombres agrégés ou du texte concaténé.

Enfin, notez que **`JSON.stringify` sur un `search.Result` de NetSuite** inclut *tous* les champs et valeurs tels que décrits, mais **n'inclut pas les méthodes ou les propriétés non énumérables**. Il sérialise essentiellement les données du résultat. Cela signifie que le JSON capturera exactement ce que vous obtiendriez en appelant `search.Result.getValue()` et `getText()`, mais sous une forme de données brutes.

Modèles d'analyse et meilleures pratiques

Compte tenu de la structure JSON décrite ci-dessus, la tâche pour l'étape Map est d' *analyser* cette chaîne et d'extraire les valeurs nécessaires. Un idiome courant est :

```
function map(context) {
```

// Analyser la chaîne JSON du résultat de recherche en un objet var result = JSON.parse(context.value);

// Accéder aux propriétés standards var recordId = result.id; var recordType = result.recordType;

// Accéder aux valeurs des champs var someText = result.values.someTextField; var someNumber = parseFloat(result.values.someNumericField); var aDate = new Date(result.values.someDateField);

// Accéder aux champs de liste/enregistrement var lookupValue = result.values.someListField.value; var lookupText = result.values.someListField.text;

// Accéder aux champs joints (avec un point dans la clé) var joinedId = result.values["item.workOrder"].value;

// ... traiter l'enregistrement ... // Émettre éventuellement une nouvelle paire clé/valeur context.write({ key: recordId, value: recordType }); }

```
Ce modèle apparaît dans de nombreux exemples de la communauté (Source: [cloud.tencent.com](https://cloud.tencent.com/deve

```js
var searchResult = JSON.parse(context.value);
var invoiceId   = searchResult.id;
var entityId    = searchResult.values.entity.value;
```

Ce code extrait l' `id` et l'ID interne de la colonne `entity` à partir du JSON analysé (Source: cloud.tencent.com). Il utilise ensuite `context.write({key: entityId, value: invoiceId})` pour regrouper par entité dans l'étape de réduction (reduce).

Nous décrivons ci-dessous les **modèles dérivés et considérations** pour l'analyse (parsing) :

- **JSON.parse est obligatoire :** Comme `context.value` est une chaîne de caractères, vous devez appeler `JSON.parse(context.value)` (ou équivalent) pour obtenir un objet JavaScript. Ne pas l'analyser signifierait devoir manipuler la chaîne, ce qui n'est pas pratique. Presque tous les exemples font exactement cela (Source: houseblend.io) (Source: suitescriptwithramesh.blogspot.com). **Exception :** Si votre fonction `getInputData` a renvoyé un tableau littéral de valeurs au lieu d'une recherche, alors `mapContext.value` pourrait déjà être une primitive (nombre/chaîne) ou un objet issu de l'entrée. Mais pour un résultat de recherche (le sujet ici), l'analyse est nécessaire.

- **Analyse sécurisée :** Le JSON dans le contexte est bien formé (fourni par NetSuite), donc les erreurs d'analyse sont peu probables à moins que la recherche ne contienne des caractères inhabituels. Néanmoins, il est de bonne pratique d'encapsuler l'analyse dans un bloc try/catch si les données peuvent être suspectes.

- **Conversion de type :** Après l'analyse, les valeurs issues de `result.values` arrivent sous forme de chaînes (comme noté). Si vous avez besoin d'opérations numériques, convertissez explicitement. Ex : `var qty = parseInt(result.values.quantitypacked, 10);`. Il en va de même pour les dates.

- **Clés de champs joints :** Pour extraire un champ joint, utilisez la notation entre crochets. Par exemple, si la clé est `"item.workOrder"`, alors dans le code : `var id = result.values["item.workOrder"].value;`. Exemple tiré de StackOverflow :

  ```
  var data = JSON.parse(result);
  var workOrderId = data["item.workOrder"].value;
  ```

  où `result` était le texte JSON (Source: stackoverflow.com).

- **Utilisation de l'API search.Result à la place :** Parfois, au lieu d'analyser le JSON, on peut utiliser directement les méthodes de l'objet `search.Result` dans l'étape Map. Si vous renvoyez un objet de recherche dans `getInputData` (plutôt qu'une *exécution* de recherche), alors dans l'étape Map, `context.value` est déjà un objet `search.Result` (pas une chaîne), et vous pouvez appeler `getValue({name:"field", join:"...", summary:...})`. Cependant, la documentation de la suite indique que lorsque l'entrée est une recherche (ResultSet), `context.value` est sérialisé (docs.oracle.com. En pratique, la fonction Map ne reçoit qu'une chaîne JSON, pas l'objet `Result` actif. Par conséquent, l'analyse est généralement nécessaire (les réponses StackOverflow montrent systématiquement l'utilisation de `JSON.parse` (Source: stackoverflow.com) (Source: cloud.tencent.com). La seule exception est si un développeur place manuellement un tableau ou un objet dans `context.write` puis le lit dans le reduce, mais là encore, ces données sont également sérialisées.

- **Itération sur plusieurs valeurs :** Si un map émet plusieurs valeurs par clé (via plusieurs appels `context.write`), alors dans le reduce, `context.values` sera un tableau de ces chaînes JSON. Un modèle courant dans le reduce est `context.values.forEach(val => { let obj = JSON.parse(val); ... });`. Pour faire une synthèse, utilisez `summaryContext.output/*.iterator*/` à la place (Source: stackoverflow.com).

- **Considérations sur la mémoire :** Comme `context.value` peut être volumineux, soyez attentif à la mémoire. Évitez les copies inutiles et n'analysez qu'une seule fois par valeur. Attention également : `JSON.parse` peut produire des chaînes pour les champs verrouillés (non, ils produisent des nombres corrects, mais vérifiez). Assurez-vous que les grands tableaux de valeurs sont traités de manière streamée si possible.

- **Exemple en pratique :** Un exemple de code explicite (tiré de [73]) :
  ```

```
function map(context) {
  var searchResult = JSON.parse(context.value);
  var invoiceId = searchResult.id;
  var entityId = searchResult.values.entity.value;
  // appliquer la logique de remise...
  context.write({
    key: entityId,
    value: invoiceId
  });
}
```

Ce code a analysé le JSON et accédé aux champs nommés. Il a enregistré `entityId` et l'a utilisé comme clé de regroupement (Source: cloud.tencent.com).

- **JSON de l'étape Reduce :** Dans l'étape reduce, `reduceContext.values` est généralement un tableau de chaînes JSON (si le map a écrit du JSON). Vous analysez souvent chacune d'elles : `var objs = context.values.map(JSON.parse);` (cela a été suggéré dans [69], bien que commenté). L'exemple [69†L124-L129] montre l'utilisation de `context.values.map(JSON.parse)` pour transformer toutes les valeurs.
- **Étape Summarize :** Si vous passez des valeurs du reduce au summarize (en utilisant `context.write` dans le reduce), le résumé les reçoit dans `summaryContext.output`. On ne peut pas faire `JSON.parse(context.values)` ici car `context.values` n'existe pas dans le summarize (Source: stackoverflow.com). Utilisez plutôt `summaryContext.output.iterator().each((key,val) => { ...})` (Source: stackoverflow.com). La réponse citée [30] montre comment assembler des références de fichiers en itérant sur `ctx.output.iterator()` dans l'étape de résumé.

Tableau des exemples d'analyse

À titre d'illustration, le tableau suivant résume les **types de champs courants** et la façon dont leurs données apparaissent dans le JSON, ainsi que des exemples de code pour les extraire :

CHAMP/REPRÉSENTATION JSON	TYPE DE COLONNE	MODÈLE D'EXTRACTION	SOURCE EXEMPLE/API
`"amount": "100.00"`	Numérique (formule)	`result.values.amount`	donne `"100.00"` (chaîne)
`"trandate": "2023-09-05"`	Date	`result.values.trandate`	donne `"2023-09-05"`
`entity: {"value":"34","text":"Acme Corp"}`	Liste (Client)	`result.values.entity.value` → `"34"`	[73†L170-L174] (exemple entité)
`item: {"value":"172","text":"Widget A"}`	Liste (Article)	`result.values.item.value` → `"172"`	[73†L170-L174] (si `columns: ['item']`)
`"item.workOrder": {"value":"1517","text":"X"}`	Champ joint	`result.values["item.workOrder"].value` → `"1517"`	[14†L19-L23] (exemple jointure)
`"custbody_notes": "Important"`	Champ texte	`result.values.custbody_notes` → `"Important"`	chaîne directe

> *Tableau : Modèles d'analyse pour différents types de valeurs de colonnes de recherche. Les champs simples apparaissent directement sous forme de chaînes ; les champs de liste/enregistrement deviennent des objets (accédez à `.value` et éventuellement `.text`) ; les champs joints incluent l'alias de jointure dans la clé (utilisez la notation entre crochets) (Source: stackoverflow.com) (Source: cloud.tencent.com).*

Résumé et cas limites

- **Valeurs vides :** Si un champ n'a pas de valeur (null), il apparaît généralement comme `null` ou la clé peut être absente. `JSON.parse` convertira le `null` JSON en `null` en JavaScript. Vérifiez toujours la présence de `undefined` ou `null` avant d'utiliser la valeur.
- **Champs de type chaîne :** Même les colonnes de texte brut apparaissent comme des chaînes JSON dans les valeurs. Les guillemets et les caractères spéciaux sont échappés selon les règles JSON.
- **Champs à sélection multiple :** Parfois, une sélection multiple (liste à valeurs multiples) apparaîtra comme un tableau d'objets. Par exemple, un champ personnalisé à sélection multiple pourrait apparaître comme : `"custfield_colors": [ {"value":"1","text":"Rouge"}, {"value":"2","text":"Bleu"} ]`. Il faut boucler sur le tableau. (Ce format a été observé en pratique, bien qu'il ne soit pas officiellement documenté.)
- **Conseil de performance :** Si seuls quelques champs sont nécessaires, définissez vos colonnes de recherche de manière étroite afin que le JSON soit petit. Sinon, `context.value` peut être volumineux ; chaque instance de map effectue alors plus de travail d'analyse et utilise plus de mémoire. Utilisez les méthodes `search.Result.search()` ou `search.runPaged` en dehors du Map/Reduce pour les très grands ensembles de données si nécessaire.

Études de cas et exemples

Traitement en masse des factures : Comme noté dans la documentation de NetSuite et les exemples de la communauté, une utilisation typique de Map/Reduce est le traitement en masse des factures ou des paiements par client (Source: docs.oracle.com). Par exemple, un système peut rechercher toutes les factures ouvertes. L'étape Map émet ensuite `(CustomerID, InvoiceData)` pour chaque facture, où `InvoiceData` est analysé à partir de `context.value`. Le Shuffle regroupe toutes les factures par client. L'étape Reduce peut alors créer un paiement consolidé pour chaque client, en utilisant le tableau des objets de facture analysés. Sous forme narrative :

> *Par exemple, supposons que 100 factures appartiennent à 5 clients. Dans `getInputData`, nous chargeons la recherche de ces factures. L'étape Map s'exécute 100 fois, chaque fois avec `context.key = <invoiceID>` et `context.value` = JSON de la facture. Nous l'analysons :*
> *`var rec = JSON.parse(context.value); var customerId = rec.values.entity.value;`*
> *Nous faisons ensuite `context.write({key: customerId, value: JSON.stringify(rec)});`.*
> *L'étape Shuffle regroupe les factures par ID client. Le Reduce reçoit alors 5 appels, chacun avec key=customerId et values = [tableau des factures JSON]. Dans chaque fonction reduce, nous analysons chaque JSON dans les valeurs (`values.forEach(v => JSON.parse(v))`) et agrégeons selon les besoins.*
> *Enfin, le summarize enregistre le résultat ou envoie des notifications.*

Cet exemple est cohérent avec les modèles et confirme comment `context.value` est utilisé (Source: docs.oracle.com) (Source: cloud.tencent.com).

Exemple de champ joint : Un autre cas est celui où une recherche inclut des enregistrements joints. Considérez une recherche enregistrée sur des enregistrements de travail qui se joignent via `item` au `workOrder` associé. L'étape Map verra des clés avec des noms de jointure pointés. Un exemple de JSON (simplifié) pourrait être :

```
{"recordType":"workrecord","id":"999","values":{
   "item.workOrder":{"value":"1517","text":"Order XYZ"},
   "status":"Pending"
}}
```

Pour obtenir l'ID de l'ordre de travail, le code doit faire :

```
var parsed = JSON.parse(context.value);
var workOrderId = parsed.values["item.workOrder"].value;
```

Ce modèle a été explicitement discuté par un contributeur StackOverflow de l'ère NetSuite (Source: stackoverflow.com), illustrant que les clés JSON pour les jointures doivent être accédées avec la notation entre crochets.

Performance/Débit : En activant les jobs map parallèles, Map/Reduce peut réduire considérablement le temps de traitement. Par exemple, la mise à jour de 10 000 enregistrements séquentiellement dans un script pourrait prendre beaucoup de temps, alors que la division en 5 jobs parallèles de 2 000 chacun peut se terminer beaucoup plus rapidement (Source: houseblend.io). Dans une étude de cas (paraphrasée de [64]), un détaillant a rapporté que l'utilisation de Map/Reduce a réduit son traitement par lots nocturne de plusieurs heures à moins d'une heure en répartissant la charge entre les processeurs. (Les chiffres exacts varient selon le cas d'utilisation, mais le principe est soutenu par les conseils de NetSuite (Source: houseblend.io).)

Gouvernance et résilience : Étant donné que chaque invocation de map ou reduce a sa propre limite de gouvernance (1 000 unités pour le map, 5 000 pour la génération du reduce en 2.x), les très gros jobs peuvent être des étapes de nombreuses exécutions. Map/Reduce s'interrompt automatiquement si une invocation approche de sa limite (Source: houseblend.io). Par exemple, un script qui traite des millions de lignes peut s'exécuter en continu sans intervention manuelle, grâce à ce fractionnement. Cependant, de très grandes valeurs JSON pourraient toujours atteindre la limite de 10 Mo, donc en pratique, on pourrait filtrer ou réduire les données dans `getInputData` (par exemple, utiliser `search.createColumn({ name: "internalid" })` plutôt que de sélectionner tous les champs).

Approches alternatives : Comme le note Mark Dietrich, les développeurs utilisent de plus en plus les requêtes **SuiteQL** au sein de Map/Reduce, surtout s'ils ont besoin de très grands ensembles de résultats ou d'un meilleur contrôle sur le formatage des données (Source: timdietrich.me). SuiteQL peut renvoyer des objets JSON purs dans l'étape map (via le module `N/query`), ce qui peut parfois être plus facile à manipuler que le JSON de résultat de recherche standard, mais le modèle général d'analyse reste le même (`JSON.parse(context.value)` si nécessaire). SuiteAnswers (la base de connaissances interne de NetSuite) inclut des exemples d'utilisation de SuiteQL dans `getInputData` et de mappage de ses résultats (Source: timdietrich.me). La tendance suggère qu'à mesure que SuiteQL mûrira, la dépendance aux anciens modèles `N/search` pourrait diminuer, bien que l'analyse JSON sera toujours impliquée si l'on émet des objets bruts.

Implications et orientations futures

Tendances de l'industrie – Le JSON partout : L'utilisation du JSON dans le Map/Reduce de NetSuite s'aligne sur les tendances de l'industrie vers les API basées sur JSON. Les enquêtes montrent qu'en moyenne **~97 % des API web utilisent JSON** (Source: www.glyphwidgets.com). Les développeurs travaillant sur différents systèmes connaissent très bien le JSON, ce qui en fait un choix naturel pour l'échange de données. L'adoption par NetSuite du JSON pour le flux de données de script interne garantit la compatibilité avec les services web et les plateformes d'intégration, et évite la sérialisation personnalisée.

Volumes de données croissants et traitement parallèle : Les entreprises continuent de gérer des ensembles de données toujours plus volumineux. Les dirigeants de NetSuite font état d'une croissance annuelle de 18 % des unités d'utilisation et de milliards de transactions ERP dans le cloud (Source: www.anchorgroup.tech) (Source: houseblend.io). Le modèle « diviser pour régner » de Map/Reduce pour les opérations en masse ne fera que devenir plus essentiel. Par exemple, l'intégration de l'IA et de l'apprentissage automatique (comme le font 65 % des organisations (Source: www.anchorgroup.tech) nécessite souvent le prétraitement de grands ensembles de données. Map/Reduce dans SuiteScript pourrait être utilisé à l'avenir pour préparer des flux de données pour des charges de travail analytiques ou des modèles d'apprentissage automatique au sein de NetSuite ou de systèmes connectés, rendant la compréhension de l'analyse de `context.value` importante au-delà des mises à jour d'enregistrements traditionnelles.

Évolution technique – SuiteQL et au-delà : L'introduction par NetSuite de SuiteQL (une API de requête de type SQL) signifie que les développeurs peuvent récupérer des données via des requêtes SQL. Comme le note Dietrich (Source: timdietrich.me), l'utilisation de SuiteQL dans Map/Reduce peut produire une **sortie JSON bien structurée** qui peut éliminer certains des tracas d'analyse manuelle du JSON de `search.Result`. En d'autres termes, au lieu de traiter l'objet `"values": {...}`, un Map/Reduce SuiteQL pourrait renvoyer directement des objets avec des propriétés nommées pour chaque colonne, augmentant potentiellement la clarté. Cependant, le support de SuiteQL dans Map/Reduce est encore nouveau (SuiteScript 2.x avec `N/query` nécessite le paramètre de version 2.1, au moment de la rédaction). Avec le temps, nous pourrions voir davantage de méthodes intégrées pour gérer l'entrée Map/Reduce qui contournent complètement le JSON, mais actuellement, la méthode documentée reste via les résultats de recherche. Le passage à SuiteQL peut être considéré comme faisant partie d'une orientation future plus large visant à moderniser l'accès aux données dans NetSuite (analogue à la façon dont de nombreux systèmes ERP ajoutent des couches de requête SQL et OData).

Veille sur les meilleures pratiques : Les futures mises à jour de la documentation pourraient ajuster les limites sur la taille de `context.value`, ou modifier les comportements par défaut (par exemple, introduire la pagination automatiquement, ou le streaming natif des résultats). Les développeurs doivent surveiller les notes de version de SuiteCloud. Pour l'instant, la meilleure pratique consiste à garder la sortie petite et à analyser le JSON judicieusement. En utilisant les rapports de gouvernance de NetSuite, on peut analyser combien de temps est passé dans `JSON.parse` et s'ajuster en conséquence.

Conclusion

En conclusion, **le `context.value` de SuiteScript Map/Reduce pour les résultats de recherche est fondamentalement une chaîne JSON** qui encapsule une ligne `search.Result` (Source: docs.oracle.com) (Source: suitescriptwithramesh.blogspot.com). Comprendre parfaitement sa structure — `recordType`, `id` et l'objet imbriqué `values` — est crucial pour une extraction correcte des données dans les scripts Map/Reduce. Grâce à des exemples détaillés et à l'expérience de la communauté, nous avons constaté que le modèle d'analyse habituel est `let rs = JSON.parse(context.value)`, suivi de l'accès à `rs.values.fieldName` ou `rs.values["join.field"]` (Source: cloud.tencent.com) (Source: stackoverflow.com).

D'un **point de vue technique**, l'approche JSON permet à NetSuite de paralléliser les tâches en toute sécurité, au prix de la nécessité d'une sérialisation en chaîne de caractères. Les développeurs bénéficient de la flexibilité des objets JavaScript une fois analysés, mais doivent également gérer les particularités du format JSON (telles que la conversion des types et la gestion des clés avec points). Nous avons mis en évidence des stratégies clés : toujours analyser une seule fois, se protéger contre les valeurs nulles, convertir les formats et privilégier les méthodes natives `getValue` lorsque cela est possible pour éviter la surcharge liée au JSON (Source: stackoverflow.com).

D'un **point de vue architectural et de performance**, les scripts Map/Reduce offrent des avantages de mise à l'échelle considérables (Source: houseblend.io). Diviser des tâches de 10 000 enregistrements en 5 travaux simultanés, par exemple, peut réduire considérablement le temps écoulé. Cette capacité est de plus en plus importante à mesure que la base de clients de NetSuite (plus de 40 000 organisations aujourd'hui (Source: www.anchorgroup.tech) traite des ensembles de données toujours plus volumineux. Les fonctionnalités de rendement automatique et de concurrence signifient que Map/Reduce peut gérer des tâches qui submergeraient les approches héritées.

Pour l'avenir, il faut s'attendre à une évolution continue. L'adoption par NetSuite de SuiteQL (Source: timdietrich.me) et des intégrations d'IA suggère que les futurs scripts pourraient davantage s'appuyer sur des requêtes structurées et des échanges JSON. Cependant, même à mesure que les API évoluent, les modèles fondamentaux de passage de clés-valeurs et d'analyse JSON appris ici resteront pertinents. En résumé, la maîtrise de `context.value` et de ses modèles d'analyse est une compétence essentielle pour tout développeur SuiteScript travaillant sur des personnalisations NetSuite gourmandes en données.

Toutes les déclarations ci-dessus sont fondées sur des sources faisant autorité et des exemples pratiques : la documentation officielle d'Oracle (Source: docs.oracle.com) (Source: docs.oracle.com), des blogs d'experts (Source: houseblend.io) (Source: suitescriptwithramesh.blogspot.com), des réponses StackOverflow (Source: stackoverflow.com) (Source: cloud.tencent.com), et des statistiques sectorielles (Source: www.glyphwidgets.com) (Source: www.anchorgroup.tech). En suivant assidûment ces pratiques, les développeurs peuvent écrire des scripts Map/Reduce robustes qui analysent efficacement les résultats de recherche, évitent les pièges courants et gèrent des scénarios de données à grande échelle dans NetSuite.

Étiquettes: suitescript-map-reduce, contextvalue, analyse-json-netsuite, resultats-recherche-sauvegardee, suitescript-20, developpement-netsuite, structure-json, modeles-map-reduce

---