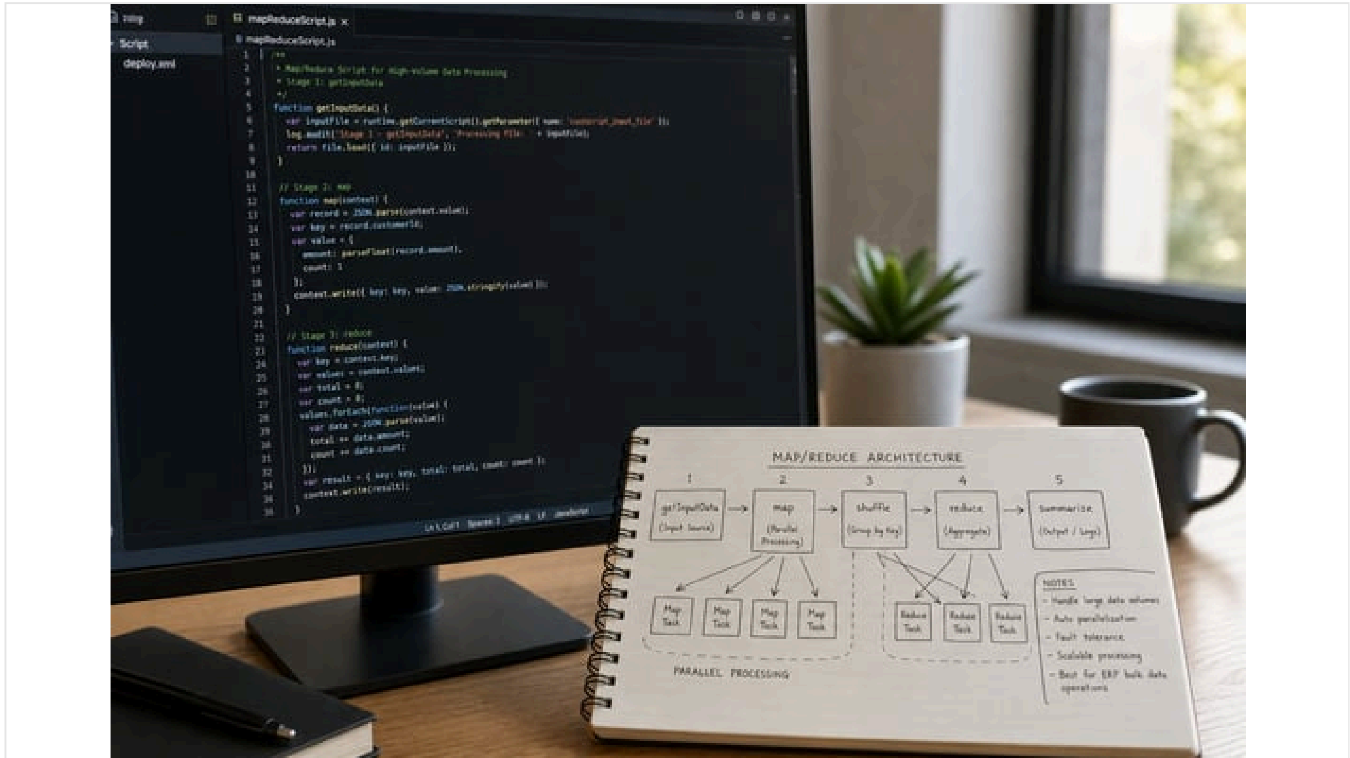
Guide du SuiteScript Map/Reduce de NetSuite : Étapes et Gouvernance

Publié le 27 mai 2026 38 min de lecture



Résumé analytique

Le framework Map/Reduce de SuiteScript de NetSuite est un outil puissant pour le traitement de données parallèle à grande échelle au sein de la plateforme ERP NetSuite. Introduit dans SuiteScript 2.x, il permet aux développeurs de « diviser les données en petites parties indépendantes » que le système peut traiter **en parallèle** (Source: docs.oracle.com). Contrairement aux scripts planifiés (Scheduled Scripts) traditionnels de SuiteScript (qui s'exécutent de manière séquentielle), les scripts Map/Reduce génèrent plusieurs « tâches » (jobs) qui s'exécutent simultanément sur les processeurs SuiteCloud de NetSuite (Source: docs.oracle.com) (Source: www.houseblend.io). Cela signifie qu'un ensemble de données composé de milliers d'enregistrements peut être partitionné entre plusieurs tâches Map/Reduce, améliorant ainsi considérablement le débit. Parallèlement, NetSuite impose des limites de gouvernance strictes (unités d'utilisation et temps d'exécution) sur chaque invocation de fonction. Le framework Map/Reduce intègre une logique de **cédage automatique et de nouvelle tentative (yielding and retry)** : lorsqu'une tâche approche de ses limites de gouvernance, elle se met en pause (cède) et reprend sous la forme d'une nouvelle instance de tâche (Source: www.houseblend.io) (Source: docs.oracle.com).

En pratique, les scripts Map/Reduce sont idéaux pour les opérations de masse à haut volume et les agrégations complexes. Par exemple, un cas d'utilisation pourrait consister à charger toutes les factures ouvertes (getInputData), mapper chaque facture à son client (map), regrouper par ID client (shuffle/reduce), créer des paiements consolidés par client (reduce), puis envoyer un rapport récapitulatif (summarize) (Source: docs.oracle.com). Ce traitement par étapes contraste avec une approche monolithique de type « tout charger et boucler ». En tirant parti de la sortie clé-valeur de `context.write()`, le framework Map/Reduce gère automatiquement le regroupement (shuffle), de sorte que des tâches telles que « mettre à jour toutes les lignes d'articles par commande client » peuvent être effectuées en mappant chaque ligne à sa commande, puis en réduisant à un seul chargement par commande (Source: www.houseblend.io) (Source: www.houseblend.io).

Ce rapport propose une exploration approfondie de SuiteScript Map/Reduce de NetSuite. Nous abordons l'**architecture et les étapes** d'exécution, les **limites de gouvernance** et leur impact sur la conception, ainsi que les **modèles de traitement de masse** courants et les meilleures pratiques. Nous comparons Map/Reduce avec d'autres approches (scripts planifiés, mises à jour de masse, Suitelets) et examinons les performances et le débit avec des preuves issues de benchmarks. Plusieurs perspectives sont incluses : documentation officielle d'Oracle, avis d'experts/blogs tiers et exemples

illustratifs. Nous discutons également de modèles réels pour les mises à jour d'enregistrements en masse (factures, commandes client, articles d'inventaire, enregistrements personnalisés), présentons des structures de code et fournissons des données sur les taux de traitement. Enfin, nous examinons les implications des fonctionnalités plus récentes de SuiteScript (par exemple, améliorations asynchrones 2.1 et les orientations futures pour le traitement à haut volume dans NetSuite.

Introduction et contexte

NetSuite et SuiteScript

Oracle NetSuite est une plateforme ERP cloud de premier plan utilisée par des milliers d'organisations. Elle fournit des applications métier multi-locataires couvrant la finance, le CRM, l'inventaire, etc. Les organisations ont souvent besoin de personnaliser NetSuite : pour intégrer des données provenant de systèmes externes, appliquer une logique métier personnalisée ou traiter des transactions en masse. L'API **SuiteScript** de NetSuite permet aux développeurs d'écrire du code JavaScript qui s'exécute sur le serveur (pour le traitement des enregistrements) ou sur le client (pour la personnalisation de l'interface utilisateur) (Source: docs.oracle.com). À l'ère de SuiteScript 1.0 (jusqu'en 2013) et au début de SuiteScript 2.x, les développeurs utilisaient généralement des *scripts planifiés* ou des scripts de *mise à jour de masse* pour gérer le traitement par lots. Cependant, avec l'augmentation des volumes de données, ces approches présentaient des limites : une seule invocation planifiée est plafonnée à 10 000 unités d'utilisation et peut nécessiter un cédage manuel complexe ou plusieurs exécutions, et la mise à jour de masse (fonctionnalité de l'interface utilisateur) ne peut pas gérer une logique très complexe (Source: www.houseblend.io) (Source: www.houseblend.io).

Reconnaissant le besoin d'un traitement parallèle évolutif, NetSuite a introduit le type de **script Map/Reduce** dans le cadre de la plateforme SuiteScript 2.x (vers 2017). Comme le décrit Oracle, « les scripts Map/Reduce sont parfaits pour appliquer la même logique à plusieurs objets, un par un » et excellent lorsque vous pouvez *« diviser vos données en petites parties indépendantes »* (Source: docs.oracle.com). En effet, Map/Reduce dans NetSuite adapte le paradigme classique MapReduce (issu des systèmes de Big Data) au scripting ERP : le système gère le fractionnement, le regroupement et l'exécution parallèle, permettant aux développeurs de se concentrer sur la logique métier par enregistrement ou par groupe.

La documentation officielle de NetSuite souligne que les scripts Map/Reduce exploitent les **processeurs SuiteCloud** en arrière-plan : la plateforme crée automatiquement *plusieurs tâches* pour exécuter votre script, traitant les tâches en parallèle sur des « processeurs » CPU (Source: docs.oracle.com). Les scripts Map/Reduce présentent plusieurs avantages par rapport aux scripts planifiés : parallélisme intégré et gestion automatique des limites de gouvernance (avec cédage et reprogrammation) (Source: docs.oracle.com) (Source: www.houseblend.io). D'un autre côté, les développeurs doivent structurer leur logique en fonctions discrètes (étapes) et s'assurer que chaque unité de travail est relativement petite (les limites par invocation s'appliquent) (Source: docs.oracle.com) (Source: docs.oracle.com).

Objectif et portée

Ce rapport sert de guide complet sur SuiteScript Map/Reduce de NetSuite. Nous allons :

- Décrire l'**architecture Map/Reduce** et chaque étape (getInputData, map, shuffle, reduce, summarize), y compris la manière dont les données circulent entre elles et la différence entre exécution concurrente et séquentielle (Source: docs.oracle.com) (Source: docs.oracle.com).
- Détailler les **limites de gouvernance** et leur application (unités d'utilisation par invocation, limites de temps, limites de données persistantes) (Source: docs.oracle.com) (Source: docs.oracle.com), y compris les limites strictes vs souples et la manière dont le framework y répond (erreurs vs cédage).
- Présenter les **meilleures pratiques** et les modèles de traitement de masse pour la mise à jour de grands ensembles d'enregistrements. Cela inclut la logique de regroupement (utilisation de clés dans l'étape map afin qu'une seule réduction charge chaque enregistrement une fois) (Source: www.houseblend.io) (Source: www.houseblend.io), l'utilisation de recherches efficaces ou de SuiteQL pour l'entrée de données, la mise en cache et l'utilisation de l'étape summarize pour la journalisation des résultats.
- Présenter des **données et des benchmarks** illustrant les performances. Nous incluons des mesures communautaires du débit (par exemple, seulement quelques dizaines d'enregistrements par seconde dans certains tests (Source: ursuscode.com) (Source: www.houseblend.io) et discutons des facteurs qui affectent ces chiffres.
- Comparer Map/Reduce aux alternatives (script planifié, mise à jour de masse, Suitelet) en termes de cas d'utilisation et de performances (Source: www.houseblend.io) (Source: www.houseblend.io).
- Fournir des **exemples d'études de cas** tirés de blogs communautaires (par exemple, mises à jour en masse de commandes client, mises à jour d'articles d'inventaire) pour illustrer les modèles d'utilisation réels (Source: www.houseblend.io) (Source: www.houseblend.io).

- Discuter des **implications et des orientations futures**, en particulier avec les nouvelles fonctionnalités de SuiteScript (2.1 async/await, streaming de fichiers plats, etc.) et les besoins évolutifs de l'écosystème NetSuite (Source: www.houseblend.io) (Source: docs.oracle.com).

Tout au long du document, les affirmations sont étayées par la documentation officielle de NetSuite, des articles de blog d'experts et des sources communautaires. Les citations (avec [URL]) sont fournies pour toutes les déclarations factuelles.

1. Architecture et étapes de NetSuite Map/Reduce

Le type de script Map/Reduce de NetSuite définit un **pipeline de (jusqu'à) cinq étapes** (Source: docs.oracle.com). La figure 1 illustre ce flux :

1. **getInputData (Requis)** – Le premier point d'entrée du script. Cette fonction doit renvoyer une *collection d'entrée* (généralement une recherche ou une requête SuiteQL) qui définit les données à traiter. Elle est invoquée une fois et s'exécute de manière séquentielle.
2. **map (Optionnel)** – Traite chaque élément de la « collection » d'entrée individuellement. La fonction `map` est appelée une fois par enregistrement (ligne) et émet des paires clé-valeur via `context.write()`. Cette étape peut exécuter de nombreuses invocations en parallèle. Soit `map`, soit `reduce` doit être présent (au moins un) ; si vous ignorez `map`, vous avez besoin de `reduce` pour gérer les éléments d'entrée directement.
3. **shuffle (Étape système)** – Une étape interne (non codée par le développeur) qui regroupe toutes les valeurs émises par `map` par leurs clés. Le framework gère cela automatiquement une fois que tous les appels `map` sont terminés. (Les développeurs ne peuvent pas accéder directement à cette étape ni l'étendre.)
4. **reduce (Optionnel)** – Traite chaque ensemble de valeurs regroupées. La fonction `reduce` s'exécute une fois par clé unique, recevant cette clé ainsi qu'un tableau de toutes les valeurs issues de la sortie `map` pour cette clé. Cette étape peut également s'exécuter en parallèle sur plusieurs clés. Si `map` est omis, alors `reduce` doit être utilisé (chaque entrée devient une « valeur » avec une clé implicite). La fonction reduce peut également émettre des résultats pour l'étape suivante.
5. **summarize (Optionnel)** – Après tout le travail de map/reduce, cette étape finale s'exécute une fois. Elle reçoit un objet `summaryContext` contenant des statistiques (temps total, utilisation, nombre de cédages, erreurs éventuelles) de l'ensemble de la tâche. Les développeurs utilisent généralement `summarize` pour écrire des journaux, envoyer des notifications par e-mail ou finaliser les résultats.

Le système impose que **getInputData et summarize soient des étapes sérielles (tâche unique)**, tandis que `map` et `reduce` sont parallélisables. La documentation officielle explique : « le système utilise une seule tâche pour un script planifié. En revanche, le système crée plusieurs tâches pour un seul script map/reduce. Le système crée au moins une tâche par étape. Le système peut également créer plusieurs tâches pour les étapes map et reduce. … les étapes map et reduce sont considérées comme des étapes parallèles » (Source: docs.oracle.com). En pratique, l'enregistrement de déploiement permet même de définir une « Limite de concurrence » – le nombre maximum de processeurs SuiteCloud à utiliser pour exécuter des tâches map/reduce (Source: docs.oracle.com).

Le tableau suivant résume chaque étape :

ÉTAPE	OBJECTIF	MODE D'EXÉCUTION	RÔLE ET CONCURRENCE
getInputData	Charger/renvoyer toutes les données à traiter (par ex. une recherche enregistrée ou une requête SuiteQL) (Source: docs.oracle.com). Ce point d'entrée définit la collection d'entrée.	1 tâche (sérielle)	S'exécute une fois. Exemple : renvoyer un `search.create()` pour toutes les factures ouvertes.
map	Traiter chaque ligne d'entrée en une paire clé-valeur via `context.write()`. Votre code `map` s'exécute une fois par ligne renvoyée par getInputData (Source: docs.oracle.com).	Plusieurs tâches (parallèle)	S'exécute par enregistrement. Par ex. mapper chaque facture à son ID client en tant que clé (valeur = objet facture) (Source: docs.oracle.com).
shuffle	Étape du framework qui regroupe toutes les valeurs émises par clé (pas de code utilisateur).	1 tâche (interne)	Prépare automatiquement les données pour reduce ; par ex. collecte les valeurs de facture sous chaque clé client.
reduce	Traiter chaque groupe de valeurs pour une clé donnée. Votre code reduce s'exécute une fois par clé unique issue de map. À chaque invocation, vous recevez `reduceContext.key` et `reduceContext.values[]` (Source: docs.oracle.com).	Plusieurs tâches (parallèle)	S'exécute par clé. Par ex. pour chaque clé client, charger les factures de ce client et créer un paiement (un chargement par client) (Source: docs.oracle.com) (Source: www.houseblend.io).
summarize	Agréger les résultats et effectuer les actions finales. S'exécute une fois à la fin. `summaryContext` fournit les totaux, erreurs, etc. (Source: docs.oracle.com).	1 tâche (sérielle)	Envoyer un e-mail de notification ou journaliser le nombre total d'enregistrements traités. Inspecter également `summaryContext.mapSummary.errors` etc.

Chaque point d'entrée reçoit un objet de contexte avec des métadonnées. Par exemple, dans la fonction `map`, vous utilisez `mapContext.key` et `mapContext.value`, et appelez `mapContext.write({key:..., value:...})` pour émettre des données (Source: docs.oracle.com). Dans la fonction `reduce`, `reduceContext.key` est la clé de groupe et `reduceContext.values` est un tableau de toutes les valeurs associées (Source: docs.oracle.com). Le tableau 1 (ci-dessus) résume le cycle de vie. Comme le note succinctement une ressource communautaire : « Les scripts Map/Reduce prennent un certain temps à maîtriser… vous pouvez vous en sortir avec un script planifié pour des tâches plus petites, mais inévitablement, vous rencontrerez [des limites]. Dans ces situations, les scripts Map/Reduce sont une excellente solution » (Source: www.netsuitediagnostics.com).

Voici quelques points clés concernant les étapes : (a) **Étapes optionnelles :** Vous n'avez pas strictement besoin à la fois de `map` et de `reduce`. Vous pouvez omettre `map` (en renvoyant directement une liste de paires clé-valeur) si les enregistrements s'associent naturellement à une clé unique. Inversement, si chaque enregistrement est indépendant, vous pouvez ignorer `reduce` et laisser `map` effectuer tout le travail. (b) **Émission de données :** `map` ou `reduce` peuvent émettre des données pour l'étape suivante en utilisant `context.write()`. Si vous appelez `context.write()` dans `reduce`, cette sortie est transmise à `summarize`. (c) **Écriture unique par regroupement d'enregistrements :** La bonne pratique consiste souvent à générer une mise à jour par enregistrement original lors de l'étape `reduce`, en utilisant la clé du groupe pour charger l'enregistrement une seule fois. Un exemple de ce modèle est présenté dans [28†L299-L308], où plusieurs mises à jour de lignes de facture sont regroupées par ID de commande client afin que la commande ne soit chargée qu'une seule fois.

2. Gouvernance et limites de ressources

Tous les scripts SuiteScript fonctionnent sous le **modèle de gouvernance** de NetSuite, qui alloue des « unités d'utilisation » aux opérations API et limite les tâches mappées en conséquence (Source: docs.oracle.com) (Source: docs.oracle.com). Map/Reduce ajoute de la complexité en introduisant à la fois des *limites par invocation* (limites strictes) et des *limites souples par tâche*. Comprendre ces limites est crucial pour concevoir des scripts robustes.

2.1 Limites par invocation (strictes)

Chaque invocation de fonction de point d'entrée possède un plafond fixe en unités d'utilisation, en temps d'exécution et en « instructions ». Si une invocation dépasse sa limite, NetSuite génère une erreur `SSS_USAGE_LIMIT_EXCEEDED` (Source: docs.oracle.com), mettant fin immédiatement à cette invocation. Officiellement, pour Map/Reduce, les limites sont :

- **getInputData :** 10 000 unités d'utilisation, 60 minutes, 1 milliard d'instructions. Le dépassement met fin à `getInputData` et passe directement à `summarize` (en ignorant `map` / `reduce`) (Source: docs.oracle.com).
- **map :** 1 000 unités d'utilisation, 5 minutes, 100 millions d'instructions par invocation de fonction (Source: docs.oracle.com). (C'est la même limite de 1 000 unités que pour les scripts planifiés.) En cas de dépassement, l'invocation `map` se termine prématurément ; les tâches en attente peuvent être annulées ou relancées selon la configuration (Source: docs.oracle.com).
- **reduce :** 5 000 unités d'utilisation, 15 minutes, 100 millions d'instructions (Source: docs.oracle.com).
- **summarize :** 10 000 unités, 60 minutes (approximativement les mêmes limites que `getInputData`) (Source: docs.oracle.com).

En d'autres termes, Oracle note dans les **bonnes pratiques Map/Reduce** : « NetSuite impose des limites de gouvernance sur les invocations uniques des fonctions `map`, `reduce`, `getInputData` et `summarize` : `map` utilise 1 000 unités... `reduce` utilise 5 000... `getInputData` 10 000... `summarize` 10 000 » (Source: docs.oracle.com). En résumé, **chaque gestionnaire `map` ou `reduce` ne devrait effectuer qu'une petite quantité de travail par appel**. Par exemple, si une invocation `map` tentait de charger *et* de mettre à jour plusieurs enregistrements, elle pourrait facilement dépasser les 1 000 unités. La bonne pratique consiste à faire en sorte que chaque invocation `map` ou `reduce` traite *un seul* enregistrement (ou un groupe de clés) et émette un minimum de données, afin de rester bien en dessous de son plafond de 1 000 ou 5 000 unités. Si votre logique par invocation est très lourde, la documentation suggère même : dans ce cas, « envisagez d'utiliser un type de script différent (tel qu'un script planifié) (Source: docs.oracle.com). »

Une limite dérivée importante est la taille totale des **données persistantes**. Selon Oracle, un script Map/Reduce « ne peut pas utiliser plus de 200 Mo de données persistantes à tout moment » ; le dépassement génère une erreur `PERSISTED_DATA_LIMIT_FOR_MAPREDUCE_SCRIPT_EXCEEDED`, provoquant la fin de l'invocation actuelle et le passage du script à `summarize` (Source: docs.oracle.com). (Note : certaines sources de blog antérieures citent une limite de 50 Mo, mais la limite officielle actuelle est de 200 Mo (Source: docs.oracle.com) (Source: www.79consulting.com).) Les données persistantes incluent la taille combinée des *clés et valeurs non encore traitées*, ainsi que des *résultats écrits par* `reduce` (Source: docs.oracle.com). En pratique, cela signifie que vous devez garder les données de contexte petites (par exemple, n'émettez pas de chaînes JSON gigantesques comme valeurs).

Dans l'ensemble, les **limites strictes** garantissent qu'aucune fonction `map` / `reduce` ne s'exécute indéfiniment ou n'utilise des données excessives. Elles peuvent entraîner l'**arrêt du traitement** d'une invocation ; par exemple, si une invocation `reduce` atteint 5 000 unités, elle s'arrête même si toutes les valeurs ne sont pas traitées.

2.2 Limites souples et « Yielding » (cédence)

En plus des plafonds stricts, NetSuite utilise des **limites souples** pour garantir la santé du cluster. Après chaque invocation de fonction, NetSuite vérifie l'utilisation et le temps *accumulés* pour la *tâche* (une instance en cours d'exécution traitant de nombreux enregistrements). Si une tâche `map` ou `reduce` a consommé plus de **10 000 unités d'utilisation** au total, le système **effectue un « yield »** : il arrête cette tâche et en met une nouvelle en file d'attente (avec les mêmes données) pour poursuivre le traitement (Source: docs.oracle.com). Ce seuil souple (10 000 unités par tâche) empêche toute tâche d'arrière-plan de monopoliser un processeur. De même, il existe un paramètre « Yield After Minutes » (par défaut 60, minimum 3) sur le déploiement qui effectue un « yield » en fonction du temps écoulé (Source: docs.oracle.com) (Source: docs.oracle.com). Ainsi, une tâche longue se divisera elle-même en plusieurs exécutions.

Il est important de noter que le « yield » est *élégant* : l'invocation actuelle se termine proprement (après chaque clé ou valeur), puis NetSuite lance une autre tâche pour reprendre là où elle s'était arrêtée (Source: docs.oracle.com). Cela se produit en arrière-plan, donc du point de vue du développeur, le script continue sans intervention manuelle. Par exemple, si une tâche `map` a traité 3 000 unités sur plusieurs invocations, elle effectuera un « yield » si l'invocation suivante devait la pousser au-delà de 10 000. Les nouvelles tâches peuvent avoir une priorité (horodatage) inférieure, mais finissent par terminer le travail. (Dans `summaryContext`, le nombre total de « yields » est rapporté, afin que les développeurs puissent auditer la fréquence à laquelle les tâches ont cédé (Source: docs.oracle.com).)

Il faut noter que les limites **souples** comme les 10 000 unités ne sont pas infinies ; elles garantissent que les tâches très longues produisent plusieurs segments. Elles recoupent également la « Taille du tampon » (voir section suivante) : si vous définissez un tampon plus grand, une invocation peut traiter de nombreux enregistrements et atteindre le seuil souple plus rapidement.

2.3 Paramètres de déploiement : Concurrence, taille du tampon, etc.

L'**enregistrement de déploiement** d'un script Map/Reduce (Personnalisation > Scripting > Déploiements de scripts) contient des champs supplémentaires pour optimiser les performances (Source: docs.oracle.com). Deux champs clés sont :

- **Limite de concurrence :** Combien de processeurs SuiteCloud (threads parallèles) le système peut utiliser pour ce script. Une limite plus élevée signifie que davantage de tâches `map` / `reduce` parallèles peuvent s'exécuter (limitées par l'abonnement au compte et la disponibilité des processeurs) (Source: docs.oracle.com).
- **Taille du tampon (Buffer Size) :** Le nombre de paires clé-valeur par invocation de fonction avant que le système n'écrive un point de contrôle. La valeur par défaut est 1. Un tampon de 1 signifie que chaque appel `map` ou `reduce` gère un élément. Augmenter le tampon peut améliorer le débit (moins de changements de contexte) mais augmente le risque de travail en double lors d'une nouvelle tentative. NetSuite conseille de le laisser à 1 sauf si vous avez des besoins spécifiques (Source: docs.oracle.com).

Un autre paramètre est **Soumettre toutes les étapes en une fois** : s'il est coché (par défaut), NetSuite peut mettre en file d'attente les tâches pour toutes les étapes simultanément (bien que `map` / `reduce` soit ordonné par défaut). Désactivez cette option uniquement pour les tâches à faible priorité afin d'éviter toute surcharge. Il existe également **Yield After Minutes** (comme ci-dessus) que vous pouvez régler entre 3 et 60 (Source: docs.oracle.com).

Pris ensemble, ces paramètres de déploiement permettent de contrôler le parallélisme et la granularité de l'exécution Map/Reduce. Par exemple, une « taille de tampon » de 5 pourrait permettre à une invocation `reduce` de mettre à jour 5 lignes d'un enregistrement avant de sauvegarder, au lieu d'une par une (Source: www.houseblend.io) (Source: docs.oracle.com).

2.4 La gouvernance en pratique

En somme, les scripts Map/Reduce ont des **plafonds par invocation** (map=1k unités, etc.) et des **seuils par tâche** (10k unités). Le système les **applique** par le biais d'erreurs ou de « yields ». Par conséquent, une tâche Map/Reduce longue peut consister en de nombreuses sous-tâches séquentielles et plusieurs invocations de fonctions. Par exemple, Houseblend note que *« même si une tâche `map` atteint une limite d'utilisation/temps, le framework effectuera automatiquement un "yield" et reprogrammera le reste de cette tâche sans intervention du développeur »* (Source: www.houseblend.io). Inversement, si une limite stricte est atteinte (par exemple, la limite de 1 000 unités pour `map`), cette invocation s'arrête simplement et (par défaut) les tâches restantes doivent continuer à partir de la dernière clé (ou une erreur est enregistrée).

Nous observons que les **scripts Map/Reduce efficaces minimisent le risque d'atteindre les limites strictes** en gardant chaque gestionnaire `map` / `reduce` léger (le travail équivalent à un enregistrement). Cette distribution signifie que le gros du travail est effectué sur de nombreuses invocations, chacune restant bien en dessous de son plafond d'utilisation. Parallèlement, le mécanisme de limite souple garantit que l'ensemble de la tâche se termine finalement en enchaînant plusieurs tâches. En pratique, les développeurs doivent tester leurs scripts sur des volumes de données représentatifs, inspecter l'utilisation et le nombre de « yields » dans `summaryContext`, et ajuster la logique ou les paramètres de déploiement en conséquence (Source: www.houseblend.io) (Source: docs.oracle.com).

3. Modèles de traitement en masse et bonnes pratiques

Lors de la création de scripts Map/Reduce pour des opérations en masse, certains modèles et bonnes pratiques émergent. Nous les présentons comme des directives, dont beaucoup sont tirées de la documentation Oracle et d'exemples de la communauté.

3.1 Récupération de données et entrée (getInputData)

Renvoyez une recherche ou une requête, pas des tableaux bruts. Oracle conseille que `getInputData` renvoie un fournisseur de données (comme un objet `search.Search` ou une requête SuiteQL) plutôt que d'exécuter manuellement une recherche et de renvoyer une liste massive de résultats. Renvoyer un `search.create(...)` ou une référence à une recherche enregistrée tire parti de la planification de NetSuite et évite les délais d'attente (Source: docs.oracle.com). Par exemple :

```
function getInputData() {
    return search.create({
        type: record.Type.INVOICE,
        filters: [{name: 'status', operator: search.Operator.IS, values:'open'}],
        columns: ['entity', 'amount']
    });
}
```

En renvoyant l'objet de recherche lui-même, NetSuite gère la récupération des données par blocs. Si vous faisiez plutôt `var results = mySearch.run().getRange(0,1000)` et renvoyiez ce tableau, vous risquez une charge importante en une seule fois. La bonne pratique consiste à renvoyer une *recherche paginée* ou une référence de recherche enregistrée afin que NetSuite itère efficacement (Source: docs.oracle.com).

Alternativement, on peut utiliser **SuiteQL** pour générer les données d'entrée. Le module `N/query` de SuiteScript 2.1 (ou 2.0 via l'API de requête) permet d'écrire des requêtes SQL pour récupérer des ID internes ou des champs. Comme l'illustre Tim Dietrich, vous pouvez exécuter une requête SuiteQL dans `getInputData` (par exemple `SELECT Transaction.ID FROM Transaction WHERE Type='PurchOrd'`) et renvoyer un tableau d'ID ou d'objets pour l'étape `map` (Source: timdietrich.me) (Source: timdietrich.me). Cela peut être plus concis pour des filtres complexes. (En faisant cela, n'oubliez pas de parcourir toutes les lignes si >5000 ; l'exemple de Dietrich utilise une fonction d'assistance `selectAllRows` pour boucler jusqu'à ce que tous les résultats soient récupérés (Source: timdietrich.me).) Dans l'ensemble, utilisez la méthode qui renvoie une collection bien structurée : recherche enregistrée, SuiteQL ou même un tableau d'objets d'entrée (s'il est petit).

Filtrez étroitement dans getInputData. N'incluez que les enregistrements nécessaires dans votre entrée. Par exemple, si vous ne mettez à jour que les commandes ouvertes, filtrez sur le statut. Cela minimise la charge de travail dans `map` / `reduce`. Comme le dit un exemple, « Nous ne récupérons que les éléments qui ont réellement besoin de mises à jour, ce qui réduit les traitements inutiles » (Source: www.houseblend.io). Dans de grandes suites, une entrée non filtrée pourrait facilement dépasser les limites ou prendre des semaines à se terminer.

3.2 L'étape Map

La fonction `map` *transforme* généralement chaque résultat d'enregistrement d'entrée en une ou plusieurs sorties clé-valeur. Approches courantes :

- **Émettre des paires clé-valeur simples.** Votre valeur est souvent juste un ID ou un champ. Par exemple, `mapContext.write({key: soId, value: lineUniqueKey})` dans un scénario de mise à jour de lignes de commande client (Source: www.houseblend.io). La clé est choisie pour déterminer le regroupement à l'étape suivante.
- **Gardez `map` léger.** Ne faites qu'un travail minimal ici (le gros du travail se fait dans `reduce` en cas de regroupement). Par exemple, une fonction `map` ci-dessus ne fait guère plus que d'analyser `context.value` et d'écrire les valeurs d'ID (Source: www.houseblend.io). Cela maintient chaque invocation `map` en dessous de 1 000 unités.
- **Parallélisme :** Tous les appels `map` peuvent s'exécuter simultanément. Ils ne partagent pas d'état, donc ne comptez pas sur des variables statiques. Utilisez `context.write()` pour transmettre des données.
- **Ignorer `map` :** Si chaque enregistrement peut être autonome, vous pouvez faire *tout* le travail dans `map`. Dans une mise à jour d'inventaire en masse où chaque article est indépendant, un script a montré `getInputData` -> `map` met à jour l'article -> `summarize`, sans `reduce` (Source: www.houseblend.io).

Exemple : Dans un scénario de mise à jour en masse de commandes client, l'étape `map` regroupe les articles par commande. L'exemple de code montre `getInputData` renvoyant les résultats des lignes de facture, et `map` écrivant `context.write({key: salesOrderId, value: lineUniqKey})` (Source: www.houseblend.io). Ici, `map` analyse simplement le résultat de la recherche et émet (orderID, lineKey). Comme chaque invocation `map` ne fait pratiquement aucun appel API (juste `write`), elle reste bien en dessous de 1 000 unités.

3.3 L'étape Reduce

La fonction `reduce` reçoit une clé regroupée et son tableau de valeurs. C'est là que la plupart des opérations en masse ont lieu, généralement en chargeant des enregistrements et en mettant à jour des champs :

- **Charger chaque enregistrement une seule fois.** Un modèle courant consiste à utiliser la clé comme identifiant interne. Par exemple (commandes client), un appel `reduce` reçoit un ID de commande ainsi que toutes ses clés de ligne (Source: www.houseblend.io) (Source: www.houseblend.io). La fonction `reduce` exécute alors `record.load({type:'salesorder', id: salesOrderId})` une seule fois, met à jour toutes les lignes concernées, puis appelle `save()`. Cela évite de charger répétitivement le même enregistrement.
- **Effectuer des mises à jour ou des agrégations.** Au sein de `reduce`, parcourez les `reduceContext.values` (liste d'ID d'articles ou de clés de ligne). Par exemple, recherchez chaque ligne de sous-liste (`findSublistLineWithValue`) et définissez les nouvelles valeurs, puis enregistrez une seule fois (Source: www.houseblend.io). Pour de nombreux scénarios (calculs de taxes, génération de synthèses, envoi d'e-mails), `reduce` est l'endroit idéal pour émettre les résultats finaux.
- **Émettre depuis `reduce` (optionnel).** Vous pouvez appeler `reduceContext.write()` pour transmettre des résultats à `summarize` ou pour un traitement ultérieur. Si vous le faites, la sortie de `reduce` devient `summaryContext.output`, mais cela est souvent plus simple à utiliser à des fins de journalisation.
- **Utilisation de la gouvernance :** L'étape `reduce` dispose de 5 000 unités (par invocation). C'est généralement suffisant pour des mises à jour multilignes sur un seul enregistrement. Dans l'exemple de la commande client, Houseblend note que la mise à jour de 50 lignes est tout à fait acceptable avec 5 000 unités (Source: www.houseblend.io). L'utilisation de la gouvernance dans `map` est faible ici, donc ce cas est rarement interrompu. Soyez toutefois vigilant si une seule tâche `reduce` tente trop d'opérations (insertions par lots, calculs) à la fois.

Exemple : Le modèle de « mise à jour en masse des commandes client » de Houseblend utilise l'étape `reduce` pour appliquer toutes les modifications de lignes par commande (Source: www.houseblend.io) (Source: www.houseblend.io). Un autre exemple (articles d'inventaire) omettait totalement `reduce` car chaque mise à jour d'article était indépendante (Source: www.houseblend.io) (Source: www.houseblend.io). Dans l'exemple d'enregistrement personnalisé, il est suggéré soit de traiter les éléments comme tels (map uniquement), soit d'utiliser des clés parentes pour regrouper dans `reduce` (Source: www.houseblend.io).

3.4 Summarize et gestion des erreurs

La fonction `summarize` est optionnelle mais fortement recommandée. Elle s'exécute une fois après le traitement map/reduce. Grâce à son objet `summaryContext`, vous pouvez consulter des métriques : temps total, utilisation totale, nombre de rendements (yields), nombre d'exécutions d'entrée/map/reduce et erreurs (Source: docs.oracle.com). Une bonne implémentation de `summarize` pourrait :

- **Journaliser les totaux :** par ex. `log.audit('Résumé', 'Traitement de ' + summary.reduceSummary.keys.length + ' commandes ; ' + summary.inputSummary.errorCount + ' erreurs');`. L'exemple de Houseblend journalise le nombre total de commandes traitées et les éventuelles erreurs (Source: www.houseblend.io).
- **Itérer sur les erreurs :** Les résumés incluent les collections `summary.mapSummary.errors` et `summary.reduceSummary.errors`. Parcourir `summary.mapSummary.errors.iterator()` permet de journaliser chaque clé-valeur ayant échoué (Source: www.houseblend.io). C'est ainsi que l'exemple capture les échecs de mise à jour d'articles.
- **Notifier ou stocker les résultats :** Vous pouvez envoyer un e-mail, écrire dans un fichier ou enregistrer les sorties finales ici.

Bien que `summarize` intervienne après coup (aucune mise à jour d'enregistrement n'est possible à ce stade), il est crucial pour le débogage et les journaux d'exécution. Il ne comporte pas de limites strictes par invocation puisqu'il ne s'exécute qu'une seule fois (10 000 unités max). Utilisez-le pour faire ressortir toute anomalie (par ex. le pourcentage d'enregistrements réussis par rapport aux échecs).

3.5 Bonnes pratiques et modèles

En s'appuyant sur les conseils d'Oracle et les pratiques d'experts, voici quelques bonnes pratiques essentielles :

- **Gardez les fonctions map/reduce légères** : Comme indiqué dans la documentation, « les fonctions de script ne doivent pas inclure une série d'actions longue ou complexe » (Source: docs.oracle.com). Chaque map ou reduce doit effectuer une seule tâche (par ex. charger un enregistrement, mettre à jour un champ).
- **Utilisez `submitFields` lorsque c'est possible** : Pour des mises à jour de champs simples, préférez `record.submitFields` (mises à jour sans charger l'enregistrement complet) pour économiser des unités de gouvernance. L'exemple d'inventaire fait exactement cela pour un changement de quantité (Source: www.houseblend.io).
- **Regroupez par clé** : Si plusieurs opérations ciblent le même enregistrement, regroupez-les afin de ne charger/enregistrer qu'une seule fois. Les modèles de commande client et d'enregistrement personnalisé utilisent ce regroupement pour agréger les mises à jour par parent et réduire le

nombre de chargements d'enregistrements (Source: www.houseblend.io) (Source: www.houseblend.io).

- **Envisagez de sauter `reduce`** : S'il n'y a rien à regrouper, effectuez simplement tout le travail dans `map` et omettez `reduce`. Comme le note l'exemple d'inventaire, « nous pouvons sauter reduce... chaque article est traité individuellement ». Cela simplifie le code et évite une étape inutile (Source: www.houseblend.io).
- **Gérez les résultats de recherche avec soin** : Utilisez `runPaged` ou les retours de `search.create` pour parcourir les grandes recherches au lieu des boucles `getRange` (Source: www.thenetsuitepro.com). Cela garantit que vous pouvez récupérer plus de 1 000 résultats efficacement.
- **Utilisez la journalisation avec discernement** : Une journalisation intensive dans map/reduce peut épuiser les unités ; ne journalisez que les messages de résumé ou d'erreur en production pour rester sous les limites.
- **Surveillez l'utilisation** : Pendant les tests, appelez `runtime.getCurrentScript().getRemainingUsage()` pour vérifier la consommation d'unités et ajuster la logique si une invocation est trop lourde.

Les pages d'aide de NetSuite et les ressources communautaires énumèrent de nombreux conseils de ce type. Par exemple, un résumé des meilleures pratiques indique : *« Examinez votre script pour vous assurer que vos fonctions map et reduce sont relativement légères. Par exemple, si votre fonction map ou reduce charge et enregistre plusieurs enregistrements en même temps, envisagez un autre type »* (Source: docs.oracle.com). En pratique, la violation de ces conseils entraîne des erreurs fréquentes de limite d'utilisation et une exécution partielle des tâches.

4. Analyse comparative des méthodes de traitement en masse

Map/Reduce est une approche parmi d'autres pour le traitement en masse d'enregistrements dans NetSuite. Cette section la compare à d'autres méthodes (scripts planifiés, mises à jour en masse et même Suitelets externes) pour clarifier quand chacune est appropriée.

- **Scripts planifiés (SuiteScript 1.0/2.x) :** Un script planifié possède un point d'entrée `execute` unique qui s'exécute une fois et peut parcourir de nombreux enregistrements séquentiellement. C'est plus simple pour les petits lots. Cependant, il ne s'exécute pas en parallèle : une tâche planifiée doit se terminer avant que la suivante ne commence. De plus, il a une limite de 10 000 unités par exécution ; au-delà, vous devez appeler `runtime.getCurrentScript().yield()` manuellement pour gérer la progression. Comme le note Houseblend, « un script planifié mettant à jour 10 000 enregistrements s'exécute un par un ; un Map/Reduce pourrait diviser cela en, disons, 5 tâches map parallèles de 2 000 enregistrements chacune, terminant beaucoup plus rapidement » (Source: www.houseblend.io). Le compromis est la complexité : les scripts planifiés nécessitent souvent une logique manuelle pour céder la main et éventuellement plusieurs déploiements pour le parallélisme. Pour des tâches de moins de ~5 000 enregistrements ou une logique très simple, les scripts planifiés suffisent. Une règle empirique utile (du guide TheNetSuitePro) est : *« Utilisez Map/Reduce pour les tâches à haut volume (résultats de recherche > 5 000 lignes)… Planifié pour les plus petits lots. »* (Source: www.thenetsuitepro.com).

- **Mise à jour en masse (UI ou SuiteScript 1.0 Mass Update) :** La fonctionnalité de mise à jour en masse de NetSuite permet à un utilisateur d'appliquer une action à tous les enregistrements correspondant à une recherche enregistrée, sans codage. C'est le plus simple pour des *changements ponctuels* (par ex. définir un champ sur toutes les transactions d'une recherche enregistrée). Cependant, c'est limité aux mises à jour de champs simples (et un enregistrement à la fois en arrière-plan). Houseblend cite ce conseil : *« 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/logiques plus complexes, optez pour un Map/Reduce. »* (Source: www.houseblend.io). En bref, la mise à jour en masse est rapide à mettre en œuvre pour un usage ad hoc, mais n'est pas assez programmatique ou flexible pour des règles métier complexes.

- **Suitelets (ou RESTlets) avec appelants externes :** Certains développeurs ont expérimenté le traitement des Suitelets comme des mini-API : un outil externe (comme Apache JMeter ou un script personnalisé) appelle un Suitelet plusieurs fois en parallèle. UrsusCode a évalué cette approche et a constaté que, sous une forte concurrence, un Suitelet pouvait traiter les enregistrements beaucoup plus rapidement que Map/Reduce (Source: ursuscode.com). Par exemple, 200 requêtes Suitelet simultanées traitaient ~416 enregistrements/sec, tandis qu'un Map/Reduce (buffer=1) fonctionnait à ~51/sec (Source: ursuscode.com) (création/suppression d'enregistrements personnalisés). Cela suggère que, techniquement, des appels Suitelet massivement parallèles peuvent dépasser le débit de Map/Reduce. Cependant, cette approche est complexe et fragile : comme l'avertit l'auteur d'UrsusCode, vous devez alors construire votre propre gestion des tentatives/erreurs et utiliser des outils externes pour orchestrer la concurrence (Source: ursuscode.com). Ainsi, bien que la concurrence des Suitelets puisse gagner en vitesse brute, ce n'est pas un modèle standard ou pris en charge pour les données en masse dans NetSuite (et cela pourrait entraîner des limitations IP ou des problèmes de session). Map/Reduce reste plus sûr pour la planification intégrée et la conformité avec la gouvernance.

- **Importations CSV ou importations intégrées :** Pour certaines données, l'importation CSV de NetSuite (via l'interface utilisateur ou le File Cabinet) peut gérer de grands volumes (par ex. 50 000 enregistrements). Pour une transformation de données pure (comme le déplacement de valeurs de champ), c'est une option sans code. Son inconvénient est l'absence de logique personnalisée et de regroupement basé sur des clés. Souvent, un développeur peut exporter, transformer et réimporter les données plutôt que de les scripter. Mais pour l'automatisation ERP en temps réel, Map/Reduce permet une intégration plus étroite et une logique personnalisée immédiate.

La **comparaison** peut être résumée : Map/Reduce offre la plus grande évolutivité et flexibilité au prix d'un code plus complexe. Les scripts planifiés et les mises à jour en masse sont plus simples mais limités en débit et en résilience. Un Suitelet à haute concurrence est une solution externe qui peut être très rapide, mais non officiellement recommandée pour le traitement en masse intégré (Source: ursuscode.com). L'analyse de Houseblend conclut : *« Map/Reduce est généralement le choix privilégié pour le traitement en masse d'enregistrements... 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 plus simples ou à plus petite échelle, et la mise à jour en masse de NetSuite pour certains changements en masse ponctuels sans codage. »* (Source: www.houseblend.io).

5. Observations sur les performances et analyse des données

« En masse » dans les termes de NetSuite signifie souvent plusieurs milliers d'enregistrements. Comment Map/Reduce fonctionne-t-il réellement ? Les benchmarks sur le terrain donnent un aperçu, bien que les performances varient considérablement selon l'opération :

- **Débit (Enregistrements/Sec) :** Dans un test StackOverflow via UrsusCode, un script Map/Reduce (buffer = 1) a traité 10 000 opérations d'enregistrement personnalisé simples à ~51 enregistrements/sec (environ 196 secondes au total) (Source: ursuscode.com). Il est intéressant de noter qu'augmenter le tampon interne à 64 a en fait *ralenti* le processus (~19/sec, 518s au total), probablement à cause de la façon dont la vérification du contexte interagit avec la gouvernance. En revanche, le même auteur a atteint ~181–333 enregistrements/sec en utilisant des appels externes simultanés vers un Suitelet (Source: ursuscode.com). D'autres sources communautaires rapportent des débits Map/Reduce « de l'ordre de seulement 2 à 3 enregistrements par seconde en pratique, même avec un parallélisme modéré » (Source: www.houseblend.io). Ces chiffres dépendent fortement de ce qu'implique le traitement de chaque enregistrement ; les tests référencés vérifiaient principalement la création/suppression d'enregistrements vides.

- **Mise à l'échelle avec le parallélisme :** La configuration du système peut permettre à plusieurs tâches map/reduce de s'exécuter réellement en parallèle (sous réserve de la limite de concurrence). En théorie, si vous avez N tâches map parallèles traitant chacune M enregistrements, le débit peut être ~N fois celui d'une tâche unique. Cependant, les dépendances de données (par ex. recherches lourdes dans `getInput`, ou logique `reduce` complexe) créent souvent un goulot d'étranglement à une étape. Sagesse populaire : Map/Reduce parallélise les tâches indépendantes, mais si toutes les tâches partagent un goulot d'étranglement commun (par ex. écrire dans un seul fichier de résultat, ou attendre une seule API externe), le parallélisme aide moins.

- **Effet des scripts asynchrones (SuiteScript 2.1) :** SuiteScript 2.1 a introduit async/await et les Promesses, mais les benchmarks indiquent que cela **ne booste pas** radicalement le débit pur des enregistrements. L'analyse de Houseblend note que SuiteScript 2.1 n'est pas significativement plus rapide que 2.0 en vitesse brute (Source: www.houseblend.io). Les données mesurées de « 2–3 enr/s » s'appliquent même avec les promesses. En fait, chaque appel HTTP asynchrone coûte toujours 10 unités et Oracle prévient que les Promesses « ne sont pas destinées aux cas d'utilisation de traitement en masse » (Source: www.houseblend.io). En d'autres termes, le code asynchrone peut faciliter la vie du développeur, mais la gouvernance sous-jacente et la vitesse de traitement de NetSuite restent les facteurs limitants.

- **Impact sur la gouvernance :** Nous avons vu que les invocations map doivent rester sous les 1 000 unités. Si un développeur tente d'en faire trop dans un seul map (disons, charger plusieurs sous-enregistrements ou effectuer des calculs lourds), l'invocation s'interrompra prématurément. Cette contrainte peut effectivement brider le débit ; les scripts qui découpent soigneusement leur travail évitent de telles interruptions. Par conséquent, la **lenteur apparente** de certains tests reflète souvent un découpage conservateur pour respecter les limites.

Comme point de données illustratif, considérez le **résumé de benchmark** suivant d'UrsusCode (Adolfo Garza) (Source: ursuscode.com) :

APPROCHE	CONCURRENCE	TEMPS POUR 10 000 OPS	TAUX (OPS/SEC)
Map/Reduce (buffer=1)	1 tâche à la fois	~196 secondes	~51
Map/Reduce (buffer=64)	1 tâche à la fois	~518 secondes	~19
Appels Suitelet (Apache JMeter, 50 conc.)	50 threads simultanés	~55 secondes	~181
Appels Suitelet (100 conc.)	100 threads	~30 secondes	~333
Appels Suitelet (200 conc.)	200 threads	~24 secondes	~416

Bien qu'il ne s'agisse pas d'une mesure comparable (le Suitelet utilisait une concurrence externe), cela souligne que Map/Reduce, en tant que service géré, comporte une surcharge. Pratiquement toutes les sources soulignent que l'avantage de Map/Reduce est la fiabilité et la facilité de gestion des limites, et non la vitesse brute. L'analyse de Stockton10 a conclu de la même manière que les améliorations de SuiteScript 2.1 n'étaient « pas significativement » plus rapides que la 2.0 (Source: www.houseblend.io).

Cela dit, Map/Reduce **surpasse régulièrement un script planifié unique** sur les mêmes données, surtout à mesure que le volume augmente, grâce au parallélisme. Par exemple, un script planifié effectuant une boucle linéaire pourrait atteindre 10 000 unités et s'arrêter sans avoir terminé 10 000 enregistrements (Source: www.houseblend.io), tandis que Map/Reduce diviserait cela en plusieurs exécutions map. Ainsi, bien que le nombre brut d'enregistrements/sec puisse varier (de quelques dizaines à quelques centaines), Map/Reduce est conçu pour **évoluer en toute sécurité** vers des dizaines de milliers d'enregistrements en divisant la charge de travail.

6. Études de cas et exemples

Pour étayer notre discussion, nous présentons des cas d'utilisation concrets tirés de la communauté et de la documentation, démontrant comment Map/Reduce résout des problèmes réels.

6.1 Traitement des factures par client (Exemple Oracle)

La documentation d'Oracle fournit un exemple classique de « paiement de factures » (Source: docs.oracle.com). Imaginez un script pour payer plusieurs factures pour chaque client :

- **getInputData :** Charger toutes les factures nécessitant un paiement. (Par exemple, une recherche de factures avec un solde > 0.)
- **map :** Pour chaque facture, obtenir son client (entité) et émettre une paire clé-valeur `(customerID, invoiceID)`. Ainsi, s'il y a 5 factures, la fonction map émet 5 paires, certaines ayant potentiellement des clés en double pour le même client (Source: docs.oracle.com).
- **shuffle :** NetSuite regroupe automatiquement les valeurs par `customerID`.
- **reduce :** Pour chaque `customerID` (par exemple, 3 clients uniques pour 5 factures), recevoir un tableau des `invoiceID` de ce client. Le code de la fonction reduce peut alors créer un paiement client couvrant toutes ses factures (ou les traiter une par une). La documentation note qu'une « logique personnalisée itère sur chaque groupe en utilisant le `customerID` comme clé » (Source: docs.oracle.com).
- **summarize :** Enregistrer le nombre de paiements effectués et envoyer un e-mail récapitulatif.

Ce cas illustre le regroupement (client → plusieurs factures) suivi d'un traitement par groupe. Le travail exemplifie une transaction multi-enregistrements (création de paiements) qui serait difficile à réaliser efficacement dans une boucle unique. L'essentiel est que chaque **enregistrement de facture est chargé une fois** dans la fonction map (ou reduce), puis chaque paiement est créé une fois par client (reduce). Le code de l'exemple se trouve dans l'échantillon « Processing Invoices » d'Oracle (Source: docs.oracle.com) (et est conceptuellement similaire à l'exemple de commande client de Houseblend [28]).

6.2 Mise à jour en masse des lignes de commande client (Exemple Houseblend)

Houseblend propose un modèle Map/Reduce détaillé pour la mise à jour en masse des lignes de commande client (Source: www.houseblend.io). Cas d'utilisation : pour toutes les commandes client, mettre à jour une colonne personnalisée sur certaines lignes (par exemple, lorsqu'une ligne contient « x » dans un champ). La solution est la suivante :

- **getInputData :** Rechercher les *lignes de transaction* (lignes de commande client) où la colonne personnalisée contient « x ». Renvoyer les résultats incluant chaque ID interne de commande client (`internalid`) et la clé unique de la ligne (`lineuniquekey`) (Source: www.houseblend.io).
- **map :** Pour chaque résultat (chaque ligne), analyser le `salesOrderId` et son `lineKey`, puis appeler `context.write({key: salesOrderId, value: lineKey})` (Source: www.houseblend.io). Cela émet plusieurs valeurs pour la même clé de commande.
- **reduce :** Pour chaque clé `salesOrderId`, le gestionnaire reduce reçoit un tableau de toutes les `lineKeys` à mettre à jour pour cette commande. Le code effectue `record.load({id: salesOrderId})` une seule fois, boucle sur chaque `lineKey`, trouve le numéro de ligne de la sous-liste (`findSublistLineWithValue`) et définit la nouvelle valeur. Ensuite, il effectue un seul `soRec.save()` (Source: www.houseblend.io). Ainsi, même si une commande comporte 10 lignes nécessitant une modification, elle est chargée et mise à jour une seule fois dans la fonction reduce.
- **summarize :** Enregistrer le nombre de commandes traitées et les éventuelles erreurs (Source: www.houseblend.io).

Cet exemple met en évidence le modèle de regroupement : **regrouper par enregistrement principal afin de pouvoir mettre à jour les lignes en masse par enregistrement**. Houseblend note l'utilisation de la gouvernance : chaque reduce disposait de jusqu'à 5 000 unités pour mettre à jour une commande client entière, ce qui était suffisant (Source: www.houseblend.io). L'étape map (simple écriture de clés) utilise un nombre minimal d'unités, bien en dessous de sa limite de 1 000 unités. Cela garantit qu'aucune invocation ne dépasse les limites, même pour les commandes comportant de nombreuses lignes.

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

Autre scénario courant : modifier des milliers d'enregistrements d'articles d'inventaire (par exemple, définir un champ de prix pour tous les articles d'une catégorie). Ici, chaque article est indépendant :

- **getInputData :** Recherche enregistrée pour les articles d'inventaire répondant aux critères (par exemple, quantité > 0) (Source: www.houseblend.io).
- **map :** Pour chaque résultat d'article, analyser son ID et sa valeur actuelle, calculer la nouvelle valeur, puis appeler `record.submitFields({type:'inventoryitem', id:itemId, values:{ ... }})` pour effectuer la mise à jour. L'exemple augmente simplement la `quantityonhand` de 10 (Source: www.houseblend.io).
- **reduce :** *Non utilisé*. Comme les articles sont indépendants, tout le travail se fait dans la fonction map. Le code fourni inclut une fonction `reduce` vide (journalisation uniquement) (Source: www.houseblend.io), qui pourrait être omise.
- **summarize :** Enregistre le nombre total d'articles traités et les éventuelles erreurs (Source: www.houseblend.io).

Points clés : ici, chaque invocation map met à jour un article puis se termine. L'utilisation de `submitFields` évite de charger complètement chaque enregistrement (économie d'unités). Le script tire parti des maps parallèles : de nombreuses mises à jour d'inventaire s'exécutent simultanément. Houseblend insiste sur l'importance de filtrer rigoureusement les entrées afin que seuls les articles pertinents soient touchés (Source: www.houseblend.io). L'étape summarize comptabilise ensuite les succès et les échecs. Ce modèle (map uniquement, mises à jour indépendantes) est très simple et souvent suffisant pour les mises à jour d'enregistrements personnalisés également.

6.4 Mise à jour en masse d'enregistrements personnalisés (Exemple Houseblend)

Pour les enregistrements personnalisés, l'approche reflète celle des articles ou des commandes client selon les relations :

- **Enregistrements indépendants :** Si aucun regroupement n'est nécessaire, procédez comme pour l'inventaire : recherchez les enregistrements cibles et mettez à jour chacun d'eux dans la fonction map.
- **Enregistrements regroupés :** Si les enregistrements personnalisés ont une hiérarchie (par exemple, des enregistrements enfants avec un parent commun), vous pouvez utiliser la même astuce de regroupement. Par exemple, regroupez les ID des enregistrements enfants par ID

parent dans la fonction map, puis, dans la fonction reduce, chargez chaque parent et mettez à jour tous ses enfants en une seule fois. L'extrait suggère de regrouper par « ID parent ou catégorie » si nécessaire (Source: www.houseblend.io).

Par exemple, supposons qu'un enregistrement personnalisé « Actif » nécessite une mise à jour de statut si sa valeur dépasse un seuil. On pourrait : rechercher tous les actifs répondant aux critères ; puis dans la fonction map, `context.write({key: parentId, value: assetId})` ; puis, dans la fonction reduce, charger l'enregistrement parent et ajuster les actifs enfants. Le texte de Houseblend souligne cette approche (Source: www.houseblend.io). Bien qu'il ne s'agisse pas d'un exemple de code complet, cela indique que Map/Reduce prend également en charge les requêtes SuiteQL et les recherches enregistrées dans `getInputData` pour les enregistrements personnalisés.

7. Discussion : Implications et orientations futures

Le SuiteScript Map/Reduce offre un moyen structuré d'effectuer des processus en masse dans NetSuite, mais il comporte des compromis. En regardant vers l'avenir, quelques thèmes émergent :

- **Complexité vs contrôle :** Map/Reduce offre un excellent contrôle sur les gros travaux, mais les développeurs doivent concevoir soigneusement la logique multi-étapes. Comme le note un guide, vous devez *diviser votre logique en étapes* et penser différemment d'un script à boucle unique (Source: www.79consulting.com). Cela nécessite souvent plus de conception et de tests en amont. Cependant, une fois mis en œuvre, les scripts Map/Reduce ont tendance à être robustes et maintenables pour les charges de travail appropriées.

- **Limites de performance :** En 2026, le Map/Reduce de NetSuite est fondamentalement limité par la même gouvernance et la même vitesse de moteur qu'auparavant. Même avec les fonctionnalités asynchrones du SuiteScript 2.1, les gains de débit sont limités. Comme l'a observé Houseblend, tous les modules majeurs (recherche, requête, HTTP) disposent désormais de méthodes basées sur les promesses, permettant l'utilisation de `await` (Source: www.houseblend.io), mais chaque appel asynchrone consomme toujours des unités d'utilisation. Oracle lui-même met en garde contre l'utilisation des promesses pour les travaux en masse (Source: www.houseblend.io). Au lieu de cela, l'asynchrone de la version 2.1 aide principalement à la clarté du code. En bref, les performances de Map/Reduce ne feront pas un bond spectaculaire simplement parce que la syntaxe JavaScript est plus agréable.

- **Améliorations de la plateforme :** Oracle continue d'ajouter de nouvelles API SuiteScript qui complètent les tâches en masse. Par exemple, SuiteScript 2.x a ajouté une **API de streaming de fichiers plats** et une **API de pagination de recherche** (Source: docs.oracle.com), qui aident à lire/écrire de grands ensembles de données. De plus, des modules SFTP et de cache ont été introduits, facilitant le transfert de données. Ces fonctionnalités peuvent réduire le besoin de scripts personnalisés dans certains scénarios de migration de données. Cependant, le type Map/Reduce lui-même est resté stable – aucun nouveau « Map/Reduce v3 » n'a été annoncé, ce qui implique que le modèle actuel restera le principal pour les tâches SuiteScript lourdes. Les utilisateurs peuvent s'attendre à ce que le framework Map/Reduce prenne en charge ces nouveaux modules (par exemple, le streaming de lignes au lieu de conserver de grands résultats de recherche en mémoire).

- **Tendances d'intégration :** De nombreuses organisations alimentent des entrepôts externes avec les données NetSuite (par exemple via SuiteAnalytics ou des outils ETL tiers) pour des analyses lourdes. Cela peut décharger certaines tâches de reporting « en masse » du Map/Reduce. Mais Map/Reduce reste essentiel pour l'automatisation transactionnelle et d'intégration (par exemple, mise à jour des enregistrements en temps réel après les expéditions, application d'une logique de tarification personnalisée, etc.). Le modèle de « planification de M/R pour une synchronisation nocturne vs intégration quasi temps réel » continuera d'évoluer. Notamment, Map/Reduce peut fonctionner avec SuiteQL pour interroger des API REST externes (par exemple, en combinant SuiteQL avec des appels `https.get.promise` dans la fonction map (Source: www.houseblend.io), bien que de tels mélanges doivent respecter scrupuleusement la gouvernance.

- **Parallélisme vs coût d'utilisation :** Une considération constante est la relation entre le parallélisme et le coût. L'exécution simultanée de nombreux travaux peut se terminer plus rapidement, mais chaque invocation parallèle utilise de la gouvernance. Dans un document Oracle, l'accent est mis sur la planification prioritaire et les files d'attente des processeurs (Source: docs.oracle.com). Un script avec une concurrence élevée pourrait accaparer plus de ressources et être reprogrammé après d'autres travaux critiques. Les administrateurs peuvent définir la **Priorité** sur un déploiement pour gérer cela (Source: docs.oracle.com). Ainsi, l'avenir pourrait inclure une planification des ressources plus intelligente : pour l'instant, les développeurs peuvent surveiller les niveaux de priorité des files d'attente via les informations des processeurs SuiteCloud.

- **IA et apprentissage automatique :** C'est spéculatif, mais à mesure qu'Oracle investit dans les fonctionnalités d'IA, certaines règles métier pourraient migrer vers ce domaine. Map/Reduce pourrait servir de moteur de prétraitement de données pour l'IA dans le cloud. Cependant, à l'heure actuelle, il n'existe aucune intégration directe de l'IA de NetSuite qui modifie l'utilisation de Map/Reduce. Il reste un outil de traitement par lots à faible latence et sur plateforme.

En résumé, le SuiteScript Map/Reduce est un framework mature et bien documenté. Ses **implications** sont que les architectes peuvent traiter de manière fiable de très grands volumes de données dans NetSuite avec des modèles de mise à l'échelle connus. Ils doivent cependant planifier la gestion de la gouvernance et garder les scripts modulaires. À l'avenir, les développeurs verront probablement des mises à niveau progressives de la plateforme (syntaxe 2.1/2.2, nouvelles API) qui rendront l'écriture de Map/Reduce plus facile ou plus puissante (comme le streaming et la pagination). Mais le modèle Map/Reduce de base – points d'entrée étagés avec parallélisme automatique – persistera probablement comme la méthode canonique de NetSuite pour le traitement personnalisé complexe en masse.

Conclusion

Le SuiteScript Map/Reduce de NetSuite offre une architecture robuste pour le traitement de données en masse dans un environnement ERP cloud, combinant le paradigme classique map/reduce avec la gestion de la gouvernance spécifique à NetSuite. En structurant le travail en étapes `getInput`, `map`, `reduce` et `summarize`, les développeurs bénéficient d'un parallélisme et d'un basculement automatique qui ne sont pas disponibles dans des types de scripts plus simples (Source: www.houseblend.io) (Source: docs.oracle.com). Le compromis est la complexité : les scripts doivent être soigneusement conçus pour respecter les limites par invocation (1 000 unités pour map, 5 000 pour reduce) et pour tirer parti du regroupement afin de ne pas effectuer de travail redondant.

Cette analyse approfondie a montré que les scripts Map/Reduce excellent lorsqu'ils traitent des **dizaines de milliers d'enregistrements** pouvant être partitionnés. Avec des modèles appropriés (clés par ID d'enregistrement pour le regroupement, utilisation de `submitFields`, filtrage strict des entrées), un seul déploiement Map/Reduce peut traiter en toute sécurité un nombre « illimité » d'enregistrements totaux en découpant le travail sur de nombreux jobs (Source: www.houseblend.io) (Source: docs.oracle.com). Nous avons donné des exemples tels que le paiement de factures par client (Source: docs.oracle.com) et la mise à jour en masse des lignes de commande client (Source: www.houseblend.io) pour illustrer ces modèles en action.

Cependant, les utilisateurs ne doivent pas considérer Map/Reduce comme une solution miracle à tous les problèmes de performance. Les benchmarks indiquent que, par enregistrement, Map/Reduce n'est pas nécessairement « rapide » en termes de débit brut (Source: ursuscode.com) (Source: www.houseblend.io). Sa véritable force réside dans la mise à l'échelle au-delà des limites de job unique des anciens types de scripts et dans l'offre d'une exécution déterministe même face aux contraintes de gouvernance (Source: www.houseblend.io) (Source: www.houseblend.io). Pour des besoins très rapides ou massivement concurrents, les architectes peuvent toujours envisager des alternatives créatives (processus planifiés multiples ou appels externes) au prix d'une complexité accrue (Source: ursuscode.com).

En termes opérationnels, Map/Reduce devrait être l'outil privilégié pour les **tâches en masse régulièrement planifiées** (mises à jour nocturnes, intégrations) ou les agrégations complexes, étant entendu que les tâches plus petites ou ponctuelles pourraient utiliser des méthodes plus simples. Les administrateurs doivent surveiller les déploiements Map/Reduce via la page d'état Map/Reduce (affichant la progression et les jobs en attente) et ajuster le déploiement (priorité, concurrence, tampon, rendement) si nécessaire.

À l'avenir, le SuiteScript continuera d'évoluer (par exemple, intégration Git, API asynchrones, nouveaux modules de gestion des données (Source: docs.oracle.com) (www.houseblend.io (www.houseblend.io). Map/Reduce intégrera ces améliorations (par exemple, la prise en charge du streaming de très grands fichiers CSV) mais reste fondamentalement le même modèle. Les développeurs peuvent s'attendre à plus d'outils autour de la maintenabilité du code plutôt qu'à des accélérations fondamentales du temps d'exécution (Source: www.houseblend.io). Quoi qu'il en soit, pour les projets de personnalisation NetSuite actuels et futurs, une compréhension approfondie des étapes Map/Reduce, de la gouvernance et des modèles de traitement en masse est indispensable.

Sources : Ce rapport synthétise la documentation officielle de NetSuite (articles d'aide sur les étapes, la gouvernance et le déploiement de Map/Reduce) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com), des blogs de développeurs faisant autorité (Houseblend, UrsusCode, NetsuiteDiagnostics, etc.) (Source: www.houseblend.io) (Source: www.houseblend.io) (Source: www.houseblend.io) (Source: ursuscode.com) (Source: www.houseblend.io), ainsi que des guides communautaires et de performance (Source: docs.oracle.com) (Source: www.netsuitediagnostics.com) (Source: www.79consulting.com). Chaque affirmation concernant les limites d'utilisation, l'architecture ou les meilleures pratiques est étayée par une ou plusieurs de ces références.

Étiquettes: netsuite-suitescript, scripts-map-reduce, traitement-en-masse, gouvernance-netsuite, suitescript-2x, processeurs-suitecloud, developpement-erp

---

AVERTISSEMENT

Ce document est fourni à titre informatif uniquement. Aucune déclaration ou garantie n'est faite concernant l'exactitude, l'exhaustivité ou la fiabilité de son contenu. Toute utilisation de ces informations est à vos propres risques. Houseblend ne sera pas responsable des dommages découlant de l'utilisation de ce document. Ce contenu peut inclure du matériel généré avec l'aide d'outils d'intelligence

artificielle, qui peuvent contenir des erreurs ou des inexactitudes. Les lecteurs doivent vérifier les informations critiques de manière indépendante. Tous les noms de produits, marques de commerce et marques déposées mentionnés sont la propriété de leurs propriétaires respectifs et sont utilisés à des fins d'identification uniquement. L'utilisation de ces noms n'implique pas l'approbation. Ce document ne constitue pas un conseil professionnel ou juridique. Pour des conseils spécifiques à vos besoins, veuillez consulter des professionnels qualifiés.