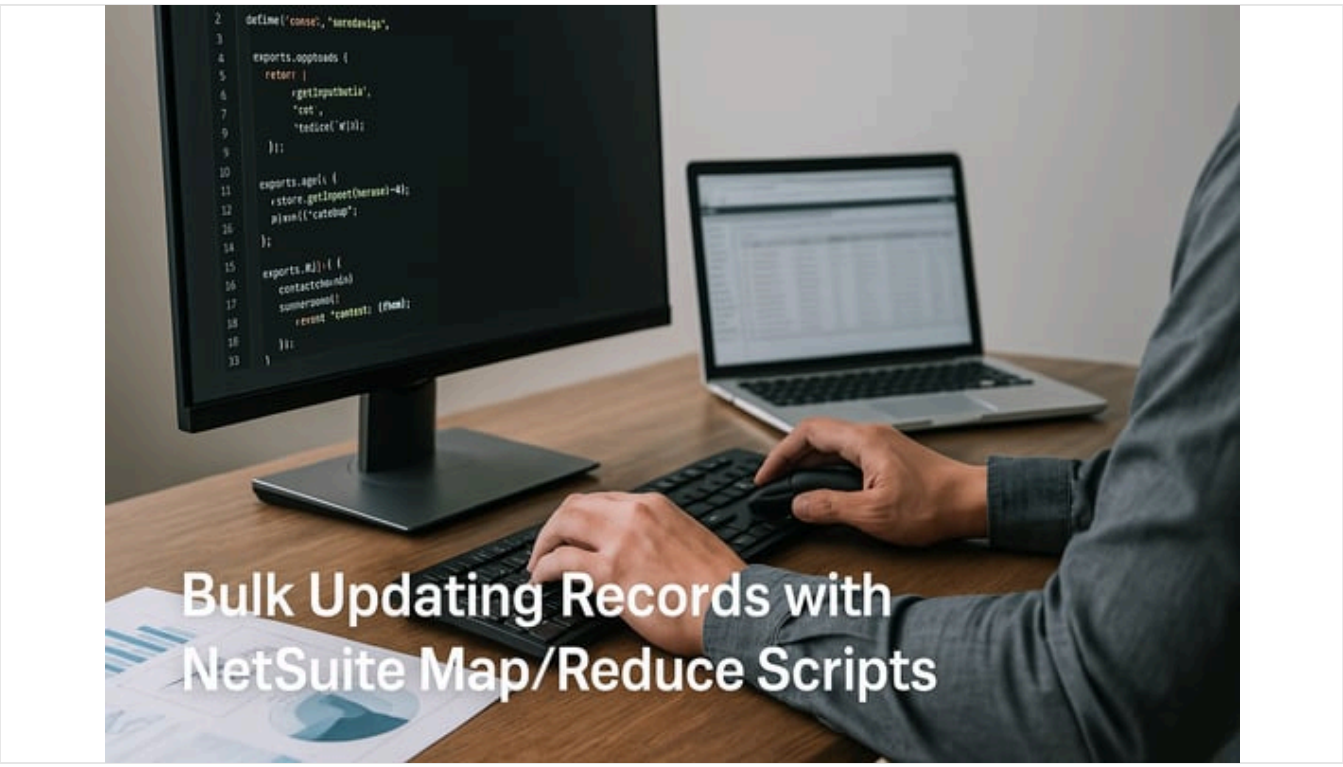
# NetSuite Map/Reduce : Mise à jour en masse d'enregistrements avec SuiteScript

Publié le 2 septembre 2025  55 min de lecture




# Mise à jour en masse d'enregistrements avec les scripts Map/Reduce de NetSuite

**Table des matières**

# Introduction

Le type de script Map/Reduce de NetSuite est un outil puissant pour le traitement en masse de grands volumes d'enregistrements. Il est conçu pour **diviser les tâches importantes en morceaux gérables et les exécuter en parallèle**, ce qui le rend idéal pour les opérations de données intensives et les mises à jour en masse. Cet article propose une exploration approfondie de l'architecture Map/Reduce et démontre comment les développeurs NetSuite SuiteScript expérimentés peuvent l'exploiter pour la mise à jour en masse d'enregistrements tels que les commandes clients, les articles en stock et les enregistrements personnalisés. Nous comparerons Map/Reduce avec d'autres types de scripts (tels que les scripts planifiés et les mises à jour en masse), discuterons des cas d'utilisation avancés et des meilleures pratiques, et fournirons des **exemples de code détaillés** pour chaque étape ( `getInputData` , `map` , `reduce` , `summarize` ). Nous aborderons également les techniques d' optimisation des performances, la gestion de la gouvernance (limite d'utilisation), la journalisation des erreurs, les stratégies de récupération, et noterons les différences entre SuiteScript 2.x et 2.1. L'objectif est de doter les professionnels de NetSuite de connaissances complètes pour construire des scripts Map/Reduce efficaces et résilients pour le traitement de données en masse.

# Comprendre l'architecture Map/Reduce de NetSuite

Les scripts Map/Reduce de NetSuite suivent une **architecture étagée de traitement parallèle** inspirée du paradigme MapReduce. Contrairement à un script planifié à exécution unique, un script Map/Reduce s'exécute en **cinq étapes** (avec quatre points d'entrée définis par le développeur) qui **s'exécutent dans un ordre spécifique** :

- **getInputData** – La première étape (toujours requise). Collecte ou génère l'ensemble de données d'entrée à traiter. S'exécute séquentiellement (un seul thread).
- **Map** – Une étape facultative qui traite chaque entrée et produit des paires clé/valeur intermédiaires. Si elle est utilisée, le framework appelle la fonction `map` une fois par enregistrement d'entrée, exécutant potentiellement de nombreuses exécutions `map` en parallèle (simultanément). Vous pouvez ignorer l'étape map si votre logique ne l'exige pas, mais une étape reduce est alors requise.
- **Shuffle** – Une étape interne (sans code personnalisé) où le système regroupe automatiquement les sorties de map par leurs clés. Toutes les valeurs partageant la même clé sont collectées ensemble. Shuffle s'exécute séquentiellement et prépare les données groupées pour l'étape reduce.
- **Reduce** – Une étape facultative qui traite chaque groupe de valeurs issu de shuffle. Le framework invoque la fonction `reduce` une fois par clé unique (avec un tableau de toutes les valeurs pour cette clé) (Source: netsuitedocumentation1.gitlab.io). Plusieurs tâches `reduce` peuvent s'exécuter en parallèle. Si vous avez ignoré l'étape map, alors reduce est requise (et elle recevra les groupes de la sortie de getInputData, regroupés par clé).
- **Summarize** – L'étape finale (facultative) pour le nettoyage et le rapport. S'exécute une fois séquentiellement après la fin de toutes les autres étapes. Dans `summarize`, vous pouvez agréger les résultats, journaliser les métriques, envoyer des e-mails ou effectuer toute tâche de finalisation (elle est techniquement en dehors du flux de données principal de Map/Reduce).

**Comment les données circulent :** Dans un Map/Reduce, les données passent de getInputData à map (ou directement à reduce si map est ignorée), puis via shuffle à reduce (si défini), et enfin à summarize. À chaque transition, NetSuite s'assure que les données sont transmises sous forme de **chaînes sérialisées** pour éviter les problèmes de référence entre les contextes parallèles. Le système utilise automatiquement `JSON.stringify()` sur les clés/valeurs si elles ne sont pas déjà des chaînes lors de l'écriture d'une étape à l'autre. Dans votre code map ou reduce, vous appelez généralement `JSON.parse()` sur `context.value` s'il contient des données JSON, pour reconstruire des objets utilisables. Gardez à l'esprit que les clés ou valeurs extrêmement grandes ne sont pas autorisées (les clés de plus de 3000 caractères ou les valeurs de plus de 10 Mo généreront des erreurs).

**Exécution séquentielle vs parallèle :** Chaque étape doit se terminer complètement avant que l'étape suivante ne commence. Cependant, au sein de l'étape Map, plusieurs invocations de fonction map peuvent s'exécuter simultanément (dans des "tâches map" NetSuite distinctes), et de même pour l'étape Reduce – plusieurs exécutions reduce peuvent s'exécuter en parallèle, chacune gérant un groupe de clés différent. Le degré de parallélisme est configurable lors du déploiement (le paramètre **"Nombre de tâches map/reduce concurrentes"**) – par exemple, vous pouvez autoriser jusqu'à 2 ou 5 tâches map parallèles pour accélérer le traitement sur de grands ensembles de données. Les étapes getInputData et summarize s'exécutent toujours comme une seule tâche (pas de threads parallèles), et l'étape interne shuffle est également à thread unique.

**Exemple de Map/Reduce :** Pour illustrer, considérons un Map/Reduce qui traite les factures par client (Source: netsuitedocumentation1.gitlab.io). Dans getInputData, nous récupérons toutes les factures nécessitant une attention. Dans l'étape map, chaque facture est émise avec l'ID de son client comme clé et la valeur contenant les données de la facture (Source: netsuitedocumentation1.gitlab.io). S'il y a 5 factures, map s'exécute 5 fois (éventuellement en parallèle) – produisant des paires clé/valeur comme `(Client123 -> Facture1)`, `(Client456 -> Facture2)`, etc. L'étape shuffle les regroupe ensuite par client : supposons que ces 5 factures correspondent à 3 clients uniques, shuffle les regroupera en 3 groupes par clé (Source: netsuitedocumentation1.gitlab.io). L'étape reduce s'exécutera 3 fois (pourrait être en parallèle), recevant à chaque fois un ID client et un tableau des factures de ce client. Dans reduce, nous pourrions effectuer une action par groupe de clients (par exemple, créer un paiement consolidé ou un enregistrement récapitulatif pour chaque client) (Source: netsuitedocumentation1.gitlab.io). Enfin, summarize s'exécute une fois pour éventuellement journaliser le nombre de factures traitées ou envoyer un e-mail de fin de traitement.

**Ignorer Map ou Reduce :** Vous n'êtes pas obligé d'utiliser à la fois map et reduce dans chaque script. Vous pouvez implémenter uniquement une map ou uniquement une reduce en fonction de vos besoins. Si vous ignorez l'étape map, NetSuite prendra les résultats de getInputData et les mélangera/regroupera directement par clé pour l'étape reduce. Si vous ignorez l'étape reduce, la sortie de l'étape map (paires clé/valeur) passe directement à summarize (Source: docs.oracle.com)(Source: docs.oracle.com). **Conseil de conception :** Utilisez map-only lorsque chaque enregistrement peut être traité indépendamment sans avoir besoin d'agréger, et utilisez reduce lorsque vous devez agréger ou regrouper des enregistrements par une clé. Si vous n'avez besoin que d'une seule étape de traitement mais que vous souhaitez des limites de gouvernance plus élevées par exécution, vous pouvez choisir d'implémenter uniquement l'étape reduce (et d'ignorer map) – NetSuite traitera chaque getInputData result comme un groupe distinct, vous donnant une limite d'utilisation plus élevée par itération (l'étape reduce autorise 5 000 unités par invocation contre 1 000 dans map). (Avis de développeur : Certains experts choisissent une approche à étape unique en utilisant uniquement reduce pour une logique très lourde par enregistrement, car **"lorsque vous n'avez besoin que d'une fonction, la fonction reduce offre une plus grande capacité de gouvernance"**.)

En résumé, l'architecture Map/Reduce vous permet de **diviser pour mieux régner sur de grands ensembles de données**. Elle gère automatiquement les détails de bas niveau : le lancement de plusieurs threads d'exécution, l'équilibrage de la charge, la suspension et la replanification si nécessaire, et la collation des résultats. Ensuite, nous examinerons quand préférer Map/Reduce aux autres types de scripts pour les mises à jour en masse.

## Quand utiliser les scripts Map/Reduce par rapport à d'autres types de scripts

Les scripts Map/Reduce chevauchent en termes d'objectif les **scripts planifiés** (et dans une certaine mesure l'outil de mise à jour en masse de NetSuite), mais ils présentent des avantages distincts pour le traitement en masse. Voici les principales considérations concernant le pourquoi et le quand utiliser Map/Reduce :

- **Gestion de grands volumes de données :** Le type de script Map/Reduce a été explicitement conçu pour les scénarios impliquant un *très grand* nombre d'enregistrements ou des transformations de données lourdes qui peuvent être divisées en morceaux indépendants. NetSuite orchestre automatiquement le traitement parallèle de ces morceaux, vous n'avez donc pas à gérer la replanification manuelle ou la logique de division. Si vous avez des milliers d'enregistrements à mettre à jour et que vous pouvez traiter chacun (ou chaque groupe) isolément, Map/Reduce est généralement le meilleur choix.

- **Parallélisme intégré :** Les scripts planifiés (scripts planifiés SuiteScript 1.0 ou 2.x) s'exécutent en un seul thread — une exécution gère l'ensemble des données séquentiellement. En contraste, Map/Reduce peut exécuter le traitement en parallèle sur plusieurs files d'attente (pour les étapes map et reduce), réduisant potentiellement considérablement le temps d'exécution total. Par exemple, un script planifié mettant à jour 10 000 enregistrements s'exécute un par un ; un Map/Reduce pourrait les diviser en, disons, 5 tâches map parallèles de 2 000 enregistrements chacune, terminant beaucoup plus rapidement.

- **Gestion automatique de la gouvernance :** NetSuite impose des limites de gouvernance (unités d'utilisation de l'API, temps de traitement, etc.) aux scripts. Si un script planifié atteint la limite de 10 000 unités d'utilisation en une seule exécution, il doit manuellement se suspendre ou se replanifier. Les scripts Map/Reduce ont une **suspension intégrée** — si une tâche map ou reduce est sur le point de dépasser certaines limites, le framework va **automatiquement suspendre et replanifier** le reste de cette tâche sans intervention du développeur. Cela signifie que les processus Map/Reduce de longue durée peuvent continuer en toute sécurité là où un script planifié pourrait échouer à moins

d'être explicitement codé pour gérer la gouvernance. (Sachez que certaines limites sont "dures" et arrêteront la tâche, mais beaucoup sont "souples" et déclenchent une suspension/reprise automatique. Nous en discutons dans la section Performance/Gouvernance.)

- **Débit total plus élevé :** Une seule exécution de script planifié peut atteindre un maximum de 10 000 unités d'utilisation *pour l'ensemble de l'exécution*, tandis qu'un Map/Reduce réinitialise efficacement la gouvernance à chaque invocation map et reduce. Par exemple, chaque invocation de fonction map peut utiliser jusqu'à 1 000 unités, chaque reduce jusqu'à 5 000, etc., de sorte que collectivement un Map/Reduce peut traiter bien plus de 10 000 unités de travail en le divisant en morceaux. **En d'autres termes, Map/Reduce peut gérer un nombre "illimité" d'enregistrements totaux en fragmentant le travail, tant que chaque fragment reste dans les limites par invocation**. Cela fait de Map/Reduce l'outil de prédilection pour les mises à jour en masse *évolutives*.

- **Codage simplifié pour les opérations en masse :** Le framework Map/Reduce "fait tout pour vous" en termes de planification et de parallélisation des tâches. Vous devez structurer votre code en fonctions d'étape, mais vous n'avez pas à écrire de logique de planification explicite ou de récursion. En revanche, un script planifié 2.x nécessite souvent une suspension manuelle (dans SuiteScript 2.x, vous pourriez utiliser `runtime.getCurrentScript().yield()` ou dans 1.0 `nlapiYieldScript`) à intervalles pour éviter la sortie de gouvernance, et éventuellement une planification externe de plusieurs déploiements si un traitement parallèle est nécessaire – c'est plus complexe et sujet aux erreurs. Map/Reduce abstrait ces préoccupations.

- **Cas d'utilisation nécessitant un regroupement/agrégation :** Si votre opération en masse nécessite d'agréger des données (par exemple, regrouper des transactions par client, ou consolider des enregistrements), Map/Reduce excelle. L'**étape reduce** offre un moyen naturel de rassembler des enregistrements liés par une clé et de les traiter ensemble, ce qu'un script planifié devrait implémenter manuellement (tri, regroupement dans le code). Le mécanisme de shuffle de NetSuite gère le regroupement pour vous avant l'étape reduce.

**Quand envisager les scripts planifiés ou d'autres alternatives :** Si votre tâche est relativement petite ou ne peut pas être facilement divisée en morceaux indépendants, un script planifié pourrait suffire ou même être plus simple. L'aide de NetSuite note que Map/Reduce n'est *"pas aussi bien adapté aux situations où vous souhaitez exécuter une fonction longue et complexe pour chaque partie de votre ensemble de données"*, surtout si chaque partie nécessite une **logique multi-étapes très complexe ou des interdépendances d'enregistrements étendues**. Par exemple, si chaque mise à jour d'enregistrement implique de lourdes opérations relationnelles (chargement et enregistrement de plusieurs enregistrements liés), le faire de manière hautement parallèle pourrait être délicat. Dans de tels cas, un script planifié séquentiel ou un lot de tâches Map/Reduce plus petites pourrait être préférable. De plus, si vous n'avez besoin d'exécuter quelque chose **une seule fois ou rarement** et que cela peut être fait via une simple mise à jour de recherche enregistrée, envisagez la **Mise à jour en masse** de NetSuite :

pour les mises à jour ponctuelles définies par une recherche enregistrée, une mise à jour en masse native (ou un script de mise à jour en masse SuiteScript 1.0) est souvent plus rapide à implémenter. En fait, une recommandation de Stack Overflow le dit succinctement : *"Si vous n'avez besoin de le faire qu'une seule fois et que votre ensemble d'enregistrements peut être défini par une recherche enregistrée, optez pour une mise à jour en masse. Si vous devez le planifier régulièrement ou si vous avez des critères/une logique plus complexes, optez pour un Map/Reduce."*.

De plus, les scripts planifiés sont disponibles dans SuiteScript 1.0 et 2.0, tandis que Map/Reduce est une fonctionnalité uniquement de SuiteScript 2.x. Si vous maintenez des scripts 1.0 plus anciens ou ne pouvez pas utiliser 2.x dans un contexte particulier, vous pourriez utiliser un script planifié (bien que la migration vers 2.x soit fortement recommandée pour tout nouveau développement).

En résumé, **Map/Reduce est généralement le choix préféré pour le traitement en masse d'enregistrements** dans NetSuite lorsqu'il s'agit de grands ensembles de données ou de la nécessité de parallélisme. Il offre une évolutivité et une fiabilité "prêtes à l'emploi" pour les tâches gourmandes en données. Les scripts planifiés peuvent être utilisés pour des tâches planifiées plus simples ou de moindre envergure, et la mise à jour en masse de NetSuite (ou l'importation CSV) peut gérer certaines modifications en masse ponctuelles sans codage. Ensuite, nous examinerons des cas d'utilisation concrets et les meilleures pratiques pour la mise à jour en masse des commandes clients, des articles en stock et des enregistrements personnalisés à l'aide de Map/Reduce.

## Cas d'utilisation avancés pour les mises à jour en masse

L'une des forces de Map/Reduce est d'appliquer la *même* opération à de nombreux enregistrements de manière efficace. Explorons quelques cas d'utilisation avancés où des mises à jour en masse sont nécessaires, et comment Map/Reduce peut être appliqué. Nous mettrons également en évidence les meilleures pratiques spécifiques à chaque scénario (comme la manière de structurer les clés et les étapes).

### Mise à jour en masse des commandes clients (Exemple)

**Cas d'utilisation :** Supposons que vous deviez mettre à jour un grand nombre de commandes clients en masse. Cela pourrait être pour ajuster un champ sur chaque commande (par exemple, cocher une case personnalisée ou mettre à jour un statut), ou pour effectuer des mises à jour au niveau des lignes sur chaque commande (par exemple, mettre à jour tous les articles de ligne qui répondent à certains critères). Un script Map/Reduce peut gérer cela en récupérant toutes les commandes clients pertinentes et en les traitant en parallèle.

**Meilleures pratiques :** Lors de la mise à jour de commandes clients qui ont plusieurs articles de ligne, il est souvent efficace de les *regrouper par commande* afin de pouvoir mettre à jour toutes les lignes nécessaires en une seule fois par commande. Cela évite de charger et d'enregistrer le même enregistrement plusieurs fois. Dans Map/Reduce, vous pouvez y parvenir en utilisant l'ID interne de la commande client comme clé dans l'étape map. Chaque invocation map peut émettre l'ID de la commande comme clé et un ou plusieurs identifiants pour la ou les lignes à mettre à jour comme valeur. Ensuite, une seule invocation reduce obtiendra **un ID de commande client avec un tableau de tous les identifiants de ligne à mettre à jour**, vous permettant de charger cette commande une seule fois et de mettre à jour toutes ses lignes.

**Exemple :** Considérons un scénario où nous voulons trouver toutes les lignes de commande client qui contiennent un certain texte dans un champ de colonne personnalisé `custcol4` (par exemple, contenant la lettre "x"), et modifier la valeur de ce champ sur ces lignes. Nous pouvons écrire getInputData pour rechercher tous les articles de ligne correspondant au critère, puis mapper chaque résultat à sa commande client. Voici un aperçu simplifié du code (SuiteScript 2.x) démontrant ce modèle :

```
Copier
/**
 * @NApiVersion 2.x
 * @NScriptType MapReduceScript
 */
define(['N/search', 'N/record'], function(search, record) {
    function getInputData() {
        // Recherche enregistrée ou définition de recherche pour les lignes de commande
        return search.create({
            type: 'transaction',
            filters: [
                ['mainline','is','F'], 'and',
                ['type','anyof','SalesOrd'], 'and',
                ['custcol4','contains','x']  // lignes où custcol4 contient 'x'
            ],
            columns: ['internalid', 'lineuniquekey']  // obtenir l'ID interne de la comm
        });
    }

    function map(context) {
        // Analyser le résultat de la recherche
        let result = JSON.parse(context.value);
        let salesOrderId = result.id;
        let lineKey = result.values.lineuniquekey;
        // Utiliser l'ID interne de la commande client comme clé, la clé unique de la li
        context.write({
            key: salesOrderId,
            value: lineKey
        });
    }

    function reduce(context) {
        let salesOrderId = context.key;
        let lineKeys = context.values;  // tableau des clés uniques de ligne pour cette
        // Charger l'enregistrement de la commande client une seule fois
        let soRec = record.load({ type: 'salesorder', id: salesOrderId, isDynamic: false
        // Parcourir chaque ligne ciblée et mettre à jour le champ
```

```
        lineKeys.forEach(function(lineKey) {
            let lineNumber = soRec.findSublistLineWithValue({
                sublistId: 'item',
                fieldId: 'lineuniquekey',
                value: lineKey
            });
            if (lineNumber !== -1) {
                soRec.setSublistValue({
                    sublistId: 'item',
                    fieldId: 'custcol4',
                    line: lineNumber,
                    value: 'MapReduced'  // nouvelle valeur pour le champ
                });
            }
        });
        soRec.save();
    }

    function summarize(summary) {
        // (Facultatif) Résumer les résultats ou journaliser les erreurs
        log.audit('Summary', 'Commandes traitées : ' + summary.reduceSummary.keys.iterat
        // Vous pourriez également itérer sur summary.reduceSummary.errors ici pour jou
    }

    return { getInputData, map, reduce, summarize };
});
```

Dans ce script, l'étape **getInputData** effectue une recherche enregistrée pour trouver toutes les lignes de commande client où `custcol4` contient "x". La recherche renvoie des résultats incluant l'ID interne de la commande client et la `lineuniquekey` pour chaque ligne correspondante. L'étape **map** s'exécute pour chaque résultat de recherche (chaque ligne de commande client) et utilise `context.write()` pour produire une paire : clé = ID de la commande client, valeur = clé unique de la ligne. Le framework de NetSuite regroupera automatiquement toutes les clés identiques, de sorte qu'au moment où nous atteignons **reduce**, chaque ID de commande client est traité une fois avec un tableau de toutes ses clés uniques de ligne. Dans l'étape reduce, nous chargeons l'enregistrement de la commande client une seule fois et parcourons toutes les lignes ciblées (en utilisant `findSublistLineWithValue` pour localiser

chaque ligne par sa clé unique) et effectuons les mises à jour, puis enregistrons l'enregistrement. Cette approche est efficace : quel que soit le nombre de lignes par commande nécessitant des modifications, chaque commande n'est chargée et enregistrée qu'une seule fois dans l'étape reduce.

**Considérations de gouvernance :** Le chargement d'un enregistrement et la mise à jour de plusieurs lignes consomment des unités d'utilisation (chargement d'enregistrement + définitions + enregistrement). L'étape reduce permet jusqu'à 5 000 unités d'utilisation par exécution, ce qui est généralement suffisant pour les mises à jour d'une seule commande. L'étape map dans cet exemple fait très peu de travail (elle écrit juste des paires clé/valeur), ce qui la maintient bien en dessous de la limite de 1 000 unités par invocation de map. Le regroupement automatique du framework signifie que nous ne gérons pas manuellement la logique de regroupement, et si une commande a, par exemple, 50 lignes à mettre à jour, ces 50 sorties de map sont toutes acheminées vers un seul appel reduce.

Cet exemple de commande client montre comment gérer les **mises à jour en masse au niveau des lignes** avec Map/Reduce et constitue un modèle que vous pouvez adapter à de nombreux besoins similaires de mises à jour en masse : utilisez map pour distribuer les identifiants de ligne ou d'enregistrement, utilisez reduce pour effectuer des mises à jour consolidées par enregistrement ou par groupe.

## Mise à jour en masse des articles d'inventaire (Exemple)

**Cas d'utilisation :** Les mises à jour en masse sur les enregistrements d'articles (par exemple, les articles d'inventaire ou d'autres types d'articles) sont une autre exigence courante. Par exemple, vous pourriez avoir besoin d'ajuster un champ sur des milliers d'enregistrements d'articles d'inventaire – peut-être mettre à jour un champ de prix, une catégorisation, ou recalculer la valeur d'un champ personnalisé pour tous les articles qui répondent à certains critères.

**Bonnes pratiques :** Si chaque mise à jour d'article est indépendante des autres, vous n'aurez peut-être pas besoin d'une étape reduce du tout – vous pouvez faire tout le travail dans l'étape map (ou même directement dans getInputData via un script de mise à jour en masse, mais ici nous nous concentrons sur Map/Reduce). Ignorer l'étape reduce simplifie le script. Cependant, assurez-vous que chaque invocation de map reste dans les limites d'utilisation (ce qui est généralement le cas si vous effectuez une seule opération d'enregistrement par map). Vous pouvez utiliser **`record.submitFields`** ou `record.save()` sur un enregistrement chargé pour appliquer les mises à jour. L'utilisation de `submitFields` est souvent plus efficace pour les changements de valeur de champ simples car elle évite un chargement complet de l'enregistrement.

**Exemple :** Supposons que nous voulions augmenter la quantité en stock de tous les articles d'inventaire de 10 (à des fins de démonstration). Nous pouvons écrire un Map/Reduce qui recherche les articles d'inventaire avec une quantité en stock > 0 (juste comme filtre d'exemple), puis dans l'étape map,

analyser chaque résultat et mettre à jour la quantité de l'article. Dans ce cas, nous n'avons pas réellement besoin d'une étape reduce puisque chaque article est traité individuellement. Nous inclurons une étape summarize pour consigner le résultat :

javascript

Copy

```
/** * @NApiVersion 2.1 * @NScriptType MapReduceScript */ define(['N/search', 'N/record', 'N/log'], (search, record, log) => { const getInputData = () => { // Search for inventory items that meet criteria (e.g., quantityonhand > 0) return search.create({ type: 'inventoryitem', filters: [['quantityonhand', 'greaterthan', 0]], columns: ['internalid', 'quantityonhand'] }); }; const map = (context) => { // Each context.value is a search.Result in string form const result = JSON.parse(context.value); const itemId = result.id; const currentQty = parseInt(result.values.quantityonhand, 10) || 0; const newQty = currentQty + 10; // business logic for new quantity try { // Update the item's quantityonhand field in bulk record.submitFields({ type: 'inventoryitem', id: itemId, values: { quantityonhand: newQty } }); log.debug('Item Updated', `Updated item ${itemId} to new quantity ${newQty}`); } catch (error) { log.error(`Error updating item ${itemId}`, error); // (Errors here will be captured in summarize via the errors iterator) } }; const reduce = (context) => { // Not used in this example (each item handled in map) log.debug('Reduce Stage', `No reduce logic for key: ${context.key}`); }; const summarize = (summary) => { log.audit('Summary', { totalItems: summary.inputSummary.executionCount, totalErrors: summary.inputSummary.errorCount }); // Log any map errors (if any were thrown and not caught in map) summary.mapSummary.errors.iterator().each((key, error, executionNo) => { log.error(`Map Error for item ${key}`, error); return true; }); }; return { getInputData, map, reduce, summarize }; });
```

Dans ce code (utilisant la syntaxe SuiteScript 2.1+ avec des fonctions fléchées), **getInputData** renvoie une recherche enregistrée d'articles d'inventaire dont la quantité en stock est > 0. L'étape **map** analyse chaque résultat et utilise `record.submitFields` pour mettre à jour le champ `quantityonhand` de l'article en ajoutant 10. Nous enveloppons la mise à jour dans un bloc try/catch et consignons les erreurs par article. Nous avons fourni une fonction reduce factice qui ne fait rien (puisque nous n'avons pas besoin de reduce, nous pourrions également omettre entièrement la fonction reduce et ne pas la définir dans le return, mais l'inclure comme une opération nulle pour plus de clarté). L'étape **summarize** consigne un audit avec le nombre total d'articles traités et le nombre d'erreurs. Elle itère également sur toutes les erreurs capturées depuis map (en utilisant `summary.mapSummary.errors`) pour consigner les détails de dépannage.

Cet exemple démontre une mise à jour en masse simple sur les enregistrements d'articles. Nous avons tiré parti du parallélisme de l'étape Map (les mises à jour sur différents articles peuvent s'exécuter sur plusieurs files d'attente simultanément) et de la gestion automatique de la gouvernance par le framework. S'il y avait des milliers d'articles, NetSuite les découperait en instances de map et gérerait la replanification si un job de map atteignait une limite d'utilisation/de temps. Une bonne pratique notable ici est de filtrer la recherche dans getInputData aussi précisément que possible – nous ne récupérons que les articles qui ont réellement besoin de mises à jour, ce qui réduit le traitement inutile. Cela fait partie de l'**"optimisation du filtrage des données"** pour améliorer les performances.

## Mise à jour en masse des enregistrements personnalisés (Exemple)

**Cas d'utilisation :** NetSuite permet la création de types d'enregistrements personnalisés à diverses fins. Vous pourriez avoir un enregistrement personnalisé qui nécessite des mises à jour par lots périodiques – par exemple, un enregistrement personnalisé "Importation de données" qui a des statuts, ou un enregistrement "Actif client" qui nécessite un recalcul régulier d'un champ. Map/Reduce peut être utilisé pour mettre à jour efficacement toutes les instances d'un type d'enregistrement personnalisé.

**Bonnes pratiques :** Déterminez si les mises à jour peuvent être effectuées enregistrement par enregistrement indépendamment ou si elles nécessitent un regroupement. Souvent, pour les enregistrements personnalisés simples, vous pouvez simplement itérer sur chaque enregistrement (comme dans le cas des articles d'inventaire) dans l'étape map. Si les enregistrements personnalisés ont une relation parent-enfant ou nécessitent une agrégation inter-enregistrements, utilisez des clés pour les regrouper (par exemple, regroupez par ID parent ou catégorie dans map, puis traitez dans reduce). Envisagez également d'utiliser SuiteQL ou des recherches enregistrées dans getInputData pour plus de flexibilité – Map/Reduce prend en charge les objets de recherche, et même les résultats de requêtes SuiteQL, comme sources de données d'entrée (SuiteQL peut être utilisé via le module N/query ou une tâche, et il est pris en charge dans Map/Reduce).

**Exemple :** Supposons que vous ayez un type d'enregistrement personnalisé "Exemple de mise à jour en masse" (custrecord_bulk_example) avec des champs incluant une valeur numérique et un statut. Vous souhaitez définir le statut sur "Traité" pour tous les enregistrements où la valeur numérique dépasse un seuil, et peut-être effectuer un calcul sur un autre champ. Un Map/Reduce pourrait faire ceci : getInputData via une recherche enregistrée de tous les enregistrements personnalisés répondant aux critères, mapper chaque enregistrement pour le mettre à jour, et résumer le nombre mis à jour. Puisque chaque mise à jour d'enregistrement personnalisé est indépendante ici, nous pouvons utiliser uniquement map.

Nous ne listerons pas un exemple de code complet par souci de concision (il ressemblerait au cas des articles d'inventaire : recherche d'entrées de customrecord, puis dans map, utilisation de `record.submitFields` ou chargement/modification/enregistrement par enregistrement). Le point clé est

d'utiliser la structure Map/Reduce pour itérer sur tous les enregistrements de ce type. Si nécessaire, vous pouvez incorporer une logique pour gérer les enregistrements liés – par exemple, si plusieurs enregistrements personnalisés sont liés au même parent, vous pourriez utiliser l'ID du parent comme clé pour les réduire ensemble.

**Scénario avancé :** Si une mise à jour d'enregistrement personnalisé doit déclencher ou se coordonner avec un autre processus (par exemple, créer un enregistrement de résumé par client basé sur de nombreux enregistrements personnalisés), le regroupement de Map/Reduce peut être utile. Par exemple, imaginez un enregistrement personnalisé "Transaction PDV" pour les magasins de détail (comme illustré par un exemple dans un article Medium). Vous pourriez regrouper par magasin et date dans l'étape map (clé = magasin+date, valeur = données de vente) et ensuite utiliser reduce pour agréger toutes les transactions PDV pour ce magasin et cette date afin de créer une commande client ou une écriture de journal résumée. Ce modèle a été utilisé pour consolider des milliers d'enregistrements de point de vente en quelques enregistrements de résumé en regroupant les clés dans Map/Reduce.

En résumé, les **enregistrements personnalisés** peuvent être mis à jour en masse à l'aide de Map/Reduce, tout comme les enregistrements standard. Utilisez des recherches enregistrées ou des requêtes pour alimenter getInputData, et choisissez map ou reduce de manière appropriée. Appliquez les mêmes bonnes pratiques : filtrez agressivement les entrées, utilisez le regroupement lorsque cela est bénéfique, et maintenez le travail de chaque invocation à un niveau faible.

Maintenant que nous avons couvert les cas d'utilisation, examinons en détail les étapes du script Map/Reduce avec des extraits de code et expliquons comment chaque étape fonctionne de manière générale.

# Présentation du script Map/Reduce : Étapes et exemples de code

Dans cette section, nous décomposons chaque étape d'un script Map/Reduce et fournissons des exemples de code et des explications pour chacune. Que vous mettiez à jour des commandes client, des articles ou des enregistrements personnalisés, la manière dont vous implémentez ces étapes suit le même modèle.

## Étape getInputData

**Objectif :** L'étape `getInputData` est l'endroit où vous collectez les données à traiter. Elle s'exécute en premier, et une seule fois, lors d'une exécution Map/Reduce. Elle doit renvoyer l'un des éléments suivants : un objet Search (ou Search.ResultSet), un tableau ou une liste de données, ou un objet (par exemple, un objet qui a un itérateur). NetSuite prendra ce que vous renvoyez et l'itérera pour alimenter l'étape map (ou l'étape reduce, si map est ignorée) (Source: docs.oracle.com)(Source: docs.oracle.com).

**Implémentation typique :** Le plus souvent, getInputData renvoie une recherche enregistrée. Vous pouvez créer une recherche avec le module SuiteScript `N/search` et la renvoyer directement – le framework Map/Reduce exécutera alors la recherche en arrière-plan et diffusera automatiquement les résultats dans l'étape map, gérant la pagination pour vous. Par exemple :

```
function getInputData() {
  return search.create({
    type: "customer",
    filters: [["stage", "is", "LEAD"]],
    columns: ["internalid"],
  });
}
```

Cela alimenterait tous les clients de type "Lead" dans l'étape map, un client par invocation de map. Vous pouvez également renvoyer un tableau ou un objet. Par exemple :

```
function getInputData() { return [1, 2, 3, 4, 5];
// an array of IDs or simple values }
```

Dans ce cas trivial, l'étape map s'exécuterait 5 fois, recevant à chaque fois un élément du tableau.

**Astuce – Grands ensembles de résultats :** Si vous utilisez une recherche enregistrée, il est préférable de renvoyer l'objet de recherche (comme indiqué ci-dessus) plutôt que de charger manuellement tous les résultats en une seule fois. NetSuite récupérera les résultats par blocs gérables et cédera automatiquement si la récupération de la recherche est lourde. Si vous faites plutôt quelque chose comme `search.run().getRange({ start:0, end:1000 })`, vous n'obtenez que les 1000 premiers résultats et pourriez avoir besoin de gérer le reste manuellement. Une meilleure approche (pour de très grands ensembles de données) est soit 1) de laisser le framework Map/Reduce gérer la recherche, soit 2) d'utiliser l'API de pagination de recherche (par exemple, `searchPaged`) pour itérer sur tous les résultats. L'exemple de JCurve Solutions note explicitement que si vous n'obtenez que 1000 résultats dans getInputData, vous devrez implémenter une logique pour obtenir les pages suivantes ou exécuter le script à nouveau. En bref, **Map/Reduce peut gérer nativement la récupération de plus de 1000 résultats de recherche** si vous renvoyez un objet de recherche ; il cédera et reprendra au besoin tout en extrayant les résultats par lots en arrière-plan.

**Sortie :** La fonction getInputData doit produire des données sous une forme qui peut être transformée en paires clé/valeur. Si vous renvoyez une recherche ou un tableau d'objets de résultats, le framework Map/Reduce traitera automatiquement chaque résultat comme une entrée `map` distincte. Si vous renvoyez un tableau simple de valeurs, chaque valeur est passée à map mais vous devrez attribuer une clé dans map si reduce doit être utilisé.

## Étape Map

**Objectif :** Le rôle de l'étape `map` est de prendre chaque entrée et de produire une sortie indexée par quelque chose. NetSuite appellera votre fonction `map(context)` pour chaque élément de getInputData (ou chaque résultat de recherche). Dans l'étape map, vous **analysez généralement l'entrée, effectuez une opération atomique, puis utilisez `context.write(key, value)` pour émettre une paire clé/valeur** pour l'étape de brassage (shuffle). Si vous prévoyez d'utiliser une étape reduce, la clé est ce qui détermine le regroupement. Si vous n'utilisez pas reduce, vous n'aurez peut-être rien à écrire du tout (ou vous écrivez avec une clé qui sera simplement transmise à summarize).

**Contenu du `context` de map :** L'objet `mapContext` passé à map a des propriétés comme `context.key` et `context.value`. Pour les entrées provenant d'un résultat de recherche ou d'un tableau, souvent `context.key` peut être un index ou un ID de résultat de recherche et `context.value` est la donnée (sous forme de chaîne JSON). Par exemple, dans SuiteScript 2.x, lorsqu'un search.Result est passé, `context.value` sera une chaîne JSON de l'objet résultat. La première chose que font de nombreuses fonctions map est donc `const result = JSON.parse(context.value)` pour obtenir un objet utilisable.

**Traitement dans map :** Une fois que vous avez les données, vous pouvez effectuer votre mise à jour d'enregistrement ou toute autre logique ici *si aucun autre regroupement n'est nécessaire*. Par exemple, dans l'exemple des articles d'inventaire, l'étape map elle-même a effectué la mise à jour `record.submitFields` pour chaque article et a consigné le succès ou l'erreur. Dans ce cas, nous n'avons pas vraiment émis de paire clé/valeur pour un reduce ; nous avons simplement traité et (implicitement) le framework ne passe rien à summarize, sauf le nombre de résultats et les erreurs éventuelles. C'est une utilisation **map-only**.

Cependant, si vous avez besoin d'une étape reduce, votre fonction map doit utiliser `context.write()`. Les arguments peuvent être `context.write(key, value)` ou un objet `{ key: k, value: v }`. La clé et la valeur peuvent être des chaînes ou seront sérialisées en chaîne. Par exemple, dans le script de lignes de commande client ci-dessus, nous avons fait :

```
context.write({ key: salesOrderId, value: lineKey });
```

Cela émet un résultat intermédiaire. NetSuite collecte toutes ces sorties, et dans l'étape de brassage, il regroupera par `salesOrderId`. **Important :** Si vous ignorez l'appel à `write` dans map, et que vous avez une étape reduce définie, alors reduce ne recevra rien (elle ne s'exécutera même pas). Vous utilisez donc `write` pour transmettre les données. Si votre map ne fait que filtrer ou transformer des données pour reduce, assurez-vous que chaque élément de données pertinent est écrit avec une clé appropriée.

**Parallélisme et ordonnancement :** Les fonctions map peuvent s'exécuter simultanément dans des processus distincts. Il n'y a pas d'ordre garanti dans lequel les exécutions de map se terminent les unes par rapport aux autres – ce qui est généralement bien car chacune gère une entrée indépendante. N'essayez pas de modifier des variables globales partagées à partir de fonctions map en vous attendant à un résultat déterministe, car les maps pourraient se chevaucher dans le temps. Si vous devez accumuler des données à travers toutes les maps, c'est à cela que servent les étapes reduce ou summarize.

**Gestion des erreurs dans map :** Si une erreur est levée dans une fonction map et n'est pas interceptée, NetSuite l'interceptera et signalera cette "entrée" de map comme ayant échoué, mais *cela n'arrêtera pas l'ensemble du job Map/Reduce*. Ces erreurs sont disponibles plus tard via le résumé (nous discuterons de la gestion des erreurs plus tard). Si vous souhaitez gérer les erreurs dans map (par exemple, les consigner et continuer), utilisez try/catch autour des opérations problématiques comme montré dans l'étape map de l'exemple d'article. Généralement, laissez les erreurs non gérées remonter et traitez-les dans summarize, sauf si vous avez une action de récupération spécifique à effectuer immédiatement.

## Étape Reduce

**Objectif :** L'étape `reduce` s'exécute si vous avez appelé `context.write()` dans map ou si getInputData a directement fourni des données clé/valeur et que vous avez défini un reduce. Le framework Map/Reduce passe chaque clé unique avec toutes ses valeurs à une invocation `reduce(context)` distincte (Source: netsuitedocumentation1.gitlab.io). L'étape reduce est l'endroit où vous traitez les données agrégées. Les modèles courants dans reduce : mettre à jour un enregistrement qui sert de "parent" ou de résumé pour le groupe, effectuer des calculs sur le groupe de valeurs, ou peut-être effectuer un appel sortant consolidé ou une écriture de fichier pour le groupe.

**Contenu du `context` de reduce :** Le `reduceContext` contient `context.key` (la clé) et `context.values` (un tableau de toutes les valeurs associées à cette clé). Notez que les valeurs sont des chaînes (sérialisées), vous itérez donc souvent sur `context.values` et `JSON.parse()` chaque élément pour obtenir des objets, s'ils étaient complexes. Dans de nombreux cas, les valeurs peuvent être de simples identifiants ou des nombres. Par exemple, dans l'exemple de ligne de commande client, `context.key` était un ID interne de commande client (sous forme de chaîne), et `context.values` était un tableau de chaînes de clés uniques de ligne. Dans la fonction reduce, nous les avons converties en mises à jour sur la commande client chargée.

**Conception de la logique reduce :** Parce que reduce est appelé par groupe, c'est un endroit naturel pour effectuer des opérations **une fois par groupe d'enregistrements**. C'est pourquoi, dans cet exemple, nous chargeons la commande client une fois dans reduce et mettons à jour toutes les lignes – une seule sauvegarde, par opposition à plusieurs sauvegardes dans map qui seraient inefficaces. Un autre exemple : si les clés sont des ID de client et les valeurs sont des factures, dans reduce, vous pourriez créer un seul enregistrement de paiement consolidé pour ce client en utilisant toutes les valeurs de facture. Ou si les clés sont un ID d'un "lot" personnalisé et les valeurs sont des points de données, vous pourriez générer un fichier par lot dans reduce.

Une stratégie subtile : Si vous constatez que chaque "groupe" n'a qu'une seule valeur (c'est-à-dire que les clés étaient uniques pour chaque enregistrement), alors vous n'aviez pas réellement besoin d'un reduce – vous auriez pu le faire dans map. Cependant, certains développeurs choisissent de tout produire vers une seule clé pour forcer un seul reduce avec un tableau de *toutes* les valeurs. C'est rarement nécessaire, mais c'est possible (par exemple, tout clé à `"ALL"` et ensuite reduce obtient une clé "ALL" avec un tableau de toutes les valeurs – transformant effectivement reduce en une phase à thread unique qui a accès à l'ensemble des données). Généralement, cependant, si vous devez opérer sur chaque enregistrement individuellement, faites-le dans map ; utilisez reduce uniquement si vous avez besoin de regroupement.

**Gouvernance dans reduce :** Chaque invocation de reduce peut utiliser jusqu'à 5 000 unités d'utilisation d'API et 15 minutes de temps d'exécution par défaut. C'est plus généreux que les 1 000 unités/5 minutes de map. Par conséquent, reduce est adapté aux tâches plus lourdes par groupe. Par exemple, si le traitement d'une clé nécessite le chargement et la mise à jour de plusieurs enregistrements (comme notre exemple de chargement d'une commande et de mise à jour de nombreuses lignes, ce qui pourrait consommer quelques centaines d'unités), il devrait rester en dessous de 5 000 unités. Si un seul groupe pouvait être extrêmement grand (par exemple, votre clé est "ALL" et vous avez 100 000 enregistrements dans un seul reduce), vous risquez d'atteindre la limite d'utilisation. Dans de tels cas, envisagez de repenser le regroupement des clés pour distribuer le travail plus uniformément, ou de gérer des tableaux partiels et d'utiliser la phase summarize pour traiter le reste.

**Pas de map, seulement reduce :** Il est bon de rappeler que vous pouvez ignorer la phase map et n'avoir que getInputData et reduce. Dans ce scénario, ce que vous retournez dans getInputData doit être un objet qui peut être itéré comme des paires clé/valeur. Généralement, si la phase map est ignorée, NetSuite traitera les *résultats de getInputData* comme s'ils étaient déjà des paires clé/valeur. Par exemple, si getInputData renvoie une recherche, par défaut, la clé de chaque résultat de recherche est l'ID de l'enregistrement et la valeur est l'objet résultat (Source: docs.oracle.com)(Source: docs.oracle.com). Ceux ayant le même ID seraient regroupés (généralement, les résultats de recherche ont des ID uniques, sauf si vous effectuez délibérément une recherche agrégée). Ou si getInputData renvoie un tableau d'objets comme `[{ key: 'A', value: 1}, { key: 'A', value: 2}, { key: 'B',`

`value: 3}]`, le framework pourrait regrouper par ces clés pour la phase reduce. En pratique, l'utilisation de map pour préparer les clés est plus simple, mais une utilisation avancée pourrait ignorer map par souci de simplicité ou pour tirer parti de la limite d'utilisation plus élevée. Marty Zigman (Prolecto) note que lorsqu'une seule fonction est nécessaire, opter pour reduce peut être bénéfique car *« la fonction reduce offre une plus grande capacité de gouvernance »* pour chaque point de données.

**Récapitulatif des exemples :** Dans le scénario précédent des **lignes de commande client**, reduce a chargé la commande et mis à jour les lignes. Dans le scénario de **consolidation des points de vente** de l'exemple Medium, reduce a pris chaque regroupement magasin+article et a additionné les quantités, puis a créé une seule commande client par groupe. Cela illustre comment reduce est utilisé pour consolider et finaliser les mises à jour groupées.

## Phase Summarize

**Objectif :** La phase `summarize(summaryContext)` s'exécute après l'achèvement de toutes les exécutions de map (et reduce). Elle vous donne une vue d'ensemble de ce qui s'est passé : combien d'entrées ont été traitées, combien d'erreurs se sont produites, et elle fournit des références à la sortie de la phase reduce (le cas échéant) et aux itérateurs d'erreurs pour map/reduce. La phase summarize est **idéale pour enregistrer les résultats, envoyer des notifications ou effectuer toute tâche de nettoyage finale** qui doit avoir lieu une fois par exécution de script.

**Accès aux informations de résumé :** L'objet `summaryContext` passé en paramètre possède plusieurs propriétés utiles :

- `summaryContext.inputSummary` – informations sur la phase d'entrée (par exemple, si getInputData a levé une erreur, elle serait ici ; combien d'éléments ont été lus, etc.).
- `summaryContext.mapSummary` – informations sur la phase map, y compris une méthode `summaryContext.mapSummary.errors` qui est un itérateur de toutes les erreurs levées par map (qui n'ont pas été interceptées dans map).
- `summaryContext.reduceSummary` – informations similaires pour la phase reduce, y compris l'itérateur `reduceSummary.errors` pour les erreurs de reduce.
- `summaryContext.output` – si votre phase reduce ou map a écrit des résultats en utilisant `context.write()` *et* que vous ne les avez pas consommés dans reduce (par exemple, si aucune phase reduce n'est définie, les écritures de map vont au résumé ; ou si reduce appelle `context.write()`, ces sorties apparaissent dans summaryContext.output). C'est moins couramment utilisé – généralement, on ne produit pas de sortie depuis reduce, sauf pour générer un jeu de données ou un fichier final.

**Journalisation des erreurs :** Un modèle courant dans summarize est d'itérer sur les itérateurs d'erreurs pour les journaliser ou prendre des mesures. Par exemple, le Centre d'aide de NetSuite suggère un code comme :

```
summary.mapSummary.errors.iterator().each(function (key, error, executionNo) {
  var errorMsg = JSON.parse(error).message;
  log.error(
    "Map error for key: " + key + " on attempt " + executionNo,
    errorMsg,
  );
  return true;
});
```

Cela parcourra toutes les erreurs de map (chaque `error` est une chaîne JSON représentant l'erreur levée). De même pour les erreurs de reduce. Dans notre exemple de résumé d'article, nous avons démontré l'itération de `summary.mapSummary.errors`. Si une erreur s'est produite pour un enregistrement spécifique dans map, vous verriez sa clé (qui pourrait être l'ID de l'enregistrement) et les détails de l'erreur ici. C'est crucial car les erreurs map/reduce n'apparaissent pas toujours dans les journaux de script, sauf si vous les affichez dans summarize (par conception, les erreurs non gérées sont capturées et conservées pour le résumé). Comme le mentionne un point du forum SuiteAnswers : *« Les erreurs non gérées dans les points d'entrée map/reduce ne sont pas journalisées. Pour les voir, vous devez itérer l'itérateur d'erreurs de résumé. »*. Donc, pour des scripts robustes, implémentez la journalisation des erreurs dans summarize.

**Collecte des résultats :** Si votre script produit une sortie finale (comme la création d'enregistrements ou l'écriture dans un fichier), summarize est un bon endroit pour, par exemple, journaliser le nombre d'enregistrements créés, ou pour envoyer un e-mail avec un fichier joint si vous en avez généré un. Vous pouvez également accéder aux totaux de temps et d'utilisation si nécessaire (bien que les propriétés spécifiques pour l'utilisation totale puissent ne pas être directement exposées, vous pourriez les calculer à partir des informations de la phase).

**Exemple d'utilisation dans summarize :** Dans notre exemple d'article d'inventaire, nous avons enregistré le nombre total d'articles traités et d'erreurs. Nous pourrions améliorer cela pour envoyer un e-mail à un administrateur si `summary.inputSummary.errorCount > 0`, par exemple, en joignant un fichier CSV des ID d'enregistrements échoués que nous avons compilés dans l'itérateur.

**Nettoyage ou étapes suivantes :** Summarize peut également déclencher une logique de suivi. Par exemple, peut-être qu'après le traitement, vous souhaitez lancer un autre script – vous pourriez soumettre un script planifié (Scheduled Script) ou un autre Map/Reduce depuis summarize via le module

`N/task` si nécessaire (attention toutefois à ne pas créer de boucles infinies de replanification).

Globalement, summarize est votre opportunité de finaliser le travail par lots : considérez-le comme la phase de « reporting » du Map/Reduce.

Maintenant que nous avons parcouru chaque phase avec des exemples, concentrons-nous sur les considérations de performance et sur la façon d'écrire des scripts Map/Reduce qui s'exécutent efficacement dans les limites de gouvernance de NetSuite.

## Optimisation des performances et stratégies de gouvernance

L'écriture d'un script Map/Reduce pour les mises à jour en masse nécessite une attention particulière aux performances et à la gouvernance (le terme de NetSuite pour les limites d'utilisation des ressources). Voici les **meilleures pratiques et astuces d'optimisation** pour garantir que vos tâches Map/Reduce s'exécutent aussi efficacement que possible :

1. **Optimiser le filtrage des données (minimiser la taille d'entrée) :** Limitez les données que vous récupérez dans `getInputData` à ce qui est strictement nécessaire. Utilisez des filtres de recherche spécifiques afin de ne pas traiter des enregistrements qui n'ont pas besoin d'être mis à jour. Par exemple, si vous n'avez besoin de mettre à jour que les enregistrements modifiés au cours de la dernière semaine, incluez un filtre de date. Cela réduit le nombre d'exécutions de map (ou reduce) et économise beaucoup de temps de traitement. Essentiellement, effectuez le gros du travail dans la requête (base de données) plutôt que dans la logique du script lorsque cela est possible.

2. **Utiliser des opérations efficaces sur chaque enregistrement :** Dans map ou reduce, privilégiez les API légères. Par exemple :

    - Utilisez `record.submitFields` pour les mises à jour d'un seul champ ou de quelques champs au lieu de charger et d'enregistrer des enregistrements complets, afin de réduire les unités d'utilisation (submitFields est une opération unique).

    - Si vous chargez un enregistrement, essayez d'effectuer toutes les modifications nécessaires en un seul chargement et une seule sauvegarde (comme nous l'avons fait dans l'exemple de reduce mettant à jour plusieurs lignes). Évitez les appels de sauvegarde multiples pour le même enregistrement.

    - Tirez parti des résultats de recherche pour obtenir des données au lieu de charger chaque enregistrement juste pour lire un champ. La phase map peut souvent obtenir les valeurs nécessaires à partir de `context.value` (qui contient les données du résultat de recherche) sans

autre recherche, comme nous l'avons montré en analysant la quantité actuelle de l'article à partir du JSON du résultat de recherche.

- Regroupez vos mises à jour : par exemple, mettre à jour 10 lignes dans une seule commande client avec une seule sauvegarde (dans reduce) est bien préférable à la mise à jour d'une ligne à la fois avec 10 sauvegardes d'enregistrements distinctes.

3. **Tirer parti des cessions de gouvernance de NetSuite :** Le framework Map/Reduce cédera et replanifiera automatiquement les *tâches* map ou reduce si elles approchent certaines limites (appelées **limites souples**). Par exemple, il existe une limite souple de 10 000 unités par *tâche* map ou reduce (non pas par invocation, mais sur l'ensemble de la tâche de la phase) où NetSuite cédera si elle est dépassée. Vous n'avez généralement pas besoin d'appeler manuellement `yield()` au sein d'une fonction map ou reduce – et en fait, NetSuite ne fournit même pas d'API yield explicite en 2.x comme l'ancien `nlapiYieldScript()` pour les scripts planifiés 1.0. Concentrez-vous plutôt sur le maintien de chaque invocation map/reduce en dessous des limites strictes (1000/5000 unités, etc.). Le framework gérera la cession entre les invocations. Cela dit, si vous avez une boucle dans une phase reduce qui pourrait s'exécuter extrêmement longtemps (comme le traitement de 10 000 entrées dans un seul reduce), vous pourriez volontairement la diviser en produisant des résultats intermédiaires et en les laissant se traiter dans une invocation reduce ultérieure ou dans summarize. Mais généralement, faites confiance à la gestion de la gouvernance du framework. (Dans SuiteScript 2.x, `yield` est principalement géré pour vous. Le blog Heliverse mentionnait toujours l'utilisation de `nlapiYieldScript()` comme astuce, mais c'est une approche 1.0 ; dans Map/Reduce 2.x, la cession automatique est l'un des avantages.)

4. **Connaissance des limites de gouvernance :** Connaissez les limites strictes afin de pouvoir concevoir vos scripts en conséquence. Les principales :

   - 1 000 unités et ~5 minutes de CPU par invocation map.
   - 5 000 unités et ~15 minutes de CPU par invocation reduce.
   - 10 000 unités et ~60 minutes de CPU pour getInputData et summarize.
   - 200 Mo de « données persistantes » (données clé/valeur en transit) à tout moment – si vous essayez d'écrire un objet extrêmement grand avec `context.write()` ou si un grand nombre de clés sont en attente, vous pouvez atteindre cette limite. Cela est rarement rencontré, sauf si vous écrivez des données massives dans map sans les réduire.
   - Si une seule invocation atteint ces limites, elle lèvera une erreur (comme SSS_USAGE_LIMIT_EXCEEDED). La réponse du moteur Map/Reduce dépend de la phase : getInputData atteignant une limite arrêtera le script et passera à summarize ; map atteignant la limite de 1000 unités arrêtera cette map, ignorera cet enregistrement (ou réessayera selon les

paramètres) et continuera les autres ; reduce atteignant 5000 unités similaire. Nous discuterons de la gestion des erreurs ensuite, mais soyez simplement attentif : concevez chaque map pour qu'elle soit petite et chaque reduce pour qu'elle soit aussi petite que nécessaire.

5. **Logique de traitement par lots et de découpage :** Si vous avez le contrôle sur la façon de découper le travail, faites-le. Par exemple, si vous mettez à jour 10 000 enregistrements, il pourrait être acceptable d'avoir 10 000 entrées map. Mais si chaque mise à jour d'enregistrement est lourde, vous pourriez plutôt concevoir getInputData pour qu'il renvoie 100 « lots » (peut-être en utilisant une recherche enregistrée qui regroupe les enregistrements en lots via certains critères ou une liste personnalisée d'ID) afin de n'avoir que 100 appels reduce qui traitent chacun 100 enregistrements. C'est une forme de traitement par lots manuel. Une autre approche consiste, au sein de reduce, si vous avez un tableau énorme, à traiter un sous-ensemble, puis à `context.write()` le reste à vous-même avec la même clé ou une autre clé pour forcer des cycles reduce supplémentaires (modèle avancé). Dans la plupart des cas, le regroupement automatique par clés est suffisant – par exemple, le regroupement par commande client pour traiter les mises à jour de lignes par lots, comme nous l'avons fait, est un exemple de *traitement par lots des mises à jour par clé*.

6. **Configuration du traitement parallèle :** Comme indiqué, vous pouvez définir la « Concurrence » dans le déploiement du script Map/Reduce. Pour les mises à jour à volume élevé, envisagez d'augmenter cette valeur (la valeur par défaut peut être 1 ou 2). Si votre compte dispose de files d'attente Map/Reduce disponibles (gérées par les SuiteCloud Processors – la plupart des comptes de production autorisent plusieurs processeurs parallèles), définir un nombre plus élevé signifie que davantage de fonctions map ou reduce s'exécuteront simultanément. Cela peut améliorer considérablement le débit pour le traitement lié au CPU. Cependant, méfiez-vous d'atteindre plus rapidement les limites de gouvernance ou de solliciter les ressources système – testez toujours avec des volumes réalistes.

7. **Surveiller et ajuster :** Après le déploiement, surveillez les performances de votre script Map/Reduce. NetSuite fournit une page d'état de déploiement de script où vous pouvez voir combien d'enregistrements ont été traités, combien de temps chaque phase a pris, combien de cessions se sont produites, etc. Utilisez ces données pour ajuster votre approche. Par exemple, si vous constatez qu'un appel reduce traite un bloc énorme (prenant près de 15 minutes), vous pourriez affiner votre stratégie de clé pour le diviser davantage. Si vous constatez que le script passe beaucoup de temps dans getInputData, la recherche pourrait peut-être être optimisée ou vous pourriez utiliser l'indexation (dans NetSuite, certains types de recherche ou champs de formule peuvent être lents).

8. **Journalisation et débogage :** En développement, journalisez abondamment pour comprendre le flux (n'oubliez pas de supprimer ou de réduire la journalisation en production pour des raisons de performance). La journalisation aux points clés (début/fin de chaque phase, décomptes, etc.) peut

aider à identifier les goulots d'étranglement. Cependant, ne journalisez pas à l'intérieur d'une boucle serrée pour des milliers d'enregistrements – cela peut ralentir le script et inonder le journal du script. Au lieu de cela, accumulez des compteurs et journalisez des résumés.

9. **Utiliser l'analyse SuiteScript (si disponible) :** Les comptes NetSuite disposent d'un outil d'analyse SuiteScript qui peut afficher les performances historiques des scripts (par exemple, le nombre de fois où votre Map/Reduce a été exécuté, le temps d'exécution moyen, etc.). Cela peut être utile pour un ajustement à long terme, en particulier pour les processus récurrents planifiés.

En suivant ces meilleures pratiques de performance, vous vous assurez que vos scripts Map/Reduce de mise à jour en masse s'exécutent sans problème dans les limites de NetSuite et se terminent aussi rapidement que possible. Ensuite, nous abordons la gestion des erreurs, la journalisation et la récupération – des aspects cruciaux pour un script de qualité professionnelle qui pourrait rencontrer l'imprévu.

## Gestion des erreurs, journalisation et récupération

Une gestion robuste des erreurs est vitale dans les scripts de mise à jour en masse – vous voulez journaliser les échecs, gérer le traitement partiel et éventuellement réessayer certaines erreurs. Le framework Map/Reduce de NetSuite offre plusieurs fonctionnalités pour vous aider :

1. **Capture d'erreurs intégrée :** Si une erreur non interceptée se produit pendant getInputData, map ou reduce, le framework la **capturera et continuera**, garantissant que le script dans son ensemble ne plante pas immédiatement (sauf dans getInputData, qui, s'il y a une erreur, passera directement à summarize sans exécuter map/reduce). Ces erreurs sont stockées et transmises à la phase summarize via le `summaryContext` comme discuté. Cela signifie que votre script n'échouera pas silencieusement – vous avez la possibilité de les journaliser dans summarize. Par exemple, si 5 enregistrements sur 1000 n'ont pas pu être mis à jour dans map (peut-être en raison de restrictions d'enregistrement ou de problèmes de données), vous pouvez voir ces 5 erreurs dans summarize et peut-être prendre des mesures (comme notifier les administrateurs ou même créer une tâche pour les réessayer).

2. **Journalisation des erreurs dans Summarize :** Comme montré précédemment, utilisez les itérateurs d'erreurs. Pour les erreurs map : `summary.mapSummary.errors`, pour les erreurs reduce : `summary.reduceSummary.errors`. Chaque entrée d'erreur vous donne la clé (par exemple, l'ID de l'enregistrement qui a échoué) et le message/la pile d'erreurs. Vous devriez les journaliser, ou si le volume est élevé, peut-être les écrire dans un fichier ou un enregistrement de journal personnalisé. Cela fournit une piste d'audit de ce qui n'a pas été traité.

3. **Options de réessai et de sécurité intégrée :** NetSuite fournit deux options de configuration importantes dans les scripts Map/Reduce – **`retryCount`** et **`exitOnError`** (Source: docs.oracle.com). Celles-ci sont définies dans l'instruction de retour du script (dans le cadre d'un objet `config`).

    - **`retryCount`** : Cela spécifie le nombre de fois où une fonction map ou reduce doit être réessayée si elle échoue en raison d'une erreur ou d'une interruption. Les valeurs valides sont de 0 à 3. Par défaut (si non défini), si une fonction map ou reduce échoue, elle *ne réessayera pas* cet enregistrement particulier (elle l'ignorera et enregistrera l'erreur). Si vous définissez `retryCount: 2`, par exemple, alors si une invocation map lève une erreur non interceptée, NetSuite tentera d'exécuter à nouveau cette map pour la même clé (jusqu'à 2 réessais). C'est utile si les erreurs peuvent être temporaires (par exemple, un enregistrement était verrouillé, ou un appel réseau transitoire a échoué). Une fois les réessais épuisés, si l'erreur persiste, elle est traitée comme une erreur permanente et journalisée. Notez que `retryCount` affecte également ce qui se passe après un **plantage de serveur** ou un arrêt inattendu – par défaut, après un plantage, NetSuite réessayera automatiquement toutes les clés incomplètes. Avec `retryCount`, il garantira également que les échecs spécifiques causés par des erreurs soient réessayés.

    - **`exitOnError`** : Ce booléen détermine si une erreur fatale doit arrêter l'ensemble du Map/Reduce ou non (Source: docs.oracle.com). Par défaut (`exitOnError: false`), l'erreur d'un enregistrement *n'arrête pas* le script – il continue simplement, comme décrit. Si vous définissez `exitOnError: true`, alors une fois les réessais autorisés (le cas échéant) terminés, une erreur dans map ou reduce entraînera la sortie complète du script de cette phase et le passage à summarize. Essentiellement, `exitOnError:true` est une stratégie de « défaillance rapide » – vous l'utiliseriez si une seule erreur devait annuler l'ensemble de la tâche. Dans les mises à jour en masse, ce n'est généralement pas souhaité (vous préféreriez journaliser les erreurs d'enregistrements individuels et laisser le reste se traiter), donc dans la plupart des cas, `exitOnError` est maintenu à false.

**Exemple de configuration :** Pour les utiliser, vous feriez :

```
return {
  config: { retryCount: 3, exitOnError: false },
  getInputData: getInputData,
  map: map,
  reduce: reduce,
  summarize: summarize,
};
```

Cela réessaierait chaque map/reduce échouant jusqu'à 3 fois et n'arrêterait pas l'ensemble de la tâche en cas d'erreurs.

4. **Idempotence et gestion des doublons :** En raison d'éventuels réessais ou même de redémarrages automatiques après un problème de serveur, il est possible que le même enregistrement soit traité deux fois. NetSuite note qu'après un plantage, le système ne peut pas toujours savoir exactement quelle clé était en cours de traitement, il peut donc retraiter une clé qui était en cours d'exécution lors du plantage. Par conséquent, **concevez votre logique pour qu'elle soit idempotente ou vérifiez les mises à jour antérieures**. Par exemple, si vous définissez un champ « Traité » sur un enregistrement après l'avoir mis à jour, vérifiez ce champ au début du traitement pour l'ignorer s'il est déjà fait. Ou dans l'exemple de point de vente de Medium, ils ont utilisé un champ personnalisé « isSynced » sur l'enregistrement personnalisé et l'ont défini à true une fois traité, de sorte que si le script s'exécute à nouveau, il ne dupliquera pas cette entrée. De même, si votre reduce crée un nouvel enregistrement (par exemple, un paiement ou un résumé), vous voudrez peut-être vous assurer de ne pas en créer deux pour le même groupe en cas de réessai. Vous pourriez gérer cela en recherchant d'abord un enregistrement existant (comme la fonction `Check_SO()` de l'exemple Medium pour voir si une commande client existe pour cette date/emplacement avant d'en créer une nouvelle).

   La documentation de NetSuite suggère d'ajouter une logique pour détecter une exécution redémarrée – par exemple, en utilisant `summary.executionId` ou en maintenant l'état dans un stockage temporaire – mais une méthode plus simple consiste à vérifier les données elles-mêmes pour des signes de traitement. S'assurer que vos mises à jour se terminent complètement ou peuvent être réexécutées sans dommage sur le même enregistrement est essentiel pour une récupération sûre.

5. **Journalisation et notifications :** Utilisez abondamment le module `log` aux endroits appropriés. Utilisez `log.debug` pour des informations détaillées (affichées uniquement lorsque le niveau de journalisation est défini sur debug), et `log.audit` ou `log.error` pour des informations importantes. Dans summarize, après avoir collecté les erreurs, vous pourriez envoyer une notification par e-mail aux administrateurs s'il y a eu des échecs significatifs. Par exemple, si 50 enregistrements n'ont pas pu être mis à jour, vous pourriez envoyer un e-mail avec un fichier CSV joint de ces ID d'enregistrements et des messages d'erreur pour examen manuel. Cela transforme votre script en un processus plus maintenable et surveillé dans un environnement de production.

**6. Test et dépannage :** NetSuite fournit une page « Script Testing » (Test de script) pour les scripts Map/Reduce où vous pouvez déclencher manuellement le script et même le déboguer (avec l'outil SuiteScript Debugger, bien qu'historiquement les scripts SuiteScript 2.1 n'étaient pas déboguables avant l'amélioration de cette fonctionnalité). Lors du développement, testez d'abord avec un petit ensemble de

données. Utilisez la page Map/Reduce Script Status (Statut du script Map/Reduce) pour vérifier si des erreurs se sont produites – elle indiquera le nombre d'erreurs dans map/reduce, etc., que vous pourrez recouper avec vos journaux.

Pour illustrer la gestion des erreurs, considérons ce scénario : Dans notre exemple de mise à jour d'articles d'inventaire, supposons qu'un enregistrement d'article particulier ne puisse pas être mis à jour (peut-être parce que NetSuite interdit la modification directe de `quantityonhand` dans certains contextes). La fonction `record.submitFields` pourrait générer une erreur. Notre script l'a interceptée et a enregistré une erreur pour cet article. Parce que nous l'avons interceptée, le framework Map/Reduce ne voit pas d'erreur non interceptée, il ne la comptera donc pas dans `summary.mapSummary.errors`. Nous avons choisi de la gérer nous-mêmes. Alternativement, si nous ne l'avions pas interceptée, le framework l'aurait interceptée, aurait marqué cet article comme ayant échoué et aurait continué avec les autres. Dans la phase `summarize`, nous itérerions alors sur `mapSummary.errors` et verrions une entrée pour l'ID interne de cet article avec les détails de l'erreur. Nous pourrions alors, par exemple, implémenter une logique dans `summarize` pour marquer cet article dans une liste personnalisée « mises à jour échouées » ou déclencher une autre tentative via un mécanisme différent.

**7. Récupération après échec partiel :** Si votre script traite des centaines d'enregistrements et que quelques-uns échouent, vous avez plusieurs options :

- Les enregistrer simplement dans les logs (et éventuellement une intervention manuelle ultérieure).
- Augmenter `retryCount` pour les retenter automatiquement quelques fois dans la même exécution (souvent, s'il s'agit d'un problème transitoire, cela le résout).
- Pour les échecs persistants, vous pourriez concevoir un autre script Map/Reduce ou Scheduled Script pour les gérer par la suite, ou même appeler `N/task.submit(MapReduceScriptTask)` depuis la phase `summarize` pour réexécuter le script sur les enregistrements échoués. Cependant, la mise en œuvre d'une réexécution automatisée des seules clés ayant échoué pourrait être complexe (vous devriez, par exemple, passer ces ID en tant que paramètres à une nouvelle exécution Map/Reduce).
- Souvent, l'approche la plus simple consiste à enregistrer les erreurs et à alerter, afin qu'un humain puisse résoudre les problèmes de données.

En résumé, Map/Reduce fournit un framework robuste où **l'erreur d'un seul enregistrement ne fera pas échouer l'ensemble du traitement par lots**. En utilisant les options de configuration et le rapport d'erreurs de la phase `summarize`, vous pouvez rendre votre script de mise à jour en masse fiable et plus facile à prendre en charge. Testez toujours les scénarios d'erreur (par exemple, provoquez délibérément un échec sur un enregistrement dans un environnement de test) pour voir comment votre script le gère et assurez-vous que la journalisation/notification répond à vos besoins.

# Différences entre SuiteScript 2.x et 2.1 dans Map/Reduce

SuiteScript 2.1 (introduit à partir de NetSuite 2019.2) est une mise à jour du langage de script pris en charge par NetSuite, apportant des fonctionnalités JavaScript modernes (ES6/ES2018) à SuiteScript. Pour les scripts Map/Reduce, les **capacités fonctionnelles et les API restent les mêmes entre 2.0 et 2.1** – les différences résident dans la syntaxe et les fonctionnalités de langage disponibles. Principales différences et ce qu'elles signifient pour le développement Map/Reduce :

- **Syntaxe JavaScript moderne :** Dans SuiteScript 2.1, vous pouvez utiliser des fonctionnalités ES2018 telles que les fonctions fléchées (`arrow functions`), `let/const` pour les déclarations de variables, les classes, l'opérateur de propagation (`spread operator`), etc. Par exemple, nos exemples de code utilisaient des fonctions fléchées et `const` dans le style 2.1. Dans SuiteScript 2.0, vous écririez `define([...], function(...) { ... return {...} });` avec la syntaxe de fonction traditionnelle, tandis qu'en 2.1 nous avons écrit `define([...], (...) => { ... return {...} });`. Il s'agit en grande partie d'une question de style et de commodité – les fonctions fléchées peuvent rendre le code plus concis. Les deux accomplissent le même résultat fonctionnel.

- **Tag @NApiVersion :** Pour utiliser SuiteScript 2.1, vous devez définir `@NApiVersion 2.1` dans l'en-tête JSDoc du fichier de script. Si un script est marqué 2.0, il s'exécute sous l'ancien moteur JS et ne reconnaîtra pas la syntaxe plus récente. Il existe également une option `@NApiVersion 2.x` qui signifie actuellement généralement « utiliser la dernière version disponible » – mais selon les notes de NetSuite, jusqu'à ce que 2.1 devienne la valeur par défaut, `2.x` pouvait encore correspondre à 2.0 dans certains cas. À partir de 2025, SuiteScript 2.1 est généralement disponible et n'est plus en version bêta, donc `2.x` signifie probablement 2.1+. Néanmoins, il est recommandé de définir explicitement 2.1 si vous souhaitez utiliser ces fonctionnalités.

- **Rétrocompatibilité :** SuiteScript 2.1 est rétrocompatible avec 2.0 – ce qui signifie que vos scripts Map/Reduce 2.0 existants fonctionneront toujours correctement dans un environnement 2.1 sans modifications. Vous pourriez potentiellement simplement changer la version de l'API à 2.1 et la plupart des scripts continueraient de fonctionner (sauf là où d'anciennes différences JS comme la liaison de `this` ou certains objets globaux se comportent différemment). Pour Map/Reduce spécifiquement, la logique des étapes ne change absolument pas.

- **Avantages des nouvelles fonctionnalités de langage :** L'utilisation de `let/const` peut prévenir les problèmes de `hoisting` et rendre la portée des variables plus claire (utile dans les boucles complexes ou les fonctions imbriquées à l'intérieur de votre map/reduce). Les fonctions fléchées n'ont pas leur propre contexte `this`, ce qui n'est généralement pas un problème dans les modules de script mais est utile à savoir. Les littéraux de gabarit (`template literals`, chaînes avec des backticks) peuvent faciliter la journalisation (par exemple, `log.debug('Status', `Processed`

`${count} records`);`). SuiteScript 2.1 prend également en charge les `promises` et `async/await` pour certaines opérations asynchrones. Cependant, notez que les points d'entrée Map/Reduce eux-mêmes ne peuvent pas être des fonctions `async` – le framework s'attend à une exécution synchrone de chaque point d'entrée. Mais à l'intérieur d'un point d'entrée, vous pourriez utiliser des `promises` (par exemple, dans `map`, si vous aviez besoin d'appeler un RESTlet asynchrone via `N/https` avec des `promises`).

- **Modules et importations :** SuiteScript 2.1 utilise toujours principalement la syntaxe AMD `define` pour la cohérence, mais il est plus tolérant avec le chargement des modules. Vous pourriez voir des exemples utilisant une approche légèrement différente pour l'importation de modules (comme l'utilisation de `import ... from 'N/record'` si NetSuite décide d'autoriser la syntaxe des modules ES dans les scripts – à l'heure actuelle, je crois qu'ils exigent toujours le style AMD `define`). La principale différence réside simplement dans la façon dont vous l'écrivez, et non dans les modules existants. Tous les mêmes modules N/* pour Map/Reduce (N/search, N/record, etc.) sont disponibles en 2.1 comme en 2.0.

- **SuiteScript Debugger :** Initialement, le débogage des scripts 2.1 n'était pas pris en charge, mais NetSuite prévoyait de le mettre à jour. D'ici 2025, supposez que vous pouvez déboguer les deux, mais si des problèmes de débogage surviennent, une solution de contournement consiste à exécuter temporairement votre script en version 2.0 pour le débogage puisque la logique est la même (il suffit d'ajuster la syntaxe si nécessaire).

En conclusion, **SuiteScript 2.1 améliore l'expérience du développeur** pour Map/Reduce en permettant une syntaxe et des fonctionnalités JS modernes, mais ne modifie pas le fonctionnement fondamental des scripts Map/Reduce. Lors de l'écriture d'un script Map/Reduce, la décision d'utiliser 2.0 ou 2.1 dépend principalement des préférences de style de codage et de la nécessité d'utiliser les fonctionnalités ES6+. De nombreux développeurs préfèrent 2.1 pour son code plus propre (fonctions fléchées, etc.) et sa pérennité, car SuiteScript continuera d'évoluer avec les nouvelles normes ECMAScript. Pour un développeur NetSuite expérimenté, l'adoption de 2.1 est recommandée, sauf s'il existe une raison spécifique de s'en tenir à 2.0.

*(Remarque : Si votre projet inclut des scripts 2.0 et 2.1, soyez attentif aux différences telles que la gestion de la portée globale ou certaines API comme N/encode qui pourraient se comporter légèrement différemment en interne. Mais pour l'essentiel, la conversion d'un Map/Reduce de 2.0 à 2.1 implique la mise à jour de la syntaxe et des tests.)*

# Conclusion

La mise à jour en masse d'enregistrements dans NetSuite peut être une tâche complexe, mais les scripts Map/Reduce offrent une solution puissante et évolutive. En comprenant l'**architecture Map/Reduce** – son exécution par étapes (`getInputData`, `map`, `shuffle`, `reduce`, `summarize`) et son parallélisme intégré – les développeurs peuvent concevoir des scripts qui traitent efficacement des ensembles de données massifs sans dépasser les délais d'exécution. Nous avons expliqué pourquoi Map/Reduce surpasse souvent les scripts planifiés (`scheduled scripts`) pour les tâches à grande échelle, grâce à la gestion automatique de la gouvernance et à la capacité d'exécuter des tâches en parallèle.

À travers des cas d'utilisation avancés comme les mises à jour en masse de lignes de commandes clients (`Sales Order`) et les modifications d'enregistrements d'articles, nous avons illustré comment appliquer Map/Reduce à des scénarios réels, en mettant l'accent sur des modèles tels que le regroupement par clés pour minimiser le travail en double (par exemple, une commande chargée par `reduce`) et l'utilisation des appels d'API appropriés pour la performance (par exemple, `submitFields` versus des chargements complets d'enregistrements). Chaque étape du flux Map/Reduce joue un rôle, et nous avons fourni des exemples de code et des conseils pour implémenter et optimiser ces étapes dans SuiteScript 2.x.

Les **optimisations de performance** – du filtrage des données d'entrée à l'utilisation du traitement par lots (`batching`), en passant par la configuration de la concurrence – garantissent que votre script de mise à jour en masse s'exécute dans les limites de NetSuite et se termine aussi rapidement que possible. Tout aussi important, une **gestion robuste des erreurs et une journalisation** rendent votre script fiable en production : avec `retryCount` et une journalisation minutieuse dans `summarize`, vous ne perdrez pas la trace des enregistrements qui n'ont pas pu être mis à jour. Au lieu de cela, vous pouvez les intercepter et même les retenter automatiquement, ou du moins alerter quelqu'un avec les détails.

Nous avons également clarifié que **SuiteScript 2.1** apporte la commodité du JS moderne aux scripts Map/Reduce, ce qui peut aider à écrire un code plus propre et plus maintenable, bien que les mécanismes fondamentaux de Map/Reduce restent inchangés.

En pratique, un script Map/Reduce bien écrit peut traiter des dizaines de milliers d'enregistrements (ou plus) en une seule exécution de déploiement, ce qui serait impraticable avec des types de scripts plus simples. En tirant parti de Map/Reduce, les développeurs NetSuite peuvent créer des solutions de mise à jour en masse – qu'il s'agisse de mettre à jour chaque commande client ouverte, de re-tarifer un catalogue entier d'articles ou de traiter de grands ensembles de données d'enregistrements personnalisés – qui sont **évolutives, efficaces et robustes**. N'oubliez jamais de tester avec des données d'échantillon, de surveiller l'exécution du script et d'affiner votre approche en utilisant les meilleures pratiques couvertes. Grâce à ces techniques, les opérations en masse qui semblaient autrefois intimidantes deviennent une autre tâche automatisée dans votre arsenal NetSuite.

**Références :**

1. Centre d'aide NetSuite – *Map/Reduce Script Stages* : Décrit les cinq étapes de Map/Reduce et leur ordre d'exécution.
2. Centre d'aide NetSuite – *SuiteScript 2.x Map/Reduce Script Type Overview* : Explique le but des scripts Map/Reduce, quand les utiliser et leurs avantages par rapport aux scripts planifiés.
3. Centre d'aide NetSuite – *Map/Reduce Governance* : Détaille les limites d'unités d'utilisation et le comportement de `yielding` automatique pour les scripts Map/Reduce.
4. Centre d'aide NetSuite – *Map/Reduce Script Error Handling* : Explique comment Map/Reduce gère les erreurs et les interruptions, y compris les redémarrages de serveur et les erreurs non interceptées.
5. Centre d'aide NetSuite – *Configuration Options for Map/Reduce (retryCount, exitOnError)* : Documentation des paramètres de nouvelle tentative et de sortie en cas d'erreur pour les scripts Map/Reduce.
6. Centre d'aide NetSuite – *Logging Map/Reduce Errors in Summarize* : Exemple montrant comment itérer sur `summary.mapSummary.errors` et enregistrer chaque erreur.
7. Support JCurve Solutions – *Update Lines on Sales Orders using Map/Reduce* : Fournit un exemple de code d'un script Map/Reduce qui met à jour les lignes de commandes clients, démontrant le regroupement par ID de commande.
8. Blog Heliverse – *Mastering Map/Reduce Scripts in SuiteScript 2.1* : Présente les fondamentaux de Map/Reduce et donne un exemple de mise à jour d'articles d'inventaire avec du code SuiteScript 2.1.
9. Blog Heliverse – *Performance Optimization Tips for Map/Reduce* : Liste les meilleures pratiques telles que le filtrage de la source de données, le traitement par lots d'enregistrements et la surveillance des performances.
10. Blog Folio3 – *Scheduled Scripts vs Map/Reduce* : Décrit les différences entre les scripts planifiés et Map/Reduce, en notant les étapes et les limites de gouvernance (par exemple, script planifié 10 000 unités vs étape map 1 000 unités).
11. Article Medium (Wilson Cheng) – *NetSuite Map/Reduce in Retail* : Décrit un cas d'utilisation de consolidation d'enregistrements personnalisés de PDV en enregistrements résumés à l'aide de Map/Reduce, avec une explication claire de chaque étape et l'utilisation de clés pour le regroupement.

12. Stack Overflow – *Map/Reduce vs Mass Update* : Conseils sur le choix de la mise à jour en masse (`Mass Update`) pour les mises à jour ponctuelles via une recherche enregistrée (`saved search`) par rapport à Map/Reduce pour les mises à jour planifiées ou avec des critères complexes.
13. Blog Prolecto (Marty Zigman) – *Map/Reduce for Project Task Updates* : Discute de l'utilisation de Map/Reduce pour mettre à jour les enregistrements de tâches de projet, et notes l'idée d'utiliser uniquement `reduce` pour une plus grande marge de gouvernance.
14. Blog Tvarana – *SuiteScript 2.1 vs 2.0* : Explique les nouvelles fonctionnalités de SuiteScript 2.1 (prise en charge d'ES2018, fonctions fléchées, etc.) et confirme que tous les types de scripts (y compris Map/Reduce) sont pris en charge en 2.1 avec une syntaxe moderne.

---

Étiquettes: netsuite, suitescript, map-reduce, traitement-masse, traitement-donnees, optimisation-performance, gouvernance, gestion-erreurs

---

# À propos de Houseblend

HouseBlend.io is a specialist NetSuite™ consultancy built for organizations that want ERP and integration projects to accelerate growth—not slow it down. Founded in Montréal in 2019, the firm has become a trusted partner for venture-backed scale-ups and global mid-market enterprises that rely on mission-critical data flows across commerce, finance and operations. HouseBlend's mandate is simple: blend proven business process design with deep technical execution so that clients unlock the full potential of NetSuite while maintaining the agility that first made them successful.

Much of that momentum comes from founder and Managing Partner **Nicolas Bean**, a former Olympic-level athlete and 15-year NetSuite veteran. Bean holds a bachelor's degree in Industrial Engineering from École Polytechnique de Montréal and is triple-certified as a NetSuite ERP Consultant, Administrator and SuiteAnalytics User. His résumé includes four end-to-end corporate turnarounds—two of them M&A exits—giving him a rare ability to translate boardroom strategy into line-of-business realities. Clients frequently cite his direct, "coach-style" leadership for keeping programs on time, on budget and firmly aligned to ROI.

**End-to-end NetSuite delivery.** HouseBlend's core practice covers the full ERP life-cycle: readiness assessments, Solution Design Documents, agile implementation sprints, remediation of legacy customisations, data migration, user training and post-go-live hyper-care. Integration work is conducted by in-house developers certified on SuiteScript, SuiteTalk and RESTlets, ensuring that Shopify, Amazon, Salesforce, HubSpot and more than 100 other SaaS endpoints exchange data with NetSuite in real time. The goal is a single source of truth that collapses manual reconciliation and unlocks enterprise-wide analytics.

**Managed Application Services (MAS).** Once live, clients can outsource day-to-day NetSuite and Celigo® administration to HouseBlend's MAS pod. The service delivers proactive monitoring, release-cycle regression testing, dashboard and report tuning, and 24 × 5 functional support—at a predictable monthly rate. By combining fractional architects with on-demand developers, MAS gives CFOs a scalable alternative to hiring an internal team,

while guaranteeing that new NetSuite features (e.g., OAuth 2.0, AI-driven insights) are adopted securely and on schedule.

**Vertical focus on digital-first brands.** Although HouseBlend is platform-agnostic, the firm has carved out a reputation among e-commerce operators who run omnichannel storefronts on Shopify, BigCommerce or Amazon FBA. For these clients, the team frequently layers Celigo's iPaaS connectors onto NetSuite to automate fulfilment, 3PL inventory sync and revenue recognition—removing the swivel-chair work that throttles scale. An in-house R&D group also publishes "blend recipes" via the company blog, sharing optimisation playbooks and KPIs that cut time-to-value for repeatable use-cases.

**Methodology and culture.** Projects follow a "many touch-points, zero surprises" cadence: weekly executive stand-ups, sprint demos every ten business days, and a living RAID log that keeps risk, assumptions, issues and dependencies transparent to all stakeholders. Internally, consultants pursue ongoing certification tracks and pair with senior architects in a deliberate mentorship model that sustains institutional knowledge. The result is a delivery organisation that can flex from tactical quick-wins to multi-year transformation roadmaps without compromising quality.

**Why it matters.** In a market where ERP initiatives have historically been synonymous with cost overruns, HouseBlend is reframing NetSuite as a growth asset. Whether preparing a VC-backed retailer for its next funding round or rationalising processes after acquisition, the firm delivers the technical depth, operational discipline and business empathy required to make complex integrations invisible—and powerful—for the people who depend on them every day.

---