# UNIVERSITÉ D'ORLÉANS

***École Doctorale Mathématiques, Informatique, Physique Théorique et Ingénierie des Systèmes***

Laboratoire d'informatique Fondamentale d'Orléans

**THÈSE** présentée par :

**Sébastien RIVAULT**

soutenue le : **02 juillet 2024**

pour obtenir le grade de : **Docteur de l'Université d'Orléans**

Discipline/ Spécialité : **Informatique**

---

# Parallélisme, équilibrage de charges et extensibilité dans le traitement des mégadonnées sur des systèmes à grande échelle

---

**THÈSE dirigée par :**

| | |
|---|---|
| **M. LIMET Sébastien** | Professeur, Université d'Orléans |

**RAPPORTEURS :**

| | |
|---|---|
| **M. D'ORAZIO Laurent** | Professeur, Université de Rennes, CNRS, IRISA |
| **M. HAINS Gaétan** | Professeur, LACL, Université Paris-Est Créteil, Président du jury |

**JURY :**

| | |
|---|---|
| **M. BAMHA Mostafa** | Maître de Conférences, Université d'Orléans, co-encadrant |
| **Mme ROBERT Sophie** | Maître de Conférences, Université d'Orléans, co-encadrante |
| **M. MAABOUT Sofian** | Maître de conférences HDR, Université de Bordeaux, LaBRI, CNRS |

# Remerciements

Je voudrais exprimer ma gratitude envers tous ceux qui, par leurs conseils ou leur amitié, m'ont apporté leur soutien. Je tiens en particulier à adresser mes plus vifs et sincères remerciements :

à **M. Gaétan Hains**, Professeur à l'université Paris-Est Créteil et Président du jury, et à **M. Laurent D'Orazio**, Professeur à l'université de Rennes, CNRS, IRISA, qui ont accepté d'être rapporteurs de cette thèse. Je leur suis très reconnaissant pour leur temps passé à la lecture du manuscrit ainsi que pour leurs précieux retours.

à **M. Sofian Maabout**, Maître de conférences à l'université de Bordeaux, pour m'avoir fait l'honneur de faire partie de mon jury.

Je tiens à remercier mes encadrants **M. Sébastien Limet**, Professeur à l'université d'Orléans et directeur de thèse, **M. Mostafa Bamha** et **Mme. Sophie Robert**, tous deux Maître de conférences à l'université d'Orléans, pour leurs patiences, leurs disponibilités, leurs confiances et leurs conseils tout au long de ma thèse.

Je suis reconnaissant envers toutes les personnes avec lesquelles j'ai eu l'opportunité de travailler durant ma thèse. Merci plus particulièrement à **M. Nicolò Tonci**, Doctorant à l'university of Pisa, et **M. Massimo Torquati**, Associate Professor à l'university of Pisa, dont la collaboration a été bénéfique et très enrichissante.

Je remercie également tous les membres du LIFO[1] et l'ensemble de mes camarades doctorants pour leur accueil.

Enfin, je remercie mes parents et mes frères, ma conjointe et tous mes amis pour leur amour et leur soutien depuis toujours.

[1]Laboratoire d'Informatique Fondamentale d'Orléans.

# Table des matières

# Liste des figures

# Liste des tableaux

# Chapitre 1

# Introduction

Durant les deux dernières décennies, grâce à la réduction des coûts de stockage, d'échange et de traitement de l'information, le volume de données générées chaque année n'a cessé d'exploser. Ces mégadonnées proviennent d'une grande variété de sources, générées par des capteurs de l'Internet des objets (IoT), des fichiers logs, mais aussi des applications industrielles et scientifiques. Ces mégadonnées ne pouvaient pas être exploitées avant le développement du cloud computing et du calcul haute performance (HPC). Elles offrent désormais un potentiel considérable pour l'extraction de nouvelles connaissances et l'amélioration de l'aide à la décision dans les entreprises et les organisations. Plus généralement, les enjeux liés au traitement de ces mégadonnées sont souvent décrits par la règle des 3V[PA16], à savoir :

- Le **Volume** des masses de données,
- La **Variété** des sources de données,
- La **Vitesse** de création, de collecte, d'analyse et de partage des données.

À titre d'exemple, le projet de radiotélescope géant, *Square Kilometre Array* [Arr], générera plusieurs téraoctets de données brutes par seconde et plusieurs centaines de pétaoctets de données analysées par an. Ces volumes de données ne représentent qu'une infime fraction du volume de données produites à l'échelle mondiale chaque année, qui se mesure en zettaoctets. Pour stocker et analyser des ensembles de données aussi volumineux, il est devenu essentiel d'utiliser des grappes de machines, ainsi que des algorithmes extensibles capables de répartir efficacement les traitements sur l'ensemble de la grappe de calcul.

Plusieurs opérations permettent de créer du lien dans ces mégadonnées pour rassembler des données provenant de diverses sources. La jointure est l'une de ces opérations très fréquemment utilisées pour rassembler des enregistrements provenant de données

relationnelles [VG84 ; SD89 ; LTO94 ; Bla10], de tables ou de sources différentes, en utilisant un attribut de jointure commun. Cependant, cette opération reste très coûteuse et des déséquilibres dans la charge de calcul peuvent survenir lors du traitement, ce qui limite les performances en pratique [SY90 ; WDJ91 ; LTO94 ; BH99 ; LL98 ; BE03 ; Bam05 ; HBL14 ; HB15]. Ces limites ont été levées en utilisant des algorithmes s'adaptant aux caractéristiques des données pour garantir une répartition équitable de la charge de traitement sur l'ensemble des machines de la grappe tout en réduisant les coûts de communication [SY90 ; LTO94 ; Bam00 ; BE03 ; Bam05 ; Has09 ; HBL14 ; HB15].

Les mégadonnées ne sont cependant pas toujours structurées de façon à avoir un attribut commun permettant de relier les enregistrements facilement entre eux. On peut donner comme exemples : le filtrage collaboratif qui nécessite de retrouver dans des masses de données, les utilisateurs ayant les mêmes goûts [Das07]. La déduplication permet de réduire drastiquement des masses de données et de les nettoyer en supprimant les doublons fortement similaires [CGK06 ; TSP08]. Lorsque les sources de données sont multiples et stockées en silos, la résolution d'entités [DSD02 ; Wan09] permet d'identifier des relations et de détecter par exemple des fraudes [MAE07] ou des malwares [OCN14]. Tous ces exemples reposent sur une notion de similarité entre les enregistrements. Cette similarité peut être exprimée mathématiquement avec une distance et un seuil, de telle sorte que deux enregistrements seront considérés similaires si la distance les séparant est inférieure au seuil donné.

L'opération de jointure par similarité consiste donc à retrouver tous les couples d'enregistrements similaires dans une ou plusieurs masses de données, et ce, sans attribut de jointure, mais avec une distance et un seuil défini par l'utilisateur. Bien que les jointures parallèles aient été étudiées et mises en œuvre avec succès sur des architectures distribuées, les algorithmes ne sont pas adaptés à l'opération de jointure par similarité, car il n'y aucune technique de hachage ou de tri permettant de retrouver tous les couples d'enregistrements potentiellement similaires dans la littérature.

Naïvement, il est possible de comparer tous les couples d'enregistrements, ce qui nécessite le calcul d'un produit Cartésien [Afr12]. Cependant, les performances et l'extensibilité de cette approche sont très fortement limitées. D'autres techniques permettent de réduire l'espace de recherche tout en garantissant la complétude du résultat, mais leurs extensibilités sont également très limitées dans la pratique et ne permettent pas le traitement de mégadonnées [Fie18]. Ces limites ne semblent pas pouvoir être facilement surmontables et une approche différente semble nécessaire pour traiter efficacement cette

opération sur des mégadonnées.

En renonçant à une petite partie du résultat, des méthodes approximatives semblent efficaces, mais il n'y a pas d'études dans la littérature le démontrant à large échelle. Ces méthodes reposent sur des fonctions de hachages dont les probabilités de collisions sont sensibles à la similarité des objets. Elles s'appuient sur le cadre théorique de la Locality Sensitive Hashing (LSH) [IM98 ; GIM99] permettant de garantir une certaine exhaustivité du résultat tout en réduisant drastiquement l'espace de recherche, ce qui ne dénature pas l'opération. Bien que le cadre soit général, il n'existe pas toujours des familles de fonctions LSH efficaces pour toutes les distances [And09]. Plus particulièrement, les familles de fonctions LSH, pour les distances se calculant en temps quadratique, présentent généralement des défauts, ce qui complique le paramétrage et limite en pratique leurs utilisations sur des mégadonnées. De plus, comme pour la jointure, des déséquilibres dans la charge de travail des différentes machines peuvent survenir et avoir un effet désastreux sur les performances.

La jointure par similarité reste donc un défi majeur pour des ensembles de données volumineux, et ce, même en se reposant sur des systèmes à très large échelle. La jointure par similarité étant reconnue dans la littérature comme étant parmi les opérations de traitement et d'analyse de données les plus utiles, il est donc essentiel de développer des algorithmes de jointure par similarité extensibles et efficaces. Finalement, la variété des mégadonnées nécessite le développement d'une solution générique capable de s'adapter à diverses distances, et ce, qu'importe le seuil défini par l'utilisateur.

## 1.1 Principales contributions

Cette thèse se concentre sur le traitement de la jointure par similarité sur des ensembles de données très volumineux et contribue à :

- Traiter la jointure par similarité sur des mégadonnées et pour divers types d'objets et de distances. Plus particulièrement, nous nous intéressons au traitement de trajectoires, d'ensembles et de séquences en utilisant diverses distances. Nous proposons également une solution générique pouvant être étendue par l'utilisateur pour supporter diverse objets, ainsi que diverses distances.
- Éviter les effets des divers déséquilibres pouvant se produire lors de la jointure par similarité en répartissant équitablement la charge de calcul sur les nœuds

de la grappe, et ce, peu importe la distribution des données en entrée. Plus précisément, nous avons réutilisé et adapté les histogrammes distribués et les schémas de communication randomisés développés pour l'équi-jointure, afin de répartir équitablement le traitement de la jointure par similarité en fonction des caractéristiques des données tout en réduisant drastiquement les coûts de communication et de traitement aux seules données pertinentes.

- Réduire drastiquement l'espace de recherche, ainsi que les coûts de traitement, de communication et de lecture/écriture des données sur les disques en utilisant des familles de fonctions LSH, évitant ainsi une approche naïve tout en produisant une très grande majorité des résultats.

- Calculer efficacement la recherche et la jointure par similarité sur des ensembles volumineux de longues séquences et pour de très larges seuils de la distance de Levenshtein. Nous proposons une famille de fonctions LSH permettant de réduire drastiquement le nombre de distances calculées et les coûts de communication tout en garantissant de produire une large majorité des résultats.

- Analyser les coûts et les performances de chaque algorithme proposé en utilisant un modèle de coût adaptant le modèle BSP (Bulk Synchronous Parallelism), nous avons également évalué à travers une série d'expériences sur de grands ensembles de données pour divers objets, distances et seuils. Nous avons aussi mesuré la réduction de l'espace de recherche et l'exhaustivité des résultats produits lors de chaque expérience.

## 1.2 Organisation de la thèse

Le chapitre 2 présente l'opération de jointure, à la fois séquentielle et parallèle, et aborde les différents déséquilibres pouvant survenir lors du traitement de la jointure parallèle. Nous présentons également les environnements distribués, Apache Hadoop et Apache Spark, conçus pour le traitement de mégadonnées, ainsi que le modèle de coût associé, qui sera utilisé tout au long de cette thèse. Enfin, nous introduisons une optimisation et son implémentation sur des architectures distribuées, visant à répartir équitablement les charges en fonction des caractéristiques des données, afin d'éviter les effets des déséquilibres des données durant toutes les étapes de traitement.

Le chapitre 3 se concentre sur l'opération de jointure par similarité et présente un état de l'art des principales techniques utilisées pour traiter cette opération sur des mégadonnées. Nous classons ces approches selon la complétude des résultats qu'elles

produisent. L'idée des méthodes approximatives consiste à sacrifier une faible proportion des résultats pour réduire considérablement l'espace de recherche et améliorer l'efficacité. Nous introduisons le cadre LSH, fondement de ces méthodes, qui permet la construction de fonctions de hachage sensibles à la similarité. Finalement, nous présentons les outils permettant de mesurer la complétude des résultats et la réduction de l'espace de recherche.

Dans le chapitre 4, nous présentons l'algorithme *MRS-join*, reposant sur le modèle de programmation MapReduce, qui réparti équitablement les charges en fonction des caractéristiques des données sur l'ensemble des machines de la grappe. Cet algorithme exploite des familles de fonctions LSH pour réduire drastiquement l'espace de recherche, tout en produisant une large proportion des résultats de la jointure par similarité pour diverses distances.

Le chapitre 5 présente l'algorithme *MRSF-join*, qui est une optimisation de l'algorithme *MRS-join* permettant de traiter efficacement des ensembles de séquences très volumineux. Nous y présentons une technique basée sur LSH qui permet de réduire considérablement l'espace de recherche en utilisant la distance de Levenshtein, même pour des seuils très élevés. Cet algorithme garantit également la répartition équitable des charges sur l'ensemble des nœuds de la grappe.

Dans le chapitre 6, nous présentons le squelette algorithmique de la jointure par similarité destiné au traitement de mégadonnées. Nous présentons également une implémentation de ce squelette dans un environnement de calcul haute performance (HPC). Ce squelette s'appuie sur une version générique de l'algorithme *MRS-join*, qui permet aux utilisateurs de fournir uniquement les composants spécifiques à leur distance et à leurs objets.

Dans le chapitre 7, nous traitons de l'opération de recherche par similarité, dont l'objectif est d'identifier rapidement l'ensemble des enregistrements similaires à un enregistrement donné au sein d'un vaste jeu de données pré-traitées. Nous proposons une implémentation de cette opération basée sur le framework Apache HBase et procédons à l'évaluation de sa performance en utilisant un large ensemble de séquences.

Le chapitre 8 conclut cette thèse et présente les perspectives pour les travaux futurs, ainsi que les problèmes ouverts pour optimiser davantage les opérations de jointure par similarité et de recherche par similarité sur des systèmes à grande échelle.

# Chapitre 2

# Traitement de la jointure, de la jointure parallèle : État de l'art

Dans ce chapitre, nous introduisons l'opération de jointure et les algorithmes de jointure sur des architectures parallèles et distribuées. Nous examinons ensuite le modèle de programmation MapReduce permettant de déployer et d'orchestrer à très large échelle le traitement d'applications distribuées. La parallélisation de l'opération de jointure permet théoriquement de traiter de très grands ensembles de données, cependant des déséquilibres dans la charge de travail des différentes machines peuvent induire des coûts de communication élevés et avoir un effet désastreux sur les performances. Nous présentons les travaux de la littérature permettant de pallier ces problèmes tout en réduisant la charge de travail aux données pertinentes.



## 2.1 La jointure séquentielle

La jointure dans le cadre de bases de données permet d'associer une ou plusieurs tables par le biais d'un attribut commun. Nous nous concentrons sur l'équi-jointure entre deux relations R et S, notée R $\bowtie$ S, ayant comme résultat la concaténation des enregistrements de R et de S vérifiant l'égalité R.$x$ = S.$x$ pour un attribut commun $x$ fournit par l'utilisateur. Une illustration de l'opération d'équi-jointure entre deux relations est présentée dans la figure Fig. 2.1.

Le nombre de tuples résultants d'une opération de jointure dépend des fréquences des valeurs de l'attribut de jointure dans les relations R et S. Au maximum, la taille du résultat peut représenter $\|R\| \times \|S\|$ où $\|T\|$ désigne le nombre d'enregistrements dans un ensemble de données T. Cela se produit uniquement dans le cas où l'attribut de jointure $x$ n'a qu'une seule valeur dans les ensembles de données R et S. Lorsque l'attribut de jointure peut prendre plusieurs valeurs, le nombre de résultats est généralement bien inférieur.

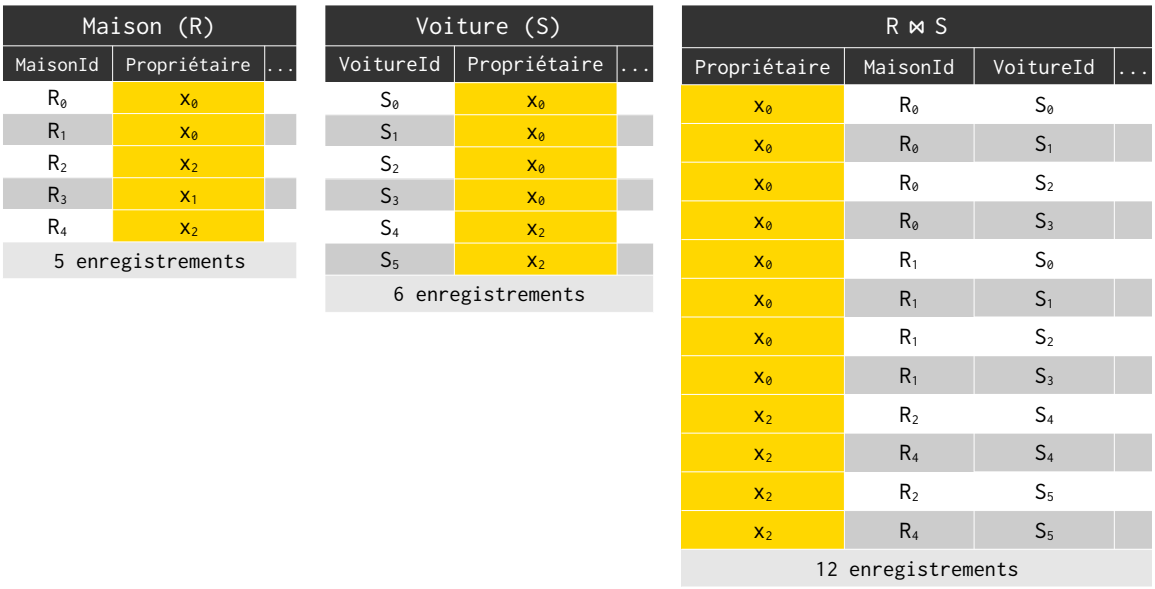



Fig 2.1 : Un exemple d'une équi-jointure entre deux relations R et S en utilisant l'attribut "Propriétaire" pour la jointure.

Nous nous concentrons sur un algorithme de jointure par hachage qui sera utilisé par la suite [VG84 ; SD89 ; LTO94], la littérature concernant les algorithmes de jointure séquentielle est en dehors du cadre de cette thèse. Cet algorithme se présente en deux phases :

1. Les prédicats de sélection sont appliqués sur les enregistrements, ainsi que la sélection des attributs ;
2. Une table de hachage est construite en utilisant les valeurs de l'attribut de jointure de la plus petite relation. Chaque valeur est associée à l'ensemble des enregistrements (ou à leurs références) lui correspondant ;
3. Pour chaque enregistrement de la plus grande relation, on recherche dans la table de hachage pour la valeur de l'attribut de jointure correspondante, et les tuples résultant de la jointure sont produits en sortie.

## 2.2 La jointure parallèle

Pour de très grands ensembles de données, la jointure parallèle a reçu beaucoup d'attention dans la littérature [Bam00 ; BE03 ; Bam05 ; Has09 ; Bla10 ; HBL14 ; HB15] afin de traiter efficacement l'opération de jointure sur des architectures parallèles et distribuées. Nous exposons ici une version parallèle de l'algorithme de hachage précédent pour aborder la jointure sur une architecture parallèle :

1. Les enregistrements sont partitionnés sur les processeurs, ou leurs disques dans le cas d'une architecture à disques répartis ;
2. Les prédicats de sélection sont appliqués sur les enregistrements locaux de chaque processeur, ainsi que la sélection des attributs ;
3. Les tuples sont redistribués et partitionnés en fonction de la valeur de leur attribut de jointure, une fonction de hachage est généralement utilisée pour répartir équitablement les valeurs sur l'ensemble des processeurs ;
4. La jointure est calculée pour les tuples ayant la même valeur de l'attribut de jointure, et les tuples résultant de la jointure sont produits en sortie.

### 2.2.1 Les problèmes de déséquilibre dans le traitement de la jointure parallèle

L'algorithme de jointure parallèle présenté dans la section précédente a quelques limites lors du traitement de très grands ensembles de données. En effet, le traitement de la jointure peut être distribué de manière inefficace, c'est-à-dire que quelques processeurs peuvent traiter la majeure partie du calcul. Ce déséquilibre peut être causé par les deux cas suivants [WDJ91] :

- Le **Partition Skew (PS)** est un déséquilibre dans le partitionnement des données, il se produit lorsque la charge de travail n'est pas uniforme sur l'ensemble des processeurs ;
- L'**Attribute Value Skew (AVS)** est un déséquilibre intrinsèque aux données, il se produit lorsque la distribution des valeurs de l'attribut de jointure n'est pas uniforme.

#### a) Les problèmes de déséquilibre de partitionnement des données (PS)

Un déséquilibre dans le partitionnement des données peut se produire même lorsque la distribution des valeurs de l'attribut de jointure est uniforme. La taxonomie de Walton

et al. [WDJ91] catégorise les différents déséquilibres de partitionnement pouvant se produire durant une opération de jointure sur une architecture parallèle en fonction de l'étape à laquelle il se manifeste :

1. **Tuple Placement Skew (TPS) :** Il se produit lorsque les enregistrements ne sont pas uniformément partitionnés sur les différents processeurs ;
2. **Selectivity Skew (SS) :** Il se produit lorsque la sélectivité des prédicats varie entre les différents processeurs ;
3. **Redistribution Skew (RS) :** Il se produit lorsque la distribution des enregistrements après la phase de redistribution n'est plus uniformément partitionnée sur les processeurs ;
4. **Join Product Skew (JPS) :** Il se produit lorsque la taille des résultats de la jointure varie entre les différents processeurs.

Nous reviendrons plus tard sur le **Tuple Placement Skew (TPS)** qui est inhérent à l'architecture parallèle utilisée pour le traitement de la jointure. Le **Selectivity Skew (SS)** dépend fortement de la requête de l'utilisateur et ne peut pas être résolu a priori. Cependant, nous supposons que les données en entrée ont été placées de manière aléatoire sur les différents processeurs, ce qui permet d'équilibrer le biais de sélectivité sur l'ensemble des processeurs. Dans la littérature, Bamha et al. [Bam00] remarque que le **Redistribution Skew (RS)** ne peut être causé que par :

- *Un mauvais choix de fonction de hachage.* Ce déséquilibre peut être évité en utilisant les techniques de hachage présentées dans la littérature [CW79] qui ont la propriété de distribuer les données uniformément sur l'ensemble des processeurs avec une très forte probabilité ;
- *Un déséquilibre intrinsèque aux données.* Nous reviendrons plus tard sur ce problème de déséquilibre et nous présenterons une solution pour y remédier.

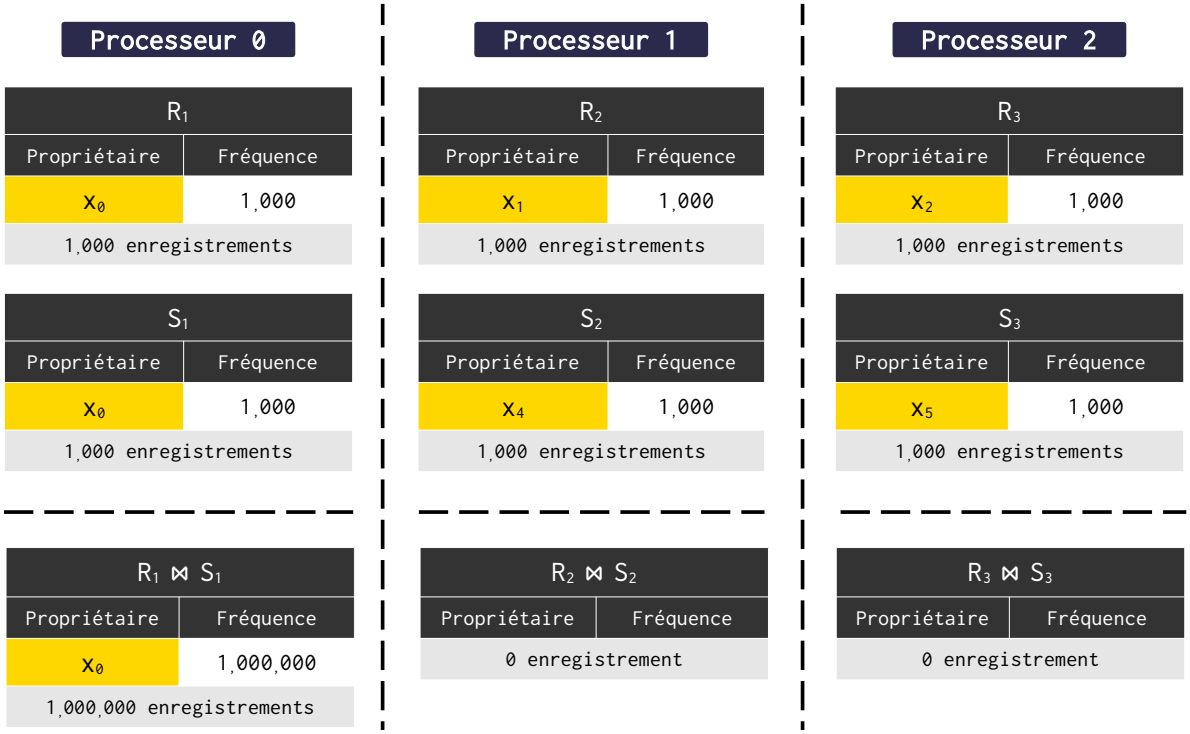



Fig 2.2 : Un exemple d'un **Join Product Skew (JPS)** pour une équi-jointure R ⋈ S sur trois processeurs en utilisant la "meilleure" fonction de hachage possible définie par $h(x) = x \mod 3$ pour partitionner les valeurs de l'attribut de jointure $x$ sur les trois processeurs. Tous les enregistrements correspondants à une valeur de l'attribut de jointure seront donc transmis à un seul et même processeur. La fréquence d'une valeur de l'attribut de jointure est définie comme le nombre d'occurrences de cette valeur dans une relation.

Il reste le **Join Product Skew (JPS)**, il se produit lorsque le calcul de la jointure n'est pas réparti uniformément sur l'ensemble des processeurs, un exemple est illustré dans la figure Fig. 2.2. Ce déséquilibre peut avoir un effet désastreux sur les performances de la jointure parallèle, dans le pire des cas, un unique processeur traite la jointure de façon séquentielle et produit l'ensemble des résultats. Notons que dans l'exemple précédent, les données en entrée sont parfaitement équilibrées sur les processeurs.

De plus, l'étape de redistribution est aussi coûteuse sur une architecture distribuée (*Shared-Nothing*), puisque l'ensemble des enregistrements doivent être transmis sur le réseau. Dans l'exemple de la figure Fig. 2.2, l'utilisation du réseau n'est pas optimisé puisque les partitions des deux derniers processeurs ne produisent aucun résultat lors de la jointure. Un algorithme optimisé traitant la jointure sur une architecture parallèle devra prendre en compte ce facteur en ne transmettant que les données pertinentes, c'est-à-dire les enregistrements ayant une valeur de l'attribut de jointure susceptible de participer aux résultats de la jointure. Plus généralement, il est impératif de garantir un

équilibre dans la répartition des charges des processeurs à chaque étape de l'algorithme pour garantir l'efficacité de l'opération de jointure sur une architecture distribuée.

### b) Le problème de déséquilibre intrinsèque aux données (AVS)

| R | |
|---|---|
| Attribut de jointure | Fréquence |
| $x_0$ | 1 000 |
| $x_1$ | 1 000 |
| $x_2$ | 5 000 |
| 7 000 enregistrements | |

| S | |
|---|---|
| Attribut de jointure | Fréquence |
| $x_0$ | 1 |
| $x_1$ | 100 |
| $x_2$ | 2 000 |
| $x_3$ | 3 010 |
| $x_4$ | 2 000 |
| 7 111 enregistrements | |

| R ⋈ S | |
|---|---|
| Attribut de jointure | Fréquence |
| $x_0$ | 1 000 |
| $x_1$ | 100 000 |
| $x_2$ | 10 000 000 |
| 10 101 000 enregistrements | |

Fig 2.3 : Un exemple de déséquilibre intrinsèque aux données pour une jointure R ⋈ S.

La figure Fig. 2.3 illustre un exemple d'**AVS**, bien que les deux ensembles de données R et S aient globalement la même taille, la distribution des valeurs de l'attribut de jointure n'est pas uniforme. Pour de très grands ensembles de données, un très fort déséquilibre provoque un effet désastreux sur les performances lors du traitement de la jointure. En effet, chaque valeur de l'attribut ne sera traitée que par un unique processeur, induisant un **RS**. Ce scénario est illustré par la figure Fig. 2.4 qui présente un exemple de partitionnement des données lors de la phase de redistribution des enregistrements.

Il faut noter dans cet exemple que l'**AVS** provoque aussi un **JPS**. En effet, la répartition du traitement de la jointure varie très fortement d'un processeur à l'autre, comme en témoigne le nombre de résultats produits par chacun des processeurs. Globalement, l'**AVS** aggrave les **déséquilibres de partitionnement**, ce qui limite l'extensibilité des algorithmes de jointure sur des architectures parallèles. Il faut par conséquent les prendre en compte lors de la conception d'algorithme afin d'assurer un équilibre dans la répartition des charges sur l'ensemble des processeurs et de garantir l'efficacité de l'approche.

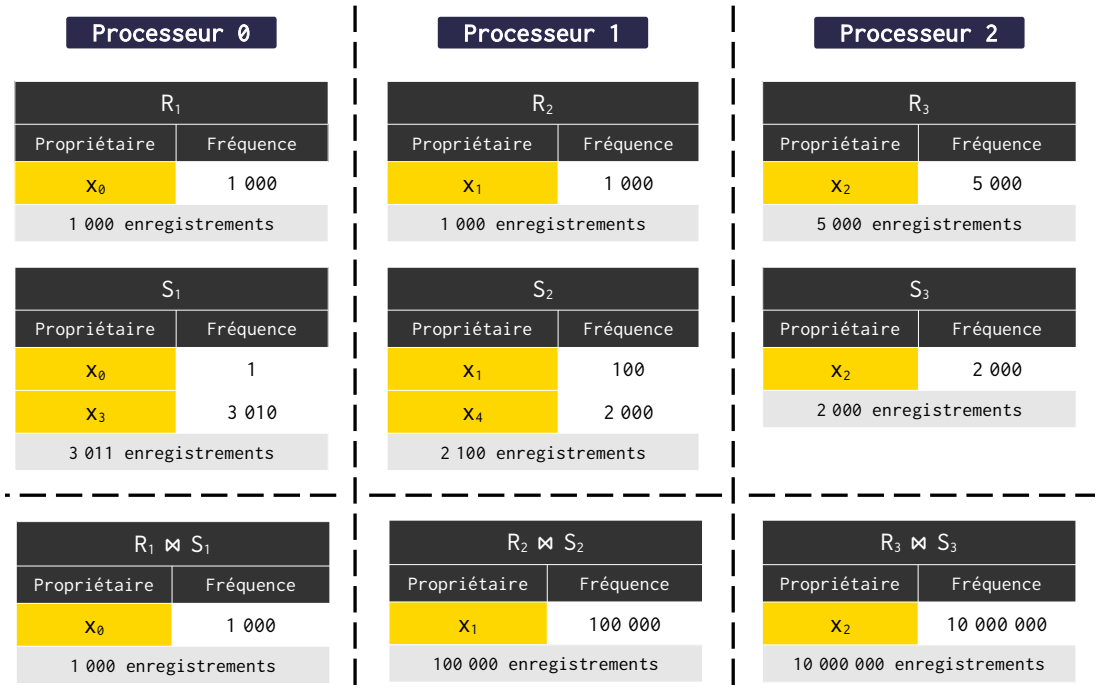



Fig 2.4 : Un exemple de distribution sur trois processeurs dans le cas d'un déséquilibre intrinsèque aux données pour une jointure entre deux ensembles de données R et S en utilisant la "meilleure" fonction possible de hachage définie par $h(x) = x \mod 3$ pour partitionner les enregistrements sur les trois processeurs. La fréquence d'une valeur de l'attribut de jointure est définie comme le nombre d'occurrences de cette valeur dans une relation.

### c) Une optimisation pour répartir les charges en tenant compte des caractéristiques des données

Pour éviter ces situations de déséquilibre et maximiser l'utilisation des capacités de traitement des machines parallèles, il est important que les données en entrée/sortie et les traitements soient équitablement répartis sur l'ensemble des processeurs. Des algorithmes s'adaptant aux caractéristiques des données sont présentés dans la littérature sur plusieurs architectures parallèles et pour divers types de jointure [BE03 ; Bam05 ; Has09 ; HBL14 ; HB15]. Généralement, ces algorithmes nécessitent au moins une étape préliminaire avant de traiter la jointure afin de déterminer les fréquences des différentes valeurs de l'attribut de jointure. Soit R un ensemble de données et $\chi(\mathrm{R})$ l'ensemble des valeurs de l'attribut de jointure de la relation R.

**Définition 1.** L'histogramme de la relation R, noté Hist(R), est défini par la liste des couples $(x, \mathbf{f}_x^{\mathrm{R}})$, où chaque valeur de l'attribut de jointure $x \in \chi(\mathrm{R})$ est associée à sa fréquence $\mathbf{f}_x^{\mathrm{R}}$, c'est-à-dire au nombre d'occurrences de la valeur $x$ dans la relation R.

L'histogramme de la jointure $\mathrm{R} \bowtie \mathrm{S}$ est construit naturellement à partir des deux histogrammes des relations R et S. Afin de réduire les coûts de communication, seules les

données pertinentes doivent être envoyées. Pour ce faire, seules les valeurs de l'attribut de jointure susceptibles de produire un résultat sont conservées dans l'histogramme de la jointure. Une valeur de l'attribut de jointure produisant un résultat implique que cette valeur soit présente à la fois dans les deux ensembles de données R et S. L'histogramme de la jointure $\mathrm{R} \bowtie \mathrm{S}$ ne contenant que les données pertinentes est défini comme suit :

**Définition 2.** $\mathrm{Hist}(\mathrm{R} \bowtie \mathrm{S}) = \{(x, \mathbf{f}_x^{\mathrm{R}}, \mathbf{f}_x^{\mathrm{S}}) \mid x \in \chi(\mathrm{R}) \cap \chi(\mathrm{S})\}$.

Il est important de rappeler que la taille de l'histogramme de la jointure est généralement très petite par rapport aux tailles des deux histogrammes des relations R et S, et encore plus petite par rapport à la taille des données en entrée et à celle des résultats de la jointure.

Dans le cas d'une auto-jointure sur un ensemble de données $\Gamma$, une valeur de l'attribut de jointure produisant un résultat implique que sa fréquence soit strictement supérieure à 1. En effet, nous ne considérons pas les couples de deux enregistrements identiques pour l'auto-jointure. L'histogramme d'une auto-jointure $\Gamma \bowtie \Gamma$ ne contenant que les données pertinentes est alors défini par :

**Définition 3.** $\mathrm{Hist}(\Gamma \bowtie \Gamma) = \{(x, \mathbf{f}_x) \mid x \in \chi(\Gamma) \wedge \mathbf{f}_x > 1\}$.

L'idée est maintenant d'utiliser cette information pour pouvoir distribuer efficacement le traitement de la jointure sur l'ensemble des processeurs. Dans ce but, des schémas de communication sont générés pour chaque valeur de l'attribut de jointure permettant un processus de redistribution des données équilibré et efficace. Nous revenons plus tard sur les détails des algorithmes et des schémas de communication employés sur notre architecture cible.

### 2.2.2 Le problème de déséquilibre dû à l'hétérogénéité de l'architecture

Des déséquilibres peuvent également être inhérents à l'hétérogénéité de l'architecture de la grappe de machines. Ces déséquilibres ne dépendent pas de l'opération de jointure spécifiquement et peuvent se produire pour tout type de traitement. Deux cas sont identifiés par [LL98] :

- **Heterogeneity Skew (HS) :** Il se produit lorsque les performances des machines parallèles varient, c'est-à-dire que la capacité de traitement des machines est déséquilibrée ;
- **Concurency Skew (CS) :** Il se produit dans un environnement multi-utilisateur lorsqu'une ou plusieurs machines sont sollicitées pour d'autres tâches simultanément.

Le **HS** est généralement dû à l'utilisation d'une grappe de machines hétérogènes présentant des différences matérielles. Pour les **déséquilibres de partitionnement**, l'hypothèse est de répartir les données uniformément sur l'ensemble des processeurs pour assurer l'efficacité. Cependant, pour maximiser l'utilisation des ressources disponibles et optimiser les performances, il serait souhaitable que la répartition des charges soit proportionnelle à la capacité de traitement des machines. Autrement dit, un nœud devrait pouvoir traiter deux fois plus de données s'il est deux fois plus puissant.

Le **CS** survient lorsque plusieurs tâches sont lancées en concurrence sur une machine par plusieurs utilisateurs. En extrapolant, ce déséquilibre peut également survenir en cas de panne matérielle sur une machine, entraînant une réduction de sa capacité de traitement. Ce déséquilibre peut survenir à tout moment et n'est pas modélisable a priori. En conséquence, la gestion d'un **HS** doit être dynamique et à la demande, c'est-à-dire qu'il faut pouvoir diviser le traitement en tâches et ajuster dynamiquement la répartition du travail lors de l'attribution des tâches en fonction des capacités de traitement courantes. Nous rappelons que le coût de la redistribution des tâches peut être également élevé dans le cas des architectures distribuées.

## 2.3 La jointure distribuée dans le cadre de MapReduce

Nous introduisons maintenant le modèle de programmation MapReduce et son architecture distribuée permettant de traiter de grands ensembles de données sur un très grand nombre de machines d'entrée de gamme, appelées machines de commodité. Nous évaluons ensuite les algorithmes traitant la jointure sur cette architecture ainsi que leurs limites.

### 2.3.1 Le modèle de programmation MapReduce

MapReduce est un modèle de programmation simple, mais puissant qui permet de développer des applications distribuées sans connaissance approfondie des problématiques liées à la redistribution des données, à l'attribution des tâches ou à la tolérance aux pannes dans les systèmes distribués à grande échelle. L'ensemble de ces problématiques étant automatiquement pris en charge par le système, et ce, sans intervention de l'utilisateur.

Le modèle de programmation MapReduce de Google [DG08], est basé sur deux fonctions : **map** et **reduce**, que l'utilisateur fournit. Ces deux fonctions ont les signatures suivantes :

**map**: $(K_{in}, V_{in}) \rightarrow List(K_m, V_m)$,
**reduce**: $(K_m, List(V_m)) \rightarrow List(K_{out}, V_{out})$.

La fonction **map** prend deux arguments en entrée, une clé $K_{in}$ et une valeur $V_{in}$. Elle produit en sortie une liste de couples clé-valeur $(K_m, V_m)$. Cette liste est ensuite partitionnée en fonction des clés $K_m$. Tous les couples ayant la même clé $K_m$ appartiennent à la même partition et seront transmis à un même *Reducer*.

La fonction **reduce** prend deux arguments en entrée : une clé intermédiaire $K_m$ et la liste des valeurs intermédiaires correspondante $List(V_m)$. Elle applique la logique de fusion définie par l'utilisateur à la liste $List(V_m)$ et produit une liste de couples clé-valeur $List(K_{out}, V_{out})$ en sortie.

Pour des raisons d'efficacité, MapReduce fournit une fonction supplémentaire, appelée **combine**, qui permet de réduire la quantité de données intermédiaires envoyées aux *Reducers*. La fonction **combine** agit de la même façon que la fonction **reduce**. Elle est appliquée au niveau des *Mappers* avant de stocker et de transmettre les résultats intermédiaires aux *Reducers* durant la phase de **map**. La signature de la fonction **combine** est la suivante :

**combine**: $(K_m, List(V_m)) \rightarrow List(K'_m, V'_m)$.

#### a) Apache Hadoop

Apache Hadoop [Had] est une solution open source développée par la fondation Apache. Il fournit un environnement fiable, extensible et distribuée permettant de stocker, traiter et d'analyser des volumes massifs de données de manière efficace. Apache Hadoop fournit les composants suivants :

**Hadoop Distributed File System (HDFS)** Un système de fichiers distribués conçu pour stocker des fichiers très volumineux avec des schémas d'accès aux données en flux continu. Pour assurer la tolérance aux pannes, le HDFS découpe les fichiers en blocs et répartit chaque *bloc* sur différents nœuds de la grappe de calcul ;

**Yet Another Resource Negotiator (YARN)** Un gestionnaire de ressource et de la planification des tâches, responsable de l'allocation des ressources aux tâches exécutées sur la grappe de calcul ;

**MapReduce** Une implémentation du modèle de programmation MapReduce ; l'exécution d'une phase de MapReduce est appelé un *job*.

D'autres composants de l'écosystème Hadoop ont complété cette pile en fournissant divers outils pour le stockage, le traitement et l'analyse de ces mégadonnées ; une illustration de cet écosystème est présente dans la figure Fig. 2.5.

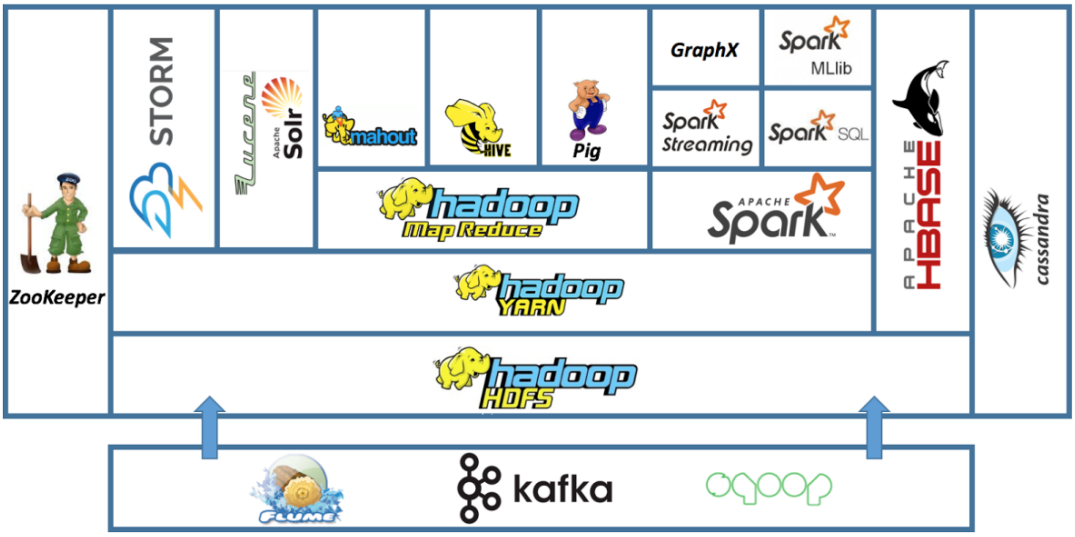



Fig 2.5 : L'écosystème Apache Hadoop (source : blog.newtechways.com/2017/10/apache-hadoop-ecosystem.html)

La figure Fig. 2.6 illustre le fonctionnement général de Hadoop MapReduce, une tâche **map** correspond à l'application de la fonction **map** à chaque enregistrement de sa portion de données. Cette portion de données est appelée un *split* et sa taille est liée à celle des *blocs*. De la même façon, une tâche **reduce** correspond à l'application de la fonction **reduce** aux données intermédiaires générées par les différentes tâches **map**. La division du travail sous forme de tâches permet facilement d'assurer la tolérance aux pannes puisqu'il suffit de relancer la tâche sur un autre nœud en cas d'échec. Les nœuds de la grappe de calcul exécutant une ou plusieurs tâches **map** sont appelés des *Mappers*. De la même manière, les nœuds exécutant une ou plusieurs tâches **reduce** sont appelés des *Reducers*.

Pour répondre à un large éventail de besoins des utilisateurs en termes de calcul et de distribution des données, MapReduce fournit deux fonctions supplémentaires, **init** et **close**, qui sont appelées avant et après chaque tâche **map** ou **reduce**.

Par ailleurs, le regroupement des clés est contrôlé par la fonction **partition** qui peut être également fournie par l'utilisateur. En pratique et par défaut, cette fonction est implémentée en appelant la méthode de hachage **hashcode()** sur les clés intermédiaires pour les répartir dans les différentes tâches **reduce**. Il faut ajouter que cette méthode doit être déterministe pour garantir qu'une même clé soit envoyée sur une seule tâche **reduce**. Le nombre total de tâches **reduce** exécutées par les *Reducers* doit être spécifié en amont par l'utilisateur ; il dépend généralement de la taille de la grappe de calcul et de la quantité de données/calcul à traiter par l'algorithme. Il faut remarquer que pour certains algorithmes très particuliers, il peut être utile de fixer le nombre de tâches **reduce** pour être égal au nombre de tâches **map** pour redistribuer des données à des

tâches **map** spécifiques. De plus, l'ordre des clés intermédiaires au sein d'une tâche **reduce** après le tri peut être contrôlé par l'utilisateur en fournissant à Hadoop une fonction de comparaison. Pour plus de détails sur le fonctionnement de Hadoop, nous renvoyons le lecteur au livre [Whi09].

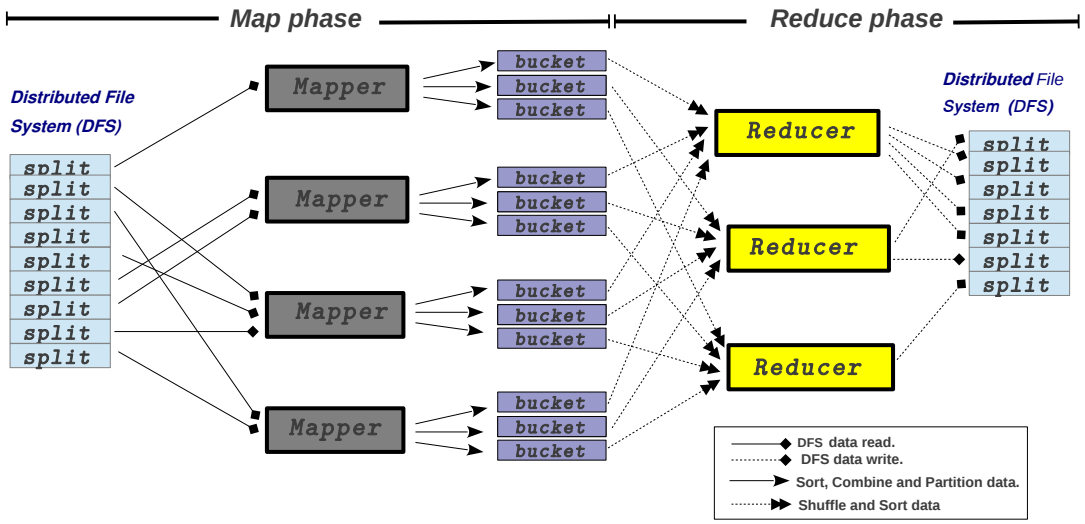



Fig 2.6 : Le fonctionnement de MapReduce. Les buckets correspondent aux données intermédiaires envoyées par les *Mappers* et partitionnées en fonction de la tâche **reduce** destinataire. Nous supposons ici que chaque *Reducer* traite une seule tâche **reduce**.

### b) Apache Spark

Apache Spark [Spa] est un autre framework pouvant entre autre exécuter des programmes MapReduce permettant de traiter les opérations de **map** et **reduce** sur la mémoire d'une grappe de calcul. Apache Spark est connu dans la littérature pour être plus rapide que Hadoop MapReduce pour certains programmes : ceci est lié à l'utilisation des *RDDs* (Resilient Distributed Dataset) [Zah12] permettant de garder en mémoire les données durant les traitements. Cependant, plusieurs aspects doivent être pris en compte pour le choix de la technologie utilisée. Tout d'abord, bien que Spark soit tolérant aux pannes, les mécanismes employés ne sont pas aussi robustes que l'implémentation fournie par Hadoop. En effet, Hadoop réplique les résultats intermédiaires sur plusieurs nœuds de la grappe de calcul à chaque étape de calcul. De plus, nous présentons par la suite des algorithmes extensibles à de très grands ensembles de données, Hadoop reste le plus performant dans ce domaine. Pour ces raisons, nous utilisons Hadoop pour le reste de cette thèse, et ce même si les algorithmes peuvent également être appliqués dans divers paradigmes et environnement de programmation.

### c) Le modèle de coût

Nous utilisons, par la suite, un modèle de coût adaptant le modèle *Bulk Synchronous Processing* (BSP) de Valiant [Val90] pour analyser les coûts de chaque étape des algorithmes proposés. Plus précisément, le modèle décrit le coût de traitement de chaque machine en fonction des coûts de lecture/écriture des données sur le système de fichiers distribués, des coûts de communication entre les nœuds de la grappe de calcul et des coûts des traitements effectués durant les phases de **map** et de **reduce**. En supposant une grappe de calcul de $\mathcal{M}$ nœuds exécutant les tâches **map**, le coût de traitement de toutes les tâches **map** dépendra du nœud le plus lent. Respectivement, nous supposons que $\mathcal{R}$ nœuds au sein de la grappe de calcul traitent les tâches **reduce**.

Plus formellement, nous utilisons les notations suivantes pour décrire les coûts des différentes étapes des algorithmes traitant la jointure entre deux ensembles de données R et S. Nous supposons que l'ensemble de données en entrée est divisé en *blocs* de données, entreposés sur le Hadoop Distributed File System (HDFS) et répliqués sur plusieurs nœuds pour des raisons de fiabilité. Nous distinguons les *blocs* de données, qui correspondent à la représentation physique des données, des *splits* de données, qui représentent la portion logique des données traitées par une fonction **map**. Nous décrivons les notations pour un ensemble de données $\mathrm{T} \in \{\mathrm{R}, \mathrm{S}\}$ :

| | |
|---|---|
| $\|\mathrm{T}\|$ | Le nombre d'enregistrements provenant de l'ensemble de données T, |
| $\lvert\mathrm{T}\rvert$ | Le nombre de *splits* de données de T, |
| $\mathrm{T}_i^{\mathrm{map}}$ | Le *fragment* (ensemble de *splits*) de T traité par le *Mapper* $i$, |
| $\mathrm{T}_i^{\mathrm{red}}$ | Le *fragment* de T traité par le *Reducer* $i$, |
| $\mathcal{M}$ | Le nombre de *Mappers*, |
| $\mathcal{R}$ | Le nombre de *Reducers*, |
| $\mathsf{c}_{\mathbf{r/w}}$ | Le coût de lecture/écriture d'une page de données depuis le Système de Fichiers Distribués (DFS), |
| $\mathsf{c}_{\mathbf{c}}$ | Le coût de communication par page de données, |
| $\chi(\mathrm{T})$ | L'ensemble des valeurs de l'attribut de jointure de l'ensemble de données T, |
| $\mathrm{Hist}(\mathrm{T}_i^{\mathrm{map}})$ | L'histogramme du *fragment* $\mathrm{T}_i^{\mathrm{map}}$ ; c'est-à-dire l'ensemble des tuples correspondant à chaque valeur de l'attribut de jointure, associée à leurs fréquences respectives dans le *fragment*, |
| $\mathrm{Hist}(\mathrm{R} \bowtie \mathrm{S})$ | L'histogramme de la jointure ne contenant que les données pertinentes (cf. Définition 2), |
| $\mathrm{R} \bowtie \mathrm{S}$ | Les résultats de la jointure. |

Ces notations seront étendues au fil des chapitres afin de prendre en compte la jointure par similarité et de s'adapter aux différents algorithmes développés dans le cadre de cette thèse. Pour certains algorithmes de la littérature, des résultats théoriques reposant sur le modèle de coût massivement parallèle (MPC) [KSV10] sont présentés. Ces deux modèles de coûts adaptent le modèle BSP, cependant ce dernier adopte une approche globale en ne tenant compte que de la charge moyenne des nœuds, c'est-à-dire le nombre total de couples clé-valeur émis durant la phase **map** divisé par le nombre de *Reducers*.

### 2.3.2 Un algorithme naïf de jointure sous MapReduce

L'idée est de l'algorithme reposant sur le modèle MapReduce est la même que pour la jointure parallèle (cf. section 2.2). Chaque *split* de données est assigné à un *Mapper*. Les enregistrements sont ensuite redistribués et partitionnés en fonction de leur valeur de l'attribut de jointure sur les *Reducers*. Enfin, les *Reducers* calculent la jointure pour les enregistrements ayant la même valeur de l'attribut de jointure. Cet algorithme se compose d'un seul *job* MapReduce, comme le montre l'exemple présenté par la figure 2.7. Nous représentons les enregistrements de R dans un *split* de données et ceux de S dans un autre par souci de simplicité. Généralement, les enregistrements peuvent être mélangés s'ils possèdent un attribut indiquant leur ensemble de données d'origine.

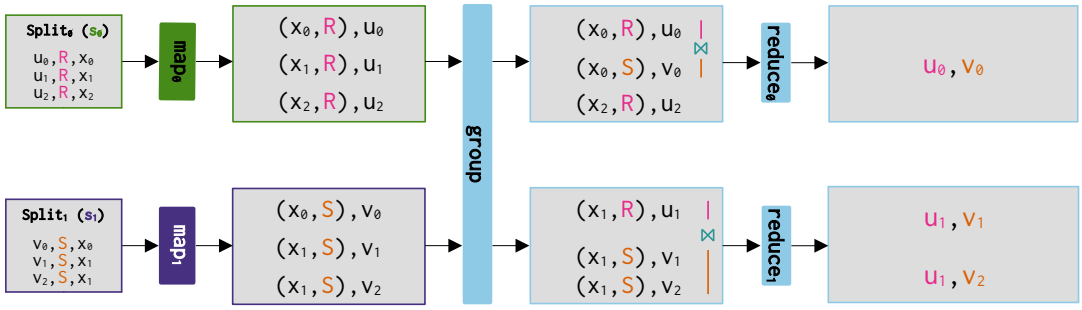



Fig 2.7 : Un exemple d'exécution de l'algorithme naïf de jointure R ⋈ S en utilisant MapReduce.

Plus formellement, l'algorithme se déroule comme suit :

**1.a** Émettre un couple clé-valeur $((x, \mathrm{R}), u)$ pour tous les enregistrements $u$ provenant de R et $((x, \mathrm{S}), v)$ pour tous les enregistrements $v$ appartenant à S, $x$ étant la valeur de l'attribut de jointure ;

**1.b** Partitionner les clés intermédiaires en fonction de la valeur de l'attribut de jointure uniquement, ce qui conduit à ce que l'ensemble des enregistrements ayant une valeur $x$ de l'attribut de jointure soit traité par un même *Reducer*. De plus, après la phase de tri, les enregistrements appartenant à la relation R seront traités avant les enregistrements de la relation S ;

**1.c** Stocker en mémoire les enregistrements de R pour une valeur de l'attribut de jointure donnée ;

**1.d** Émettre un couple d'enregistrements $(u, v) \in \mathrm{R} \times \mathrm{S}$ en sortie pour tout enregistrement $u \in \mathrm{R}$ stocké et $v \in \mathrm{S}$ reçu.

### a) Analyse de l'algorithme naïf de jointure sous MapReduce

Le modèle de coût utilise la notation $\mathcal{O}(\dots)$, qui masque uniquement de petits facteurs constants : ils dépendent uniquement de l'implémentation du programme, mais pas des ensembles de données en entrée ni des paramètres des machines composant la grappe. L'ensemble du traitement est distribué sur les nœuds de la grappe.

Sur le nœud $i$, le *Mapper* lit son fragment à un coût $\mathsf{c}_{\mathbf{r/w}} * (|\mathrm{R}_i^{\mathrm{map}}| + |\mathrm{S}_i^{\mathrm{map}}|)$. Ensuite, il traite chaque enregistrement à un coût constant, aboutissant à un coût total de $\|\mathrm{R}_i^{\mathrm{map}}\| + \|\mathrm{S}_i^{\mathrm{map}}\|$. Enfin, il trie localement les couples clé-valeur émis pour un coût total représenté par le terme $\|\mathrm{R}_i^{\mathrm{map}}\| * \log(\|\mathrm{R}_i^{\mathrm{map}}\|) + \|\mathrm{S}_i^{\mathrm{map}}\| * \log(\|\mathrm{S}_i^{\mathrm{map}}\|)$. Les données sont ensuite envoyées aux *Reducers* au coût de $\mathsf{c}_{\mathbf{c}} * (|\mathrm{R}_i^{\mathrm{map}}| + |\mathrm{S}_i^{\mathrm{map}}|)$. Par conséquent, cette phase **map** nécessite au plus :

$$\begin{aligned}\mathsf{Time}(\mathsf{Naive.Mapper}) = \mathcal{O}\Big( &\max_{i=0}^{\mathcal{M}} \mathsf{c}_{\mathbf{r/w}} * (|\mathrm{R}_i^{\mathrm{map}}| + |\mathrm{S}_i^{\mathrm{map}}|) + \|\mathrm{R}_i^{\mathrm{map}}\| + \|\mathrm{S}_i^{\mathrm{map}}\| \\ &+ \|\mathrm{R}_i^{\mathrm{map}}\| * \log(\|\mathrm{R}_i^{\mathrm{map}}\|) + + \|\mathrm{S}_i^{\mathrm{map}}\| * \log(\|\mathrm{S}_i^{\mathrm{map}}\|) + \mathsf{c}_{\mathbf{c}} * (|\mathrm{R}_i^{\mathrm{map}}| + |\mathrm{S}_i^{\mathrm{map}}|)\Big).\end{aligned}$$

Chaque *Reducer* reçoit ensuite son fragment de données et calcule la jointure. Le coût de cette étape est représenté par le terme $\|\mathrm{R}_i^{\mathrm{red}}\| + \|\mathrm{S}_i^{\mathrm{red}}\| + \|\mathrm{R}_i^{\mathrm{red}} \bowtie \mathrm{S}_i^{\mathrm{red}}\|$ pour un *Reducer* $i$. Enfin, les résultats de la jointure sont produits en sortie. Le terme $\mathsf{c}_{\mathbf{r/w}} * |\mathrm{R}_i^{\mathrm{red}} \bowtie \mathrm{S}_i^{\mathrm{red}}|$ correspond au coût de cette étape. Globalement, cette phase aura le coût suivant :

$$\mathsf{Time}(\mathsf{Naive.Reducer}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{R}} \|\mathrm{R}_i^{\mathrm{red}}\| + \|\mathrm{S}_i^{\mathrm{red}}\| + \|\mathrm{R}_i^{\mathrm{red}} \bowtie \mathrm{S}_i^{\mathrm{red}}\| + \mathsf{c}_{\mathbf{r/w}} * |\mathrm{R}_i^{\mathrm{red}} \bowtie \mathrm{S}_i^{\mathrm{red}}|\Big).$$

Le traitement de la jointure aura donc le coût suivant :

$$\mathsf{Time}(\mathsf{Naive.}) = \mathsf{Time}(\mathsf{Naive.Mapper}) + \mathsf{Time}(\mathsf{Naive.Reducer}).$$

### 2.3.3 Les problèmes de déséquilibre de la jointure sous MapReduce

Cet algorithme connaît les mêmes limites et déséquilibres que la jointure parallèle présentée dans la section 2.2. Nous revenons plus en détails sur ces déséquilibres dans le contexte de Hadoop. Lorsque les entrées sont entreposées sur le système de fichiers distribués, ils sont découpés en *blocs* (128MB par défaut), chaque *bloc* est répliqué (trois fois par défaut) éventuellement sur plusieurs armoires (racks) de machines. Généralement, une armoire correspond à un ensemble de machines connectées à un même réseau. La stratégie de réplication par défaut consiste à placer la première réplique sur un nœud d'une armoire aléatoire, puis les deux autres copies sur deux autres nœuds, si possible, d'une armoire différente. Lors de l'exécution d'un *job*, les sorties générées par les tâches **reduce** sont stockées à la fois sur les nœuds traitant chaque tâche, et de manière similaire, sur une armoire différente. Lorsque l'entrée représente un nombre important de *blocs*, le nombre de *blocs* par machine est équilibré avec une forte probabilité, ce qui permet d'éviter le **TPS**. Lors de l'exécution d'un *job*, Hadoop attribue en priorité les tâches **map** correspondant aux *splits* stockés sur les disques locaux des machines de traitement, puis dans l'ordre de priorité vient les *splits* stockés sur la même armoire et enfin sur d'autres armoires. Cette priorité à la localité permet de réduire grandement l'usage du réseau, même dans le cas de masses de données très volumineuses.

Il faut distinguer un cas particulier pour le **HS** lorsque l'entrée correspond à la sortie d'un *job* précédent comprenant plus de tâches **reduce** que de places dans la grappe. Dans ce cas, la répartition des tâches **reduce** est normalement proportionnelle à la puissance des machines ; ceci est dû au fait que les machines les plus puissantes ont traité plus de tâches que les autres et ont stocké les résultats sur leurs disques locaux. Dans ce cas, il est possible que l'attribution des tâches **map** suive globalement le même chemin en raison de la priorité donnée aux traitements en local. Dans les autres cas, la répartition des *blocs* est équilibrée et un **HS** peut se produire sur une grappe hétérogène. Hadoop propose une stratégie pour remédier à ce problème ; les machines ayant terminé leur travail peuvent relancer une tâche de manière spéculative, c'est-à-dire une tâche qui est déjà en cours de traitement sur un autre nœud. Cette stratégie peut aider à diminuer le temps d'exécution global, et améliorer les performances si un **CS** survient. Toutefois, cette stratégie reste insuffisante à résoudre tous les facteurs de déséquilibres

dans un environnement hétérogène. Zaharia et al. [Zah08] ont pointé plusieurs hypothèses implicites faites par Hadoop qui ne sont pas vérifiées en pratique. En particulier, le lancement de tâches spéculatives peut engendrer des coûts supplémentaires en raison de l'utilisation accrue du réseau.

Enfin, jusqu'à présent, nous avons considéré que chaque tâche représentait approximativement le même temps d'exécution. Cette hypothèse n'est pas réaliste lorsqu'un déséquilibre se produit, que ce soit un **JPS**, ou en raison d'un **AVS** conduisant à un **RS** sur les *Reducers*. Pour de très grands ensembles de données, un très fort déséquilibre peut fortement impacter les performances du traitement de la jointure puisque chaque valeur de l'attribut sera traitée par une seule tâche **reduce** comme représenté par la figure Fig. 2.8. Il faut noter que l'attribution des tâches **reduce** aux *Reducers* ne peut pas être déterminée a priori dans le cadre de Hadoop. De plus, lors de l'étape de jointure **1.c**, les enregistrements de la relation R doivent être stockés en mémoire d'une tâche **reduce** sans aucune garantie que la mémoire réservée soit suffisante, ce qui peut conduire à l'échec du traitement de cette tâche. Ce cas ne peut pas être résolu en relançant la tâche, et doit être pris en compte algorithmiquement.

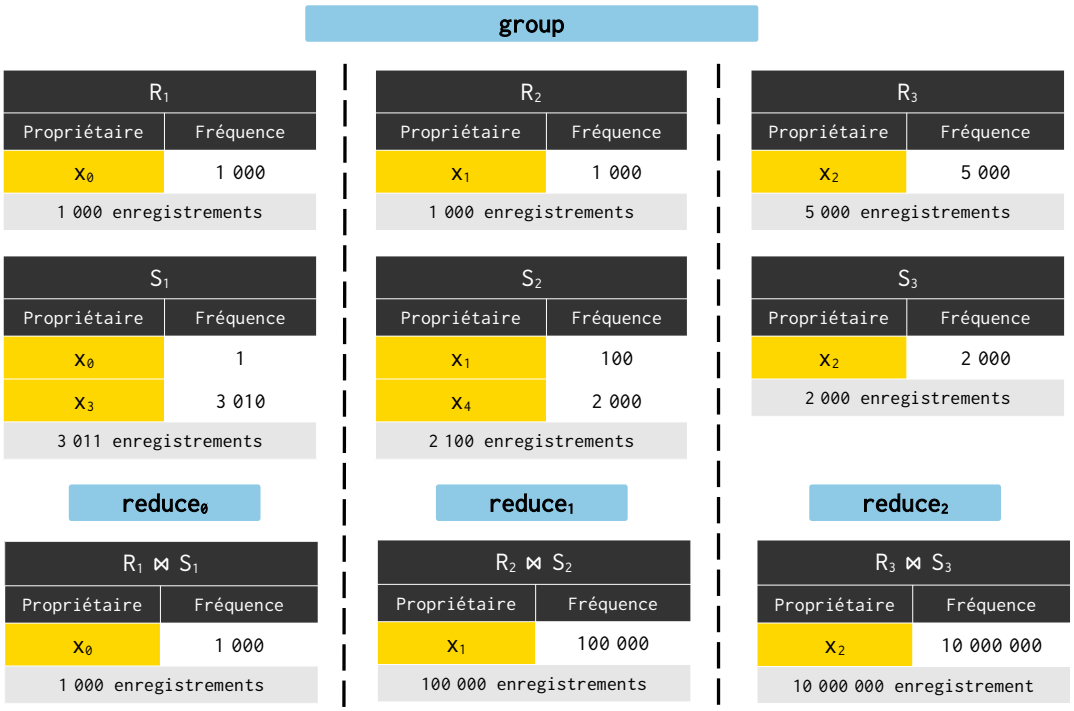



Fig 2.8 : Un exemple d'exécution de trois tâches **reduce** dans le cas d'un déséquilibre intrinsèque pour une jointure R ⋈ S en utilisant MapReduce.

### 2.3.4 *MRFA-join* : MapReduce Frequency Adaptive Join

Pour éviter ces situations, l'algorithme *MRFA-join* [HBL14] calcule la fréquence des valeurs de l'attribut de jointure avant de traiter la jointure. La fréquence de chaque valeur de l'attribut de jointure est utilisée pour déterminer si une valeur est trop fréquente pour être traitée au sein d'une seule tâche **reduce**. Si tel est le cas, l'algorithme répartit la charge de travail sur plusieurs tâches afin de garantir un équilibre dans la répartition des charges sur l'ensemble tous les nœuds de la grappe. Par ailleurs, les coûts de communication sont réduits en ne transmettant que les données pertinentes. L'algorithme *MRFA-join* nécessite deux phases de MapReduce, qui sont les suivantes :

1. Calculer l'histogramme de la jointure, c'est-à-dire les fréquences globales de chaque valeur de l'attribut de jointure,
2. Calculer la jointure en utilisant des schémas de communication spécifiques lorsque les enregistrements correspondant à une valeur de l'attribut de jointure sont trop nombreux pour être traités au sein d'une seule tâche **reduce**.

#### a) Les schémas de communication dans *MRFA-join*

Pour décider si les enregistrements correspondant à une valeur de l'attribut de jointure sont trop nombreux pour être traités au sein d'une seule tâche **reduce**, un paramètre supplémentaire, noté $\mathbf{f}_{max}$, est nécessaire. Ce paramètre traduit le nombre d'enregistrements qu'une tâche **reduce** peut stocker en mémoire durant l'étape de jointure. Lorsque les enregistrements correspondant à une valeur de l'attribut de jointure sont trop nombreux, ils seront partitionnés en plusieurs *blocs*, de telle sorte que chaque *bloc* puisse être stocké en mémoire et transmis à plusieurs tâches **reduce**. Soit $x$ une valeur de l'attribut de jointure présente dans R et S, c'est-à-dire que $x$ appartient à $\chi(\mathrm{R}) \cap \chi(\mathrm{S})$. Les schémas de communication redistribueront les enregistrements en fonction de l'un des trois cas suivants :

a. $\mathbf{f}_x^{\mathrm{R}} < \mathbf{f}_{max}$ et $\mathbf{f}_x^{\mathrm{S}} < \mathbf{f}_{max}$ : les enregistrements sont peu nombreux dans les deux relations et ils peuvent être traités par une seule tâche **reduce** ; ils sont donc redistribués en utilisant une simple méthode de hachage,

b. $\mathbf{f}_x^{\mathrm{R}} \geqslant \mathbf{f}_{max}$ et $\mathbf{f}_x^{\mathrm{R}} \geqslant \mathbf{f}_x^{\mathrm{S}}$ : les enregistrements de la relation R sont trop nombreux pour être traité par une seule tâche **reduce** et ils sont plus nombreux que ceux de la relation S ; ils sont donc distribués en utilisant une méthode de partition/duplication sur plusieurs tâches comme illustrée par la figure Fig. 2.9,

c. $\mathbf{f}_x^{\mathrm{S}} \geqslant \mathbf{f}_{max}$ et $\mathbf{f}_x^{\mathrm{R}} < \mathbf{f}_x^{\mathrm{S}}$ : à l'inverse, les enregistrements de la relation S sont plus nombreux que dans R.

Ce traitement spécifique garantit une extensibilité et une insensibilité aux effets du déséquilibre de la charge, y compris dans le cas d'ensembles de données très fortement déséquilibrés.

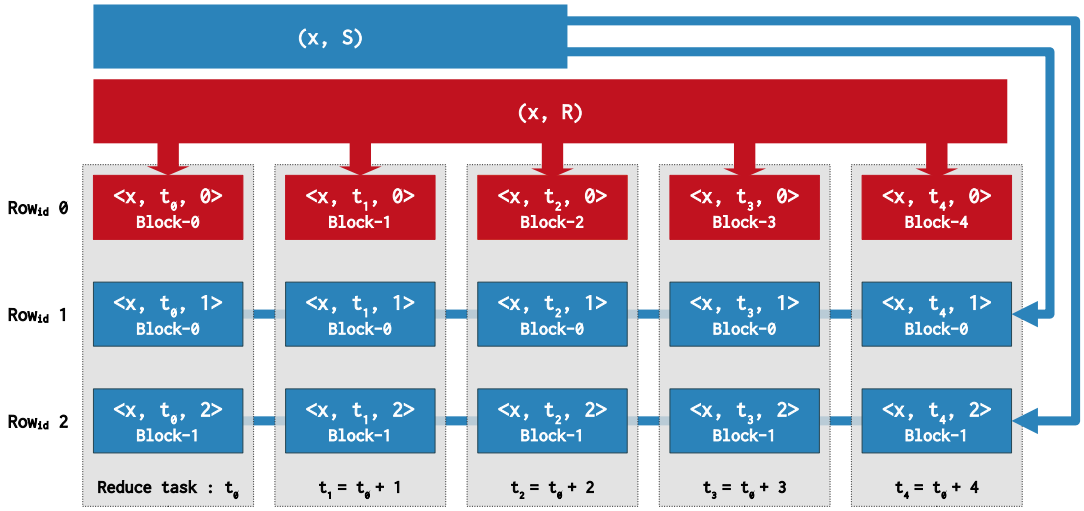



Fig 2.9 : Les schémas de communication dans le cas d'une jointure entre deux ensembles de données où le nombre d'enregistrements est trop important pour être traité au sein d'une seule tâche **reduce**. Dans cet exemple, les enregistrements correspondant à la valeur $x$ de l'attribut de jointure de la relation R sont partitionnés en cinq blocs et transmis à cinq tâches différentes, tandis que les enregistrements de la relation S sont dupliqués et transmis sur ces cinq tâches. Les schémas de communication générés sont rangés en lignes et en colonnes. Chaque cellule correspond à un bloc. Chaque colonne correspond aux données transmises à une tâche **reduce**. Ces tâches sont identifiées à partir d'un nombre entier aléatoire $t_0$ qui peut être dérivé de la valeur de l'attribut de jointure. Les blocs partitionnés (rouge) sont stockés en mémoire pour calculer la jointure avec les blocs dupliqués (bleu).

Pour s'assurer que les blocs soient ordonnés correctement lors l'exécution des phases de **reduce**, des couples clé-valeur adaptés sont utilisés. Les clés sont composées de la valeur de l'attribut de jointure, de l'identifiant de la colonne et de l'identifiant de la ligne. Les couples sont ensuite redirigés par la fonction MapReduce **partition** à l'aide de l'identifiant de colonne. Pour une tâche **reduce**, la jointure est calculée à l'aide de l'algorithme suivant en utilisant les schémas de communication spécifiques :

1. Stocker en mémoire le bloc partitionné,
   c'est-à-dire le bloc de données correspondant à la ligne $\text{Row}_{\text{id}}0$,

2. Calculer la jointure entre le bloc stocké et les blocs dupliqués,
   c'est-à-dire la jointure entre le bloc correspondant à la ligne 0 et ceux des lignes $\text{Row}_{\text{id}}$ $i : i \geq 1$.

Ces schémas de communication permettent à l'algorithme *MRFA-join* d'éviter les déséquilibres **AVS** et **JPS** et de garantir un équilibre dans la répartition des charges, même dans le cas de très grands ensembles de données extrêmement déséquilibrés, ce qui améliore grandement les performances en comparaison à l'algorithme naïf. Cela permet également de garantir que la mémoire réservée au niveau des *Reducers* soit suffisante pour traiter l'étape de jointure.

### b) *MRFA-join* : un exemple détaillé

La figure Fig. 2.10 illustre le calcul de l'histogramme de la jointure ne contenant que les données pertinentes. Le coût de cette étape est par définition inférieur au coût de traitement de la jointure dans le cadre de l'algorithme naïf puisque seule la fréquence est calculée, ce qui ne nécessite pas d'envoyer l'enregistrement complet.

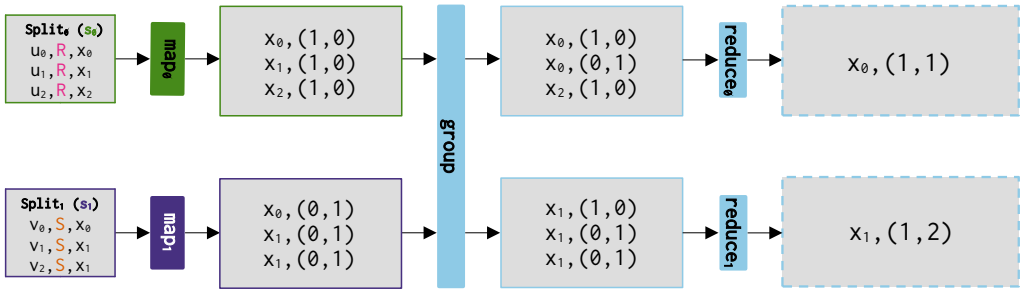



Fig 2.10 : *MRFA-join*.1 : un exemple d'exécution de l'étape du calcul de l'histogramme de la jointure.

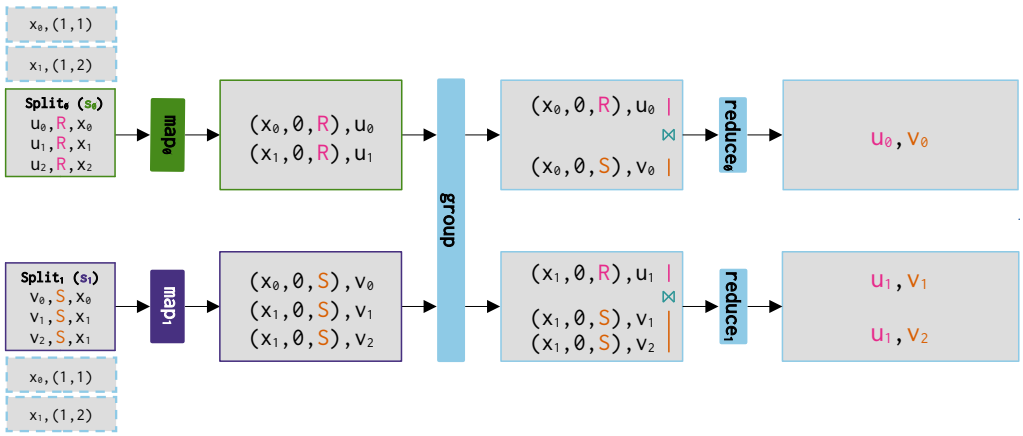



Fig 2.11 : *MRFA-join*.2 : un exemple d'exécution de l'étape de jointure. Les schémas de communication spécifiques ne sont pas utilisés dans cet exemple puisque la valeur $x_1$ n'apparaît que trois fois dans l'entrée. La valeur du 'reducerId' des clés transmises est donc mise à 0 et la valeur du 'rowId' prend l'ensemble de données d'entrées pour trier les données intermédiaires.

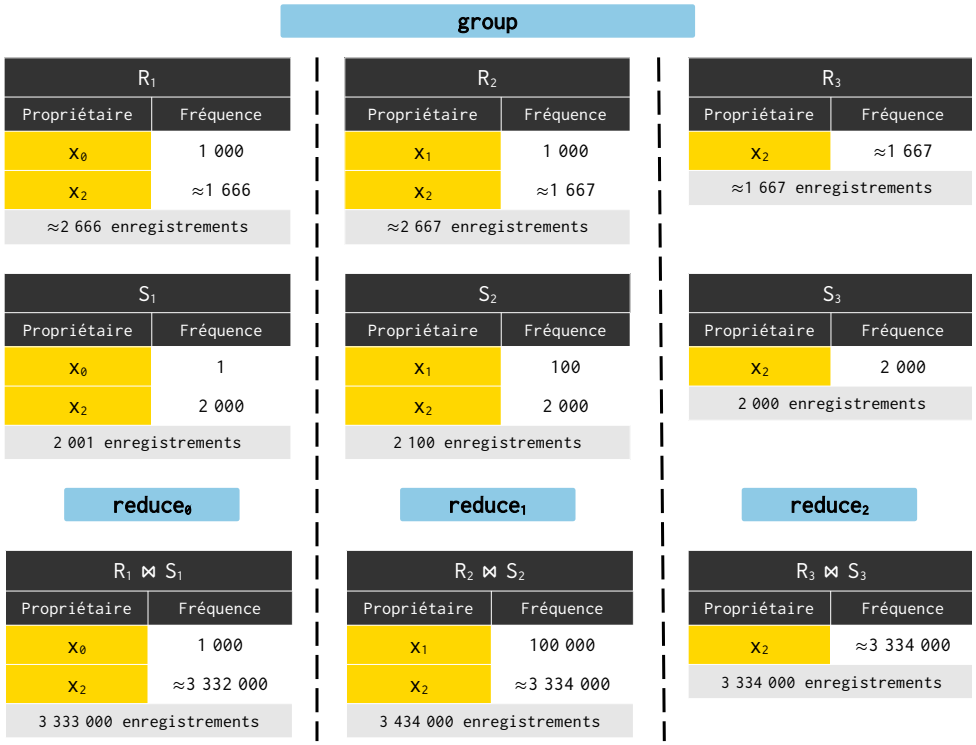



Fig 2.12 : *MRFA-join* : un exemple de traitement d'une jointure R ⋈ S en reprenant les exemples précédent (cf. Fig. 2.3) comportant un déséquilibre intrinsèque aux données. En paramétrant le nombre d'enregistrements maximum qu'un *Reducer* peut stocker durant l'étape de jointure à $\mathbf{f}_{max} = 2000$. Les enregistrements correspondant à la valeur $x_0$ sont tous transmis à la tâche 0, puisque les fréquences dans R et S sont inférieures à $\mathbf{f}_{max}$. Le raisonnement est le même pour les enregistrements correspondant à la valeur $x_1$, qui sont transmis à la tâche 1. Pour la valeur $x_2$, les enregistrements de R sont divisés en trois blocs et transmis aux trois tâches. Les enregistrements provenant de S sont dupliqués sur ces tâches. Si la fréquence de la valeur $x_2$ dans l'ensemble S avait été comprise entre $\mathbf{f}_{max} + 1$ et $\mathbf{f}^{\mathrm{R}}_{x_2}$, les enregistrements de S auraient été divisés en plusieurs blocs et auraient été transmis aux mêmes tâches en plusieurs fois.

La figure Fig. 2.11 présente l'exécution de la seconde étape. Elle consiste à utiliser l'histogramme de la jointure pour décider si chaque valeur de l'attribut de jointure peut être traitée au sein d'une seule tâche **reduce**. Pour ce faire, avant de commencer à lire les enregistrements, chaque tâche **map** met en mémoire l'histogramme global. Bien que cet histogramme soit très petit comparé à la taille des relations R et S, cette étape présente une limite puisque sa taille est liée au nombre de valeurs de l'attribut de jointure présentes dans les relations R et S, c'est-à-dire $\chi(\mathrm{R}) \cap \chi(\mathrm{S})$. Pour de très grands ensembles de données, il n'y a aucune garantie qu'il puisse tenir mémoire. Les données pertinentes sont ensuite envoyées aux *Reducers* en utilisant les schémas de communication. Les résultats de la jointure sont enfin produits en sortie des *Reducers*.

D'un point de vue plus large, la figure Fig. 2.12 présente la répartition des enregistrements sur trois tâches **reduce** en reprenant l'exemple présenté précédemment dans

les figures Figs. 2.3 et 2.8. L'utilisation de schémas de communication spécifiques sur la valeur $x_2$ permet de répartir équitablement les enregistrements correspondants et le traitement de la jointure sur les nœuds de la grappe, tout en s'assurant que la mémoire allouée soit suffisante. Pour des ensembles de données très volumineux, on s'attend à ce que la répartition des charges soit équilibrée au sein des tâches **reduce**, ce qui prévient les divers **déséquilibres** inhérents à la jointure parallèle.

### c) Analyse de l'algorithme *MRFA-join*

Une analyse de chaque étape est détaillée dans l'article [HBL14], et les notations employées sont presque identiques aux nôtres ; nous renvoyons donc le lecteur vers l'article pour plus de précision. Cette analyse conclut que l'algorithme *MRFA-join* est asymptotiquement optimal par rapport à l'algorithme naïf lorsque l'équation suivante est satisfaite :

$$\|\mathrm{Hist}(\mathrm{R} \bowtie \mathrm{S})\| \leq \max\left( \max_{i=0}^{\mathcal{M}} \left( \|\mathrm{R}_i^{\mathrm{map}}\| * \log\left(\|\mathrm{R}_i^{\mathrm{map}}\|\right) + \|\mathrm{S}_i^{\mathrm{map}}\| * \log\left(\|\mathrm{S}_i^{\mathrm{map}}\|\right) \right), \max_{i=0}^{\mathcal{R}} \left( \|\mathrm{R}_i^{\mathrm{red}} \bowtie \mathrm{S}_i^{\mathrm{red}}\| \right) \right).$$

Cette équation tient, car l'étape préliminaire nécessaire au calcul de l'histogramme est, par définition, moins coûteuse que le traitement de la jointure. Toutefois, lors de l'utilisation de l'histogramme, cet algorithme est limité par la taille de l'entrée qu'il peut traiter. En effet, les tâches **map** de l'étape de jointure doivent avoir l'information complète en mémoire, c'est-à-dire l'histogramme de la jointure contenant $\|\mathrm{Hist}(\mathrm{R} \bowtie \mathrm{S})\|$ entrées. Pour de très grands ensembles de données, il se peut que l'histogramme de la jointure soit plus grand qu'un *fragment* traité par un *Mappers*.

## 2.4 Conclusion

Dans ce chapitre, nous avons présenté l'opération de jointure ainsi que les algorithmes s'exécutant sur des architectures séquentielles et parallèles. Enfin, nous avons introduit le modèle MapReduce, le modèle de coût que nous utilisons tout au long de cette thèse, et les principaux algorithmes de jointure. En résumé,

- Le modèle MapReduce offre la possibilité de traiter de très grands volumes de données. Toutefois, l'**Heterogeneity Skew** et le **Concurency Skew** peuvent survenir.

- Le traitement de la jointure en utilisant le paradigme MapReduce peut théoriquement être extensible et permettre de traiter de très grandes masses de données. Cependant, des **déséquilibres** peuvent survenir à chaque étape de la jointure, ce qui limite fortement l'extensibilité et les performances des algorithmes. Dans le pire des cas, le traitement de la jointure peut échouer puisque les enregistrements d'une des deux relations doivent être stockés en mémoire. Une étape préliminaire est alors nécessaire pour déterminer les valeurs de l'attribut de jointure pouvant poser des problèmes. Ce traitement, bien que peu coûteux, présente une limite : l'ensemble de l'histogramme doit tenir en mémoire pour être utilisé. Ce traitement permet également de réduire drastiquement les coûts de communication en ne transmettant que les données pouvant produire un résultat.

# Chapitre 3

# L'évaluation de la jointure par similarité

Dans ce chapitre, nous introduisons l'opération de jointure par similarité et nous évaluons les différents algorithmes et techniques de la littérature. Nous séparons ces algorithmes en deux catégories en fonction de la complétude du résultat.



## 3.1 Jointure par similarité

La jointure par similarité consiste à retrouver l'ensemble des couples d'objets ayant une distance inférieure à un seuil donné par l'utilisateur. Dans un contexte de BigData, le but de cette opération est de créer des liens entre les entités similaires à partir des immenses volumes de données collectées. Par exemple, Spotify doit traiter chaque jour un grand nombre de données provenant de ses utilisateurs, telles que les titres écoutés, les artistes préférés, les playlists, etc. Le traitement de ces données permet de proposer du contenu personnalisé pour chaque client de Spotify.

Dans cette thèse, nous allons considérer deux types de jointures par similarité : la jointure entre deux ensembles de données, appelée jointure $R \bowtie_\lambda S$, et la jointure au sein d'un seul ensemble de données, appelée auto-jointure.

**Définition 4.** Soient R et S deux ensembles de données. Étant donné une distance $\mathcal{D}$

et un seuil $\lambda$ fixé par l'utilisateur, la jointure par similarité $\mathrm{R} \bowtie_\lambda \mathrm{S}$ est définie par :

$$\mathrm{R} \bowtie_\lambda \mathrm{S} = \{(u, v) \in \mathrm{R} \times \mathrm{S} \mid \mathcal{D}(u, v) \leq \lambda\}$$

**Définition 5.** Soit $\Gamma$ un ensemble de données. L'auto-jointure par similarité est définie pour tout ordre total de $\Gamma$ par :

$$\Gamma \bowtie_\lambda \Gamma = \{(u, v) \in \Gamma^2 \mid u \prec v \wedge \mathcal{D}(u, v) \leq \lambda\}$$

Toutes les techniques utilisées dans cette thèse conviennent pour ces deux types de jointures, et seulement quelques changements sont nécessaires pour le passage de l'auto-jointure vers une jointure $\mathrm{R} \bowtie_\lambda \mathrm{S}$ et inversement. L'enjeu réside dans le développement de techniques permettant de traiter efficacement cette opération sur une masse de données très importante en utilisant des systèmes à très grande échelle et pour divers types d'objets.

### 3.1.1 Espace métrique

Pour étudier ces différents objets, la jointure par similarité nécessite qu'une distance soit donnée par l'utilisateur, ce qui permet de déterminer si deux objets sont considérés comme proches ou similaires. Dans cette thèse, nous nous sommes intéressés à plusieurs distances pour différents types d'objets. Pour rappel, un espace métrique est défini de la manière suivante :

**Définition 6.** Soit $\mathbf{M} = (\mathbf{E}, \mathcal{D})$ un espace métrique, où $\mathbf{E}$ est un ensemble non-vide et $\mathcal{D}$ est une distance telle que $\mathcal{D} : \mathbf{E} \times \mathbf{E} \to \mathbb{R}_+$ vérifiant les propriétés suivantes :

- $\forall u, v \in \mathbf{E},\ \mathcal{D}(u, v) = \mathcal{D}(v, u)$ (Symétrie),
- $\forall u, v \in \mathbf{E},\ \mathcal{D}(u, v) = 0 \iff u = v$ (Identité),
- $\forall u, v, w \in \mathbf{E},\ \mathcal{D}(u, v) \leq \mathcal{D}(u, w) + \mathcal{D}(w, v)$ (Inégalité triangulaire).

Pour la suite, nous nous concentrons principalement sur les distances généralement utilisées dans la littérature pour déterminer la similarité sur des ensembles, des trajectoires et des séquences. Le choix de la distance utilisée dépend généralement du cas d'utilisation et du résultat attendu. Pour certains cas de figure, il est parfois utile de recourir à des fonctions de similarité ne satisfaisant pas nécessairement tous les axiomes d'un espace métrique.

### a) La distance de Jaccard : une distance pour les ensembles

Nous employons la similarité de Jaccard pour comparer des ensembles, elle trouve de nombreuses applications dans la littérature, notamment le nettoyage de données [CGK06], la résolution d'entités [DSD02], la détection de textes similaires [TSP08 ; Wan14a] et le filtrage collaboratif [BMS07]. Elle est également utilisée pour réduire le nombre de paires candidates pour les jointures par similarité de séquences basées sur la distance de Levenshtein [AGK06]. La similarité de Jaccard entre deux ensembles tient compte de la taille de l'intersection et de la taille de l'union pour des ensembles.

**Définition 7.** Soient $u$ et $v$ deux ensembles finis formés à partir d'un univers $\mathcal{U}$ et en notant $\|\cdot\|$ pour désigner la cardinalité d'un ensemble, la similarité de Jaccard est défini par :

$$\mathsf{Jaccard}(u, v) = \frac{\|u \cap v\|}{\|u \cup v\|}$$

Notons que par construction $0 \leq \mathsf{Jaccard}(u, v) \leq 1$, et $\mathsf{Jaccard}(u, v) = 1$ si et seulement si $u$ et $v$ sont égaux. Pour satisfaire les propriétés de l'espace métrique, nous définissons la distance de Jaccard correspondante.

**Définition 8.** La distance de Jaccard entre deux ensembles $u$ et $v$ est définie par :

$$\mathcal{D}_{\mathsf{Jaccard}}(u, v) = 1 - \mathsf{Jaccard}(u, v)$$

La distance de Jaccard peut être décidée en temps linéaire par rapport à la cardinalité des ensembles. Pour tester rapidement si un couple d'ensembles a une distance inférieure à un seuil donné, une procédure terminant aussitôt que le seuil ne peut être atteint est généralement utilisée dans la littérature [MAB16]. L'idée est de trier les ensembles préalablement et de déterminer pour chaque couple d'ensembles le nombre minimum d'éléments qu'ils doivent avoir en commun pour être similaire. Plus précisément, cela revient à rechercher les couples d'ensembles satisfaisants :

$$\|u \cap v\| \;\geq\; \left\lceil \frac{1-\lambda}{2-\lambda} * (\|u\| + \|v\|) \right\rceil$$

Comme les ensembles sont triés au préalable, il est suffisant de parcourir les deux ensembles simultanément et de compter les éléments en commun. Lorsque le nombre d'éléments non évalués ne permet plus de satisfaire l'équation précédente, le parcours peut s'arrêter.

Une illustration d'une jointure par similarité sur des ensembles est présentée dans la figure Fig. 3.1. Les couples similaires sont déterminés en utilisant la distance de Jaccard et un seuil. Nous passerons en revue les techniques de la littérature pour traiter

la jointure par similarité efficacement plus tard. Par souci de clarté, les éléments des ensembles sont appelés jetons par la suite, et ce, pour éviter de les appeler les éléments d'un enregistrement d'un ensemble de données R ou S.

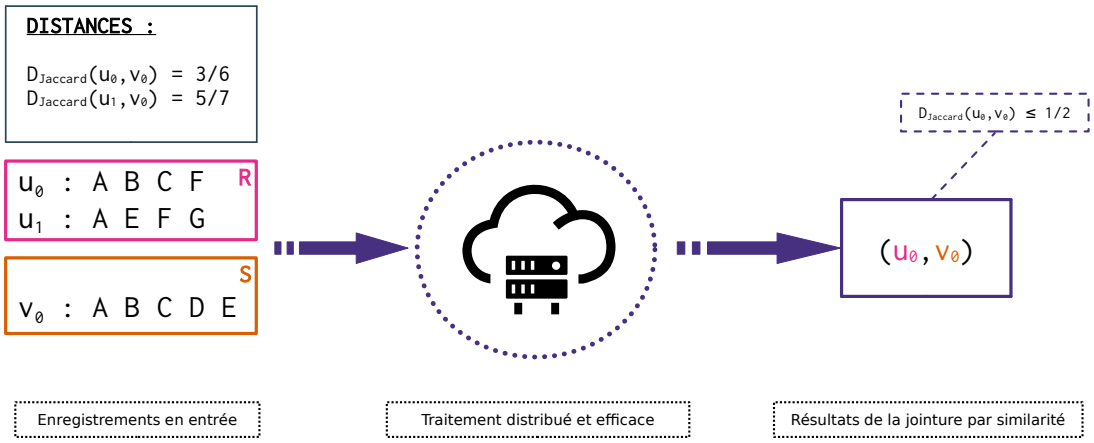



Fig 3.1 : Un exemple de jointure par similarité entre deux relations composées d'ensembles, en utilisant la distance de Jaccard, et en fixant le seuil à $\lambda = 1/2$.

### b) La distance de Fréchet : une distance pour les trajectoires

Dans le cas des trajectoires, nous utilisons la distance de Fréchet. Cette distance est souvent expliquée par la métaphore suivante : un homme tient son chien en laisse, tous deux marchent sur des trajectoires finies. L'homme et le chien peuvent varier leur vitesse, mais ne peuvent pas revenir en arrière. La distance continue de Fréchet est la longueur minimale de la laisse pour relier l'homme à son chien pendant toute la durée du trajet. La distance discrète de Fréchet est une approximation de la distance continue qui ne prend en compte que la longueur de la laisse lorsque l'homme et son chien sont situés sur les sommets de leurs trajectoires respectives. Dans la littérature, cette distance a été utilisée pour la reconnaissance de l'écriture manuscrite [SKB07], la prédiction de structures de protéines [WZ13] et principalement pour l'étude d'objets en mouvement [Kon17].

Formellement, considérons un espace à $d$ dimensions $\mathbb{R}^d$ muni de la distance euclidienne notée $\mathcal{D}_{\mathsf{E}}$. Une trajectoire $u$ est définie par ses sommets $u_1 \dots u_n$, où chaque sommet appartient à $\mathbb{R}^d$.

**Définition 9.** Soit $\mathcal{T}$ l'ensemble de ces trajectoires. Soient $u$ et $v$ deux trajectoires appartenant à $\mathcal{T}$, définies par leurs sommets $u = u_1 \dots u_n$ et $v = v_1 \dots v_m$ respectivement.

La distance discrète de Fréchet est définie par :

$$\mathcal{D}_{\mathsf{Fréchet}}(u,v) = \max \begin{cases} \mathcal{D}_{\mathsf{E}}(u_1, v_1), \\ \min \begin{cases} \mathcal{D}_{\mathsf{Fréchet}}(u_2 \dots u_n, v), & \|u\| > 1 \\ \mathcal{D}_{\mathsf{Fréchet}}(u, v_2 \dots v_m), & \|v\| > 1 \\ \mathcal{D}_{\mathsf{Fréchet}}(u_2 \dots u_n, v_2 \dots v_m), & \|u\| > 1 \wedge \|v\| > 1 \\ \mathcal{D}_{\mathsf{E}}(u_1, v_1) & \|u\| = 1 \wedge \|v\| = 1 \end{cases} \end{cases}$$

Dans la littérature, il est prouvé que la distance discrète de Fréchet entre deux trajectoires quelconques ne peut être décidée en temps strictement sous-quadratique par rapport au nombre de sommets des trajectoires sous l'Hypothèse du Temps Exponentiel Fort (SETH) [Bri14]. En conséquence, les algorithmes les plus efficaces sont quadratiques [Buc17].

Plus récemment, lors de la compétition "ACM SIGSPATIAL Cup 2017", le défi était de développer un algorithme efficace pour interroger une base de données de trajectoires. L'utilisateur fournissait à l'algorithme une liste de requêtes consistant en une trajectoire et un entier correspondant au seuil de la distance de Fréchet. L'algorithme devait pour chaque requête retrouver dans la base de données, l'ensemble des trajectoires similaires, c'est-à-dire l'ensemble des trajectoires ayant une distance inférieure au seuil fourni. Pour chaque trajectoire générée en sortie, la distance de Fréchet est calculée et comparée par rapport au seuil donné. Le gagnant de cette compétition a présenté un algorithme utilisant plusieurs heuristiques accélérant le calcul de la distance [WO18].

Dans la littérature, d'autres améliorations ont été apportées en prenant certaines hypothèses réalistes sur la forme des trajectoires en entrée pour vérifier en temps quasi-linéaire la distance de Fréchet [DHW12]. Une procédure utilisant ce concept de simplification est décrite dans [CDS19] ; nous utilisons cette procédure pour accélérer les calculs de la distance de Fréchet.

Les figures Fig. 3.2 et Fig. 3.3 présentent intuitivement le traitement qu'une jointure par similarité de trajectoires doit effectuer entre un et deux ensembles de données, en utilisant la distance de Fréchet pour déterminer les couples de trajectoires similaires.

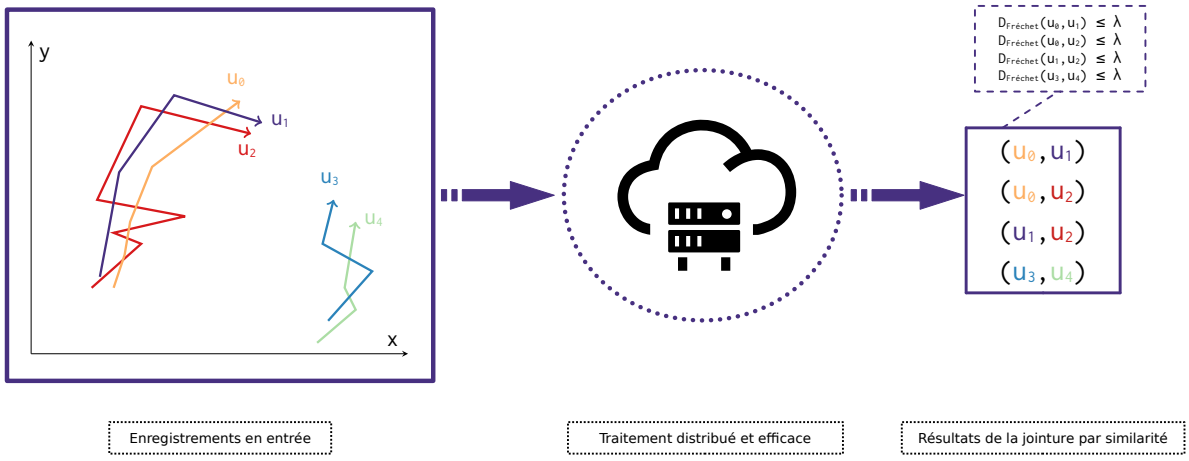



Fig 3.2 : Un exemple d'auto-jointure par similarité de trajectoires en utilisant la distance de Fréchet pour un seuil λ.

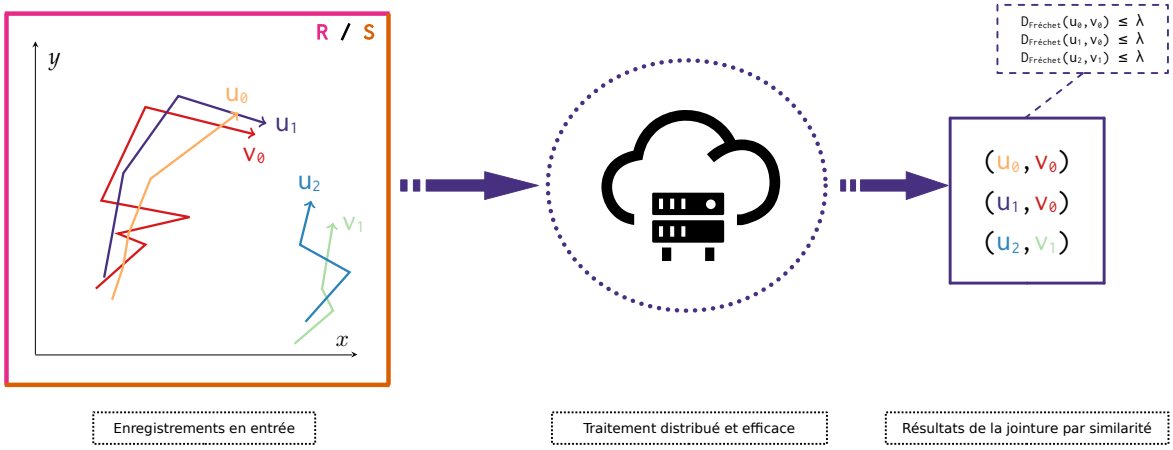



Fig 3.3 : Un exemple de jointure par similarité de trajectoires provenant de deux ensembles de données R et S en utilisant la distance de Fréchet pour un seuil λ.

### c) La Déformation temporelle dynamique (DTW[1]) : une similarité pour les séries temporelles

Les séries temporelles peuvent être considérées comme une trajectoire unidimensionnelle, ce qui permet de modéliser par exemple les données d'un électrocardiogramme ou les cours des actions au fil du temps. La déformation temporelle dynamique permet de détecter si une série temporelle a accéléré par rapport à une autre.

**Définition 10.** Soient $u$ et $v$ deux séries temporelles définies par leurs sommets $u_1 \dots u_n$ et $v_1 \dots v_m$ respectivement, où chaque sommet appartient à $\mathbb{R}$. Étant donné la distance

[1]DTW : Dynamic Time Warping

usuelle entre un couple de sommets $\mathcal{D}$, la déformation temporelle dynamique est définie par :

$$\mathcal{D}_{\mathsf{DTW}}(u,v) = \mathcal{D}_{\mathsf{E}}(u_1,v_1) + \min \begin{cases} \mathcal{D}_{\mathsf{DTW}}(u_2 \dots u_n, v), & \|u\| > 1 \\ \mathcal{D}_{\mathsf{DTW}}(u, v_2 \dots v_m), & \|v\| > 1 \\ \mathcal{D}_{\mathsf{DTW}}(u_2 \dots u_n, v_2 \dots v_m), & \|u\| > 1 \wedge \|v\| > 1 \\ 0 & \|u\| = 1 \wedge \|v\| = 1 \end{cases}$$

La Déformation temporelle dynamique ne satisfait pas l'inégalité triangulaire, de plus cette fonction est décidée en temps quadratique par rapport au nombre de sommets des trajectoires sous l'hypothèse de temps exponentiel fort (SETH) [BK15].

### d) La distance de Levenshtein : une distance pour les séquences

La distance de Levenshtein permet de comparer des chaînes de caractères (ou séquences). La jointure par similarité de séquences a de nombreuses applications, telles que l'identification d'entité dans le langage naturel [Wan09], en biologie moléculaire, et plus généralement en bio-informatique. Par exemple, le débit du séquençage de nouvelle génération a considérablement progressé au cours de la dernière décennie, ce qui a permis de constituer de très grands ensembles de données de métagénomes, de gènes et de protéines qui pourraient grandement améliorer l'annotation fonctionnelle et la prédiction de la structure des protéines [SS18].

**Définition 11.** Soit $\Sigma$ un alphabet fini de taille $\|\Sigma\|$ et soit $\Sigma^+$ l'ensemble de toutes les chaînes de caractères non-vides. La distance de Levenshtein $\mathcal{D}_{\mathsf{Lev}}$ entre deux séquences est définie par le nombre minimum d'opérations (insertions, suppressions et substitutions) de caractères uniques nécessaires pour transformer une séquence en une autre.

Par exemple, en prenant deux séquences “mardi” et “mercredi”, la distance de Levenshtein est de 4 puisqu'il faut substituer le “a” par un “e” et insérer le mot “cre”.

Le calcul complet de la distance de Levenshtein ne peut être décidé en temps strictement sous-quadratique sous l'hypothèse de temps exponentiel fort (SETH) [BI18]. Toutefois, pour calculer la jointure par similarité, il n'est pas nécessaire de calculer complètement la distance, mais seulement de décider si la distance est inférieure à un seuil donné pour un couple de séquences. Nous utilisons dans la suite de cette thèse l'algorithme d'Ukkonen [Ukk85] avec les améliorations de Berghel et Roach [BR96] pour décider rapidement si un couple de séquences est similaire au regard du seuil donné.

Un exemple d'une jointure par similarité au sein d'un seul ensemble de données de séquences et reposant sur la distance Levenshtein pour déterminer les couples similaires

est présenté dans la figure Fig. 3.4.

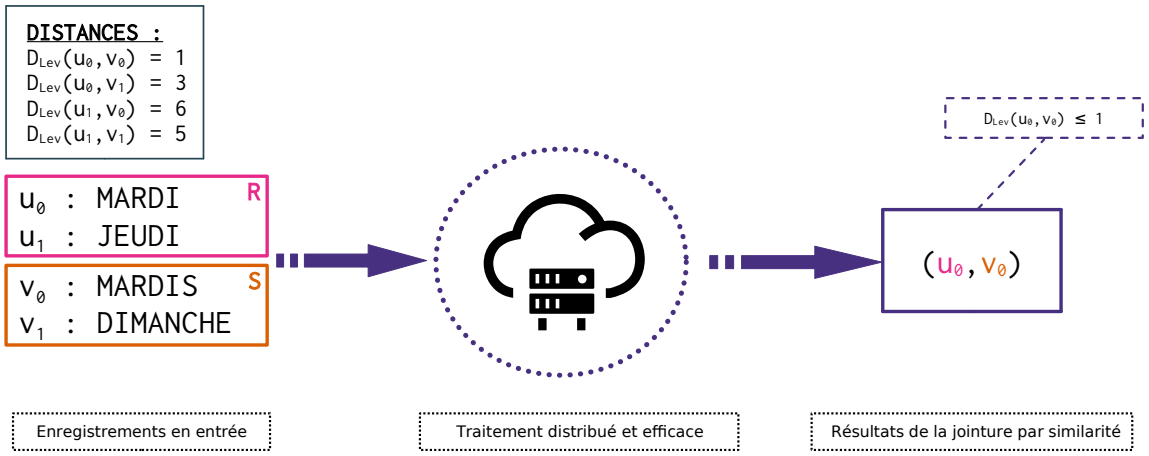



Fig 3.4 : Un exemple de jointure par similarité de séquences en utilisant la distance de Levenshtein, et en fixant le seuil à $\lambda = 1$.

Dans le cadre de la jointure par similarité, il est parfois pertinent de recourir à la distance normalisée de Levenshtein lorsque les tailles de séquences varient fortement dans les ensembles de données.

**Définition 12.** La distance de Levenshtein normalisée entre deux chaînes de caractères est définie comme suit :

$$\mathcal{D}_{\mathsf{NLev}}(u, v) = \frac{\mathcal{D}_{\mathsf{Lev}}(u, v)}{\max(\|u\|, \|v\|)}$$

## 3.2 Les algorithmes exhaustifs de jointure par similarité

À la différence de la jointure classique, il n'existe pas d'attribut de jointure prédéfini par l'utilisateur. Ainsi, il faut trouver un moyen de regrouper les enregistrements similaires pour évaluer leur distance. Une approche pour aborder ce problème consiste à définir les valeurs d'un ou plusieurs attributs de jointure de manière à rapprocher les enregistrements similaires du point de vue de la distance. Nous distinguons deux classes d'algorithmes pour la définition des attributs de jointure en fonction de l'exhaustivité des résultats. Nous appelons exhaustifs, les algorithmes qui produisent le résultat complet de la jointure par similarité, et approximatifs les autres.

Les algorithmes exhaustifs reposent principalement sur le principe des tiroirs pour construire les valeurs de l'attribut de la jointure. L'idée de ce principe est de trouver un moyen de décomposer les objets considérés en plusieurs composantes, tout en assurant que deux objets similaires aient nécessairement au moins une composante commune.

Conceptuellement, chaque composante d'un objet est utilisée en tant que valeur d'un attribut de jointure unique, assurant ainsi la complétude des résultats. Néanmoins, le nombre de composantes dépend généralement de la dimension ou de la longueur des objets, ce qui limite l'extensibilité de ces algorithmes pour de très grands ensembles de données. Nous passons maintenant en revue la littérature traitant exhaustivement l'opération de jointure par similarité.

### 3.2.1 La jointure par similarité : une approche Naïve

L'algorithme naïf [Afr12], consiste à comparer l'ensemble des enregistrements, ce qui implique le calcul d'un produit Cartésien. Pour une jointure $\mathrm{R} \bowtie_{\lambda} \mathrm{S}$ et en prenant une constante $\mathrm{J} \approx \sqrt{\mathcal{R}}$ avec $\mathcal{R} > 1$ le nombre de *Reducers* ; l'algorithme consiste à distribuer les enregistrements sur une grille de taille $\mathrm{J} \times \mathrm{J}$. Cet algorithme nécessite une seule phase de MapReduce ; l'algorithme est le suivant :

**1.a** Calculer un entier $k$ tel que $k \in [0, \mathrm{J}[$ à l'aide d'une fonction de hachage pour chaque enregistrement des deux ensembles de données,

**1.b** Émettre un couple clé-valeur $((k, i), u)$ (resp. $((i, k), v)$) pour chaque enregistrement $u \in \mathrm{R}$ (resp. $v \in \mathrm{S}$) et pour tout $i \in [0, \mathrm{J}[$. On peut remarquer que chaque couple d'enregistrements $(u, v) \in \mathrm{R} \times \mathrm{S}$ ne sera présent qu'une seule fois sur les *Reducers*,

**1.c** Calculer la jointure par similarité des enregistrements reçus, c'est-à-dire la distance de tous les couples d'enregistrements $(u, v) \in \mathrm{R} \times \mathrm{S}$ ayant la même clef,

**1.d** Émettre un couple d'enregistrements $(u, v)$ en sortie si $\mathcal{D}(u, v) \leq \lambda$.

Il faut noter qu'en fixant la constante J en fonction du nombre de *Reducers*, chaque *Reducer* a la même charge durant le traitement de la jointure par similarité. Un exemple d'exécution de l'algorithme naïf est illustré dans la figure Fig. 3.5.

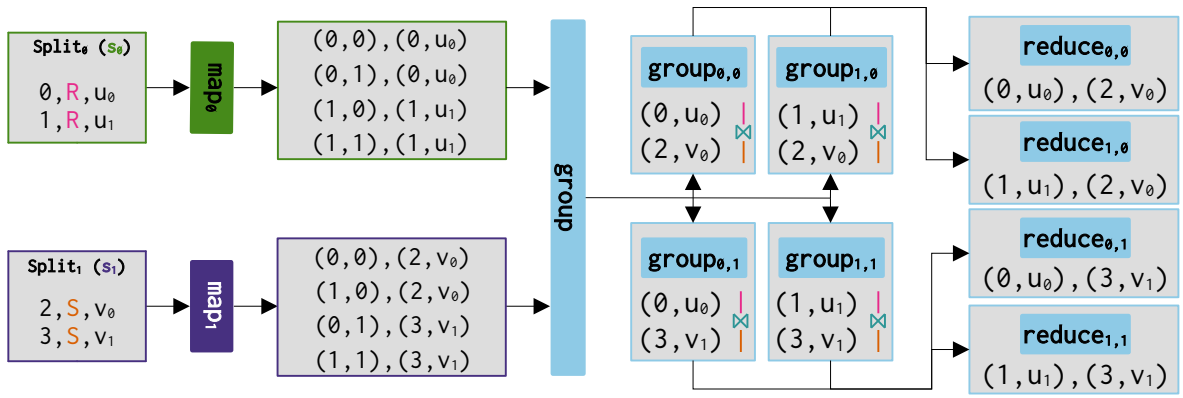



Fig 3.5 : Un exemple d'exécution de l'approche naïve pour calculer la jointure par similarité.

Les performances de cet algorithme sont désastreuses et l'extensibilité est très fortement limitée pour le traitement d'ensembles de données volumineux. Cet algorithme n'est efficace que lorsque le résultat de la jointure par similarité correspond à une grande majorité de tous les couples possibles.

### 3.2.2 La jointure floue

Dans la même étude, plusieurs autres approches ont été proposées pour traiter la distance de Hamming [Afr12] dans le cadre de la jointure par similarité. Cette distance calcule le nombre de caractères différents sur des séquences.

**Définition 13.** Soient $u$ et $v$ deux séquences appartenant à $\Sigma^n$, et de taille identique notée $n$, la distance de Hamming est la suivante :

$$\mathcal{D}_{\text{Hamming}}(u,v) = \sum_{i=0}^{n} 1\ (u_i \neq v_i)$$

#### a) L'approche Ball-Hashing

Cette approche consiste à énumérer, pour chaque séquence de l'entrée, l'ensemble des séquences voisines, c'est-à-dire l'ensemble des séquences ayant une distance de Hamming inférieure au seuil fixé par l'utilisateur.

**Définition 14.** Soit $u$ une séquence quelconque de $\Sigma^n$, le voisinage est défini par l'ensemble des séquences de taille $n$ obtenues en modifiant au plus $\lambda$ caractères.

$$\mathcal{V}(u) = \{v \mid v \in \Sigma^n \ \wedge\ \mathcal{D}_{\text{Hamming}}(u,v) \leq \lambda\}$$

Notez que la taille du voisinage dépend de trois facteurs, la longueur de la séquence considérée $n$, la taille de l'alphabet $\|\Sigma\|$ et le seuil définit par l'utilisateur $\lambda$. On peut déterminer la taille du voisinage par :

$$\|\mathcal{V}(u)\| = \sum_{k=1}^{\lambda} \mathrm{C}_n^k \ * \ (\|\Sigma\| - 1)^k$$

Cet algorithme nécessite donc un seul *job* MapReduce pour traiter la jointure par similarité $\mathrm{R} \bowtie_\lambda \mathrm{S}$ :

**1.a** Pour chaque séquence $u$ (resp. $v$) provenant de l'ensemble de données R (resp. S), un couple clé-valeur est émis pour chaque séquence de son voisinage $\mathcal{V}(u)$ (resp. $\mathcal{V}(v)$),

**1.b** Calculer la jointure par similarité des enregistrements reçus, c'est-à-dire la distance de tous les couples d'enregistrements $(u, v) \in \mathrm{R} \times \mathrm{S}$ ayant la même clef,

**1.c** Émettre un couple d'enregistrements $(u, v)$ en sortie si la distance de Hamming satisfait $\mathcal{D}_{\mathsf{Hamming}}(u, v) \leq \lambda$.

Par définition, les performances de cette approche dépendent très fortement de la taille du voisinage. En conclusion, cette approche n'est efficace que lorsque ces trois facteurs sont petits :

- La longueur des séquences,
- Le seuil fourni par l'utilisateur,
- La taille de l'alphabet.

### b) L'approche Splitting

L'idée de cette approche repose sur le principe des tiroirs, où chaque séquence est décomposée en $\lambda + 1$ sous-séquences. Cette approche garantit que deux séquences similaires aient nécessairement au moins une sous-séquence commune.

**Définition 15.** Soit $u$ une séquence de $\Sigma^+$ et supposons que la longueur de la séquence soit un multiple de $\lambda + 1$, c'est-à-dire que $\|u\| = (\lambda + 1) * p$ pour un entier $p > 0$. Pour $i \in 1, \ldots, \lambda + 1$, la sous-séquence $u_{(i-1)*p+1} \cdots u_{i*p}$ est utilisée comme une valeur de l'attribut de jointure.

Lorsque la longueur de la séquence n'est pas un multiple de $\lambda+1$, les $\|u\| \mod (\lambda+1)$ premières sous-séquences ont une longueur de $p + 1$. La distance de Hamming ne

considérant que les séquences de taille identique, la décomposition permet d'assurer que les séquences similaires aient au moins une sous-séquence en commun. En effet, deux séquences ayant au plus $\lambda$ différences partagent au moins une sous-séquence parmi leurs $(\lambda + 1)$ composantes.

Plusieurs implémentations ont été réalisées et expérimentées pour la distance de Hamming [KST15] puis pour la distance de Jaccard et de Levenshtein [KTS16]. De plus, des optimisations basées sur des filtres de Bloom [Blo70] ont été proposées pour réduire la taille du voisinage (ou le nombre de valeurs de l'attribut de jointure) à explorer en éliminant les voisins ne pouvant pas produire de résultat. Une implémentation reposant sur Spark et son modèle MapReduce a été proposé dans la littérature [TRA18 ; TRA20 ; Tra20]. Néanmoins, l'efficacité de ces approches dépend fortement du seuil et des longueurs de séquences, ce qui limite les cas d'utilisations.

### 3.2.3 La jointure par similarité exhaustive d'ensembles

La jointure par similarité exhaustive d'ensembles a reçu beaucoup d'attention [VCL10 ; MF12 ; SRT12 ; SR12 ; Ron17] et une étude expérimentale a été conduite sur les algorithmes les plus récents dans un cadre séquentiel [MAB16] et distribué [Sil16 ; Fie18]. L'étude distribuée exploite Hadoop et son modèle MapReduce, elle conclut qu'aucun des algorithmes évalués n'est extensible pour le traitement de grands ensembles de données. En effet, la plupart des algorithmes proposés ne terminent pas dans le temps imparti sur un ou plusieurs petits ensembles de données, alors qu'ils ne présentaient aucune difficulté dans le cadre séquentiel. De plus, les algorithmes distribués sont parfois beaucoup plus longs que leurs versions séquentielles. En conséquence, nous ne présentons que l'algorithme le plus performant de l'étude en termes de robustesse et de temps de traitement : *VernicaJoin* [VCL10].

#### a) L'algorithme VernicaJoin

*VernicaJoin* est un algorithme calculant la jointure par similarité d'ensembles. Plusieurs versions de l'algorithme sont présentées pour le cas de l'auto-jointure et de la jointure $R \bowtie_\lambda S$. Par souci de simplicité, nous nous concentrerons sur le cas d'une auto-jointure. Nous présentons d'abord un algorithme naïf pour expliquer le principe des tiroirs sur les ensembles, puis nous présentons l'algorithme optimisé *VernicaJoin*.

Naïvement, en utilisant le principe des tiroirs, on peut décomposer chaque ensemble par ses éléments. Un algorithme en deux phases de MapReduce serait alors le suivant :

**1.a** Émettre un couple clé-valeur $(e, u)$ pour chaque jeton $e \in u$ et pour chaque enregistrement $u \in \Gamma$,

**1.b** Calculer la jointure par similarité des enregistrements reçus, c'est-à-dire la distance de tous les couples d'enregistrements $(u, v)$ distincts ayant la même clef,

**1.c** Émettre un couple d'enregistrements $(u, v)$ en sortie si $\mathcal{D}_{\mathsf{Jaccard}}(u, v) \leq \lambda$,

**2.** Dédupliquer les résultats de la jointure par similarité.

Il convient de noter que, dans cet algorithme, le nombre de distances calculées dépend de la fréquence des jetons. De toute évidence, cet algorithme produit le résultat complet puisque tous les ensembles ayant au moins un jeton en commun sont comparés. Toutefois, puisque le seuil de la distance est connu et prédéfini par l'utilisateur, il est possible d'optimiser la charge des *Reducers* tout en conservant la complétude du résultat en sélectionnant judicieusement les couples clé-valeur émis lors de la première étape.

Soit un ensemble quelconque $u$, nous recherchons tout enregistrement $v$ tel que $\mathcal{D}_{\mathsf{Jaccard}}(u, v) \leq \lambda$. Nous pouvons donc optimiser en réduisant le nombre de jetons à sélectionner en fonction du seuil. Cette proposition se traduit par la recherche de tout enregistrement $v$ ayant au moins $(1 - \lambda) * \|u\|$ jetons en communs. Réciproquement, tout enregistrement similaire de $u$ a au plus $\lambda * \|u\|$ jetons différents, ce qui permet de réduire le nombre de jetons à sélectionner à $\lambda * \|u\| + 1$. En effet, tout couple d'enregistrements similaires aura au moins un jeton en commun. Il reste à garantir que la procédure de sélection de ces jetons soit déterministe pour tous les enregistrements afin de garantir la complétude des résultats.

**Définition 16** ([CGK06])**.** Pour un ensemble $u$ et en se donnant un ordre total, noté O, de l'univers des jetons, le $w$-préfixe correspond aux $w = \lambda * \|u\| + 1$ premiers jetons, c'est-à-dire les jetons ayant la position la plus faible dans l'ordre O.

Pour minimiser le nombre de distances calculées, il est préférable de prendre un ordre permettant de sélectionner les jetons les moins fréquents. L'algorithme *VernicaJoin* opère en quatre phases de MapReduce, s'appuyant sur le $w$-préfixe pour calculer la jointure par similarité en sélectionnant les $w$ jetons les moins fréquents.

**1.** Calculer la fréquence globale de chaque jeton de l'univers,

**2.** Trier les jetons par fréquence croissante,

**3.** Calculer la jointure par similarité en utilisant la cardinalité de chaque enregistrement et le seuil de la distance prédéfini pour déterminer le $w$-préfixe contenant les jetons les moins fréquents,

**4.** Dédupliquer les résultats de la jointure par similarité.

Le traitement de la première phase est semblable à celui expliqué précédemment pour le calcul de l'histogramme (cf. Section 2.3.4.b). Lors de la seconde phase, cet histogramme est trié par fréquence croissante sur un unique *Reducer*. Ce traitement permet de doter l'univers d'un ordre total grâce aux fréquences obtenues. Ces deux premières étapes sont illustrées dans la figure Fig. 3.6. De façon similaire à l'histogramme, on peut écarter les jetons de l'univers ayant une fréquence égale à 1, puisque ces jetons ne peuvent produire aucun résultat de la jointure par similarité. La phase suivante consiste à émettre un couple clé-valeur pour chaque jeton de son $w$-préfixe comme illustrée par la figure Fig. 3.7.

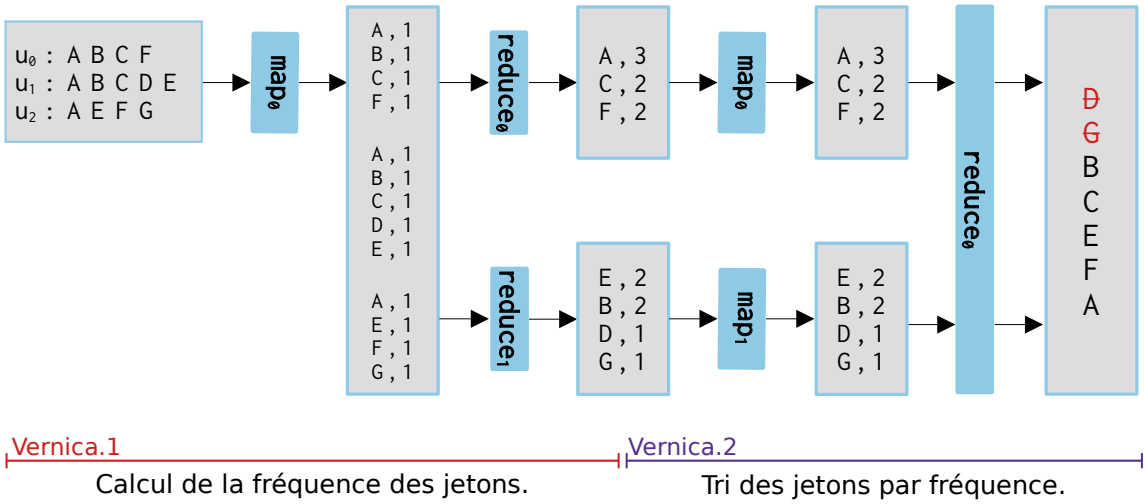



Fig 3.6 : L'algorithme *VernicaJoin* : Calcul des jetons triés par fréquence croissante. Les jetons "D" et "G" sont écartés.

Plus généralement, plus la taille des $w$-préfixes sont petits, moins il y a de communication et plus l'algorithme est efficace. Par définition, ce cas se présente lorsque les enregistrements sont de petites tailles ou lorsque le seuil de la distance est très faible. De plus, lorsque la distribution des fréquences des jetons est biaisée, c'est-à-dire qu'il y a beaucoup de jetons ayant une fréquence très faible, le $w$-préfixe est particulièrement performant pour réduire l'espace de recherche, puisque les jetons les moins fréquents sont sélectionnés en priorité.

Un deuxième filtre est appliqué par l'algorithme *VernicaJoin* sur la taille des ensembles pour écarter les ensembles ne pouvant produire un résultat de la jointure par similarité.

**Théorème 1** ([AGK06]). *Soit $u$ un enregistrement donné, on peut écarter le calcul de la distance du couple $(u, v)$ pour tout enregistrement $v$ tel que $\|u\| \leq \|v\|$ si $\|u\| < (1-\lambda) * \|v\|$.*

Dans ce but, lorsque chaque enregistrement est envoyé sur les jetons de son $w$-préfixe, l'algorithme utilise une clé composée du jeton et de la taille de l'enregistrement. Ainsi,

les enregistrements seront triés lors de leurs traitements par les *Reducers* par leurs tailles comme illustrées par la figure Fig. 3.7. Il faut noter dans cet exemple que les résultats de la jointure peuvent être dupliqués. La dernière phase permet de supprimer ces doublons.

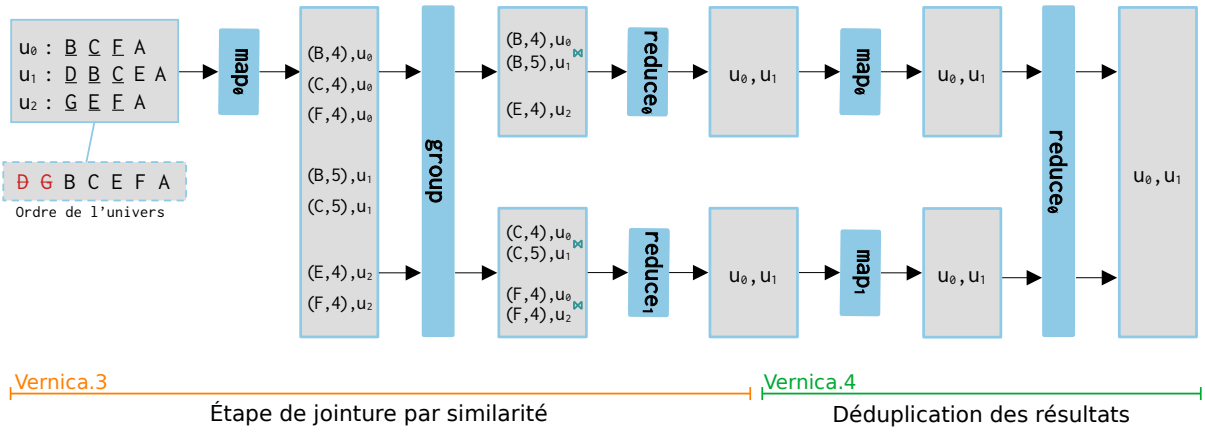



Fig 3.7 : L'algorithme *VernicaJoin* : les étapes de jointure et de déduplication. En fixant $\lambda = 0.5$, le $w$-préfixe de chaque enregistrement est souligné. Les jetons écartés sont représentés barrés en rouge. La distance de Jaccard du couple $(u_0, u_2)$ est calculée, mais le seuil n'étant pas atteint, il est filtré, contrairement au couple $(u_0, u_1)$.

Plusieurs limites au traitement de très grands ensembles de données peuvent être posées. D'abord, cet algorithme est efficace lorsque la taille du préfixe est petite, ce qui ne permet pas de calculer efficacement la jointure par similarité pour de grands ensembles ou de grands seuils de distance. De plus, si la distribution des fréquences n'est pas biaisée, l'algorithme *VernicaJoin* ne réduit pas assez le nombre de comparaisons. Enfin, lors du tri des jetons par leur fréquence sur un seul *Reducer* et lors de leurs lectures par les *Mappers* de l'étape suivante, il faut que l'ensemble de l'univers des jetons puisse tenir mémoire, ce qui n'est pas garanti dans le cas de très grands ensembles de données comprenant un très large univers. En conclusion, les performances de cet algorithme sont très dépendantes de trois facteurs :

- La taille des ensembles,
- Le seuil fourni par l'utilisateur,
- La distribution des fréquences des jetons.

La dépendance à la taille des ensembles est introduite dès la version naïve de l'algorithme (cf. VernicaBasic.1.a) dû au principe des tiroirs. Pour que les performances de cet algorithme soient indépendantes de la taille des ensembles, il faudrait une optimisation qui soit en l'inverse de la taille des ensembles, tout en conservant la complétude du résultat, ce qui semble difficile.

### 3.2.4 La similarité sur des trajectoires : État de l'art

Dans la littérature, et à notre connaissance, il n'existe pas d'algorithme de jointure par similarité distribué adapté au traitement de trajectoires et supportant la distance de Fréchet. Toutefois, il existe des approches exhaustives qui permettent de traiter la recherche de similarités et de prendre en charge les distances d'Hausdorff et de Fréchet en exploitant Spark pour distribuer le calcul sur la mémoire des nœuds de la grappe de calcul [XLP17]. D'autres fonctions de similarités ont néanmoins été exploitées dans un contexte distribué ; par exemple dans le cas de l'étude du réseau routier avec l'utilisation du plus long sous-segment de route commun [YL19].

### 3.2.5 Les techniques de filtrage pour les séquences

Nous passons en revue deux méthodes ayant fait l'objet d'une implémentation distribuée. Ces approches reposent sur le principe des tiroirs, mais les techniques pour construire les valeurs de l'attribut de jointure sont très différentes.

Plus généralement, nous renvoyons le lecteur aux trois études expérimentales [Wan14a ; Jia14 ; Yu16] pour un état de l'art complet des techniques séquentielles développées dans la littérature. La conclusion de l'étude expérimentale séquentielle [Wan14a] souligne qu'un seuil de 20% à 25% pousse les techniques actuelles à la limite. Pour de grands ensembles de données et de longues séquences ($\geq 100$), la limite est atteinte pour des seuils nettement inférieurs. Le défi réside donc dans le traitement de la jointure par similarité sur de grands ensembles de données contenant de longues séquences et pour des seuils de distance importants. Tout comme pour les algorithmes exhaustifs destinés aux ensembles, aucune des techniques présentées n'est en mesure de relever efficacement ce défi.

#### a) Le filtrage par $q$-gramme

La jointure par similarité de séquence sous la distance de Levenshtein repose généralement sur le filtrage des $q$-grammes pour réduire le nombre de distances calculées [Sha48 ; Ukk92 ; Gra01 ; AGK06 ; AB13 ; Den14 ; Wan14a ; Yu16 ; SS18].

Considérons une séquence comme un ensemble de mots de longueur $q > 0$. Par exemple, étant donné le mot "protéine" et $q = 4$, le multi-ensemble de $q$-grammes correspondant est {prot, rote, otei, tein, eine}.

**Définition 17.** Soit $u = u_1 \cdots u_n$ une séquence de $\Sigma^+$, le multi-ensemble de $q$-grammes correspondant est défini par $\mathrm{G}^q(u) = \{u_i \cdots u_{i+q-1} \mid 1 \leq i \leq n - q + 1\}$. Notez que $\mathrm{G}^q(u)$ contient $\|u\| - q + 1$ $q$-grammes.

La plupart des approches reposant sur les $q$-grammes utilise le fait qu'une seule opération (insertion, délétion, substitution) de la distance de Levenshtein ne peut détruire qu'au plus $q$ $q$-grammes conformément à la définition en fenêtre coulissante du multi-ensemble de $q$-grammes. De ce fait, la cardinalité de l'intersection des multi-ensembles de $q$-grammes de deux séquences peut être exprimée en fonction du seuil de leur distance de Levenshtein :

$$\|\mathrm{G}^q(u) \cap \mathrm{G}^q(v)\| \geq (\max(\|\mathrm{G}^q(u)\|, \|\mathrm{G}^q(v)\|)) - q * \lambda$$

Pour la distance de Levenshtein normalisée, la cardinalité de l'intersection des multi-ensembles de $q$-grammes de deux séquences est déterminée de la façon suivante lorsque $q * \Lambda < 1$ :

$$\|\mathrm{G}^q(u) \cap \mathrm{G}^q(v)\| \geq (\max(\|\mathrm{G}^q(u)\|, \|\mathrm{G}^q(v)\|)) * (1 - q * \Lambda)$$

Par ailleurs, il convient de noter que le nombre de $q$-grammes détruits dépend de la position des opérations de Levenshtein. Par exemple, lorsque deux opérations sont effectuées consécutivement sur la séquence, c'est-à-dire aux positions $i$ et $i+1$ pour tout $i$ tel que $1 \leq i < n$, il n'y a que $q+1$ $q$-grammes détruits.

De plus, les *indels* (insertions, suppressions) sont déjà inclus dans la distance de Levenshtein, nous pouvons donc utiliser deux séquences de même longueur pour déterminer la cardinalité minimale de l'intersection sans perte de généralité. Il est donc possible d'approximer la distance de Levenshtein par la distance de Jaccard puisque la cardinalité de l'union de deux multi-ensembles de $q$-grammes peut être naturellement déterminée par la cardinalité de son intersection.

**Théorème 2.** *Pour tout couple de séquences, la relation entre la distance de Levenshtein et la distance de Jaccard satisfait l'inégalité suivante :*

$$\mathcal{D}_{\mathsf{Jaccard}}(\mathrm{G}^q(u), \mathrm{G}^q(v)) \leq 1 - \frac{\|\mathrm{G}^q(u)\| - q * \lambda}{2 * \|\mathrm{G}^q(u)\| - \|\mathrm{G}^q(u)\| - q * \lambda}$$

**Corollaire 2.1.** *En utilisant la distance de Levenshtein normalisée, l'inégalité suivante est satisfaite :*

$$\mathcal{D}_{\mathsf{Jaccard}}(\mathrm{G}^q(u), \mathrm{G}^q(v)) \leq 1 - \frac{\|\mathrm{G}^q(u)\| * (1 - q * \Lambda)}{2 * \|\mathrm{G}^q(u)\| - \|\mathrm{G}^q(u)\| * (1 - q * \Lambda)}$$

Un aspect important du problème réside dans le choix de la longueur des mots $q > 0$ de manière à réduire l'espace de recherche tout en garantissant l'exhaustivité des résultats. En effet, les valeurs des attributs de jointure sont généralement établies à partir d'un seul mot de taille $q$. Par conséquent, plus les mots sont longs, plus le filtrage

est efficace puisque chaque valeur de l'attribut de jointure sera tirée d'un domaine de taille $\|\Sigma\|^q$. Toutefois, plus les mots sont longs, plus la taille de l'intersection est petite, et il peut donc être plus coûteux de retrouver les multi-ensembles similaires.

Tout algorithme calculant la jointure par similarité d'ensembles au moyen de la distance de Jaccard peut être appliqué aux séquences pour réduire l'espace de recherche et générer des couples de séquences candidates, ce qui évite le calcul du produit Cartésien. Pour minimiser davantage le nombre de distances calculées, plusieurs filtres spécifiques à la distance de Levenshtein ont été étudiés en utilisant par exemple la position des $q$-grammes [Gra01 ; Den14 ; Jia14].

Malgré tout, l'étude expérimentale sur la jointure par similarité d'ensembles [Fie18] et l'évaluation précédente (cf. section 3.2.3) ont démontré qu'aucun algorithme exhaustif et distribué de la littérature n'était extensible pour traiter de grands ensembles de données. De ce fait, ils ne conviennent pas non plus pour traiter la distance de Levenshtein efficacement.

### b) L'algorithme Pass-Join

Cette approche utilise le principe des tiroirs pour diviser les séquences en plusieurs segments de différentes tailles, c'est-à-dire l'union des $q$-grammes pour $1 < q \leq \lambda + 1$. L'idée est de garantir qu'il existe au moins un segment en commun pour tout couple de séquences similaires, tout en sélectionnant le nombre minimum de segments. Le nombre de valeurs de l'attribut de jointure correspondant au nombre de segments sélectionnés.

**Définition 18.** Soient $u$ et $v$ deux séquences de $\Sigma^+$ de taille respectives $n > 0$ et $m > 0$, et soit $\Delta$ la différence de taille de ces deux séquences, c'est-à-dire $\Delta = |n - m|$. Le nombre de segments sélectionnés est défini par :

$$\frac{\lambda^2 - \Delta^2}{2} + \lambda + 1$$

Par définition, plus le seuil est grand, plus le nombre de valeurs de l'attribut de jointure est grand. En conséquence, cette méthode peut être écartée pour de très grands seuils sur de longues séquences ($\geq 100$). Néanmoins, pour de petits seuils ($\leq 10$), cette méthode est la plus efficace [Wan14a ; Jia14].

## 3.3 La jointure par similarité : Approches approximatives

Les algorithmes approximatifs de jointure par similarité sont principalement basés sur une méthode probabiliste appelée Locality Sensitive Hashing. Cette méthode permet de construire des attributs de jointure pour différents espaces métrique, c'est-à-dire que le cadre théorique peut être appliqué à différentes distances sur différents types d'objets. L'idée est d'utiliser des fonctions de hachage dont les probabilités de collision sont sensibles à la similarité des objets. Cette méthode ne garantit plus la complétude des résultats, cependant il est possible d'assurer une certaine proportion des résultats en itérant plusieurs fois. Le nombre d'itérations requis pour générer un résultat complet dépend des probabilités de collision et non plus de la dimension ou taille des objets, ce qui la rend généralement plus efficace en termes de temps et de données transmises entre les nœuds de la grappe. Néanmoins, il n'y a pas dans la littérature d'études nous démontrant cette efficacité. Nous définirons plus tard les outils nécessaires à la comparaison de la complétude des résultats produits par cette méthode.

### 3.3.1 Locality Sensitive Hashing (LSH)

Indyk et Motwani ont introduit une méthode de hachage randomisé [IM98 ; GIM99] qui résout efficacement le problème du $(\lambda, c)$-proche voisin, y compris dans les espaces de haute dimension. Cette méthode est basée sur une famille de fonctions de hachage garantissant que les objets proches soient plus susceptibles d'entrer en collision que les objets éloignés. Plus formellement, elle est caractérisée par la définition suivante.

**Définition 19** ([IM98 ; GIM99])**.** Soit $\mathcal{D}$ une distance donnée et $\lambda$ un seuil. Étant donné un facteur d'approximation $c > 1$ et deux probabilités $p_1$ et $p_2$ telles que $0 \leq p_2 < p_1 \leq 1$, $\mathcal{H}$ est une famille de fonctions LSH pour la distance $\mathcal{D}$, si elle satisfait les conditions suivantes pour deux objets quelconques $u, v \in \mathrm{R} \times \mathrm{S}$ et pour toute fonction de hachage $h$ choisie aléatoirement et indépendamment de $\mathcal{H}$ :

- Si $\mathcal{D}(u, v) \leq \lambda$ alors $\mathsf{P}[h(u) = h(v)] \geq p_1$,
- Si $\mathcal{D}(u, v) \geq c * \lambda$ alors $\mathsf{P}[h(u) = h(v)] \leq p_2$.

En résumé, il y a de plus forte probabilité que les objets similaires aient la même valeur sous une fonction de hachage LSH que des objets éloignés. La probabilité de collision des objets ayant une distance telle que $\lambda < \mathcal{D}(u, v) < c * \lambda$ n'est pas nécessairement définie pour une famille de fonction LSH. Dans la littérature, le terme $\rho = \frac{\log(p_1)}{\log(p_2)} < 1$ est utilisé pour mesurer l'efficacité d'une famille LSH. En théorie, dans le pire des cas, le nombre

de points parcourus pour trouver un $c$-proche voisin est borné par $\mathcal{O}(\|\mathrm{IN}\|^{\rho})$ où $\|\mathrm{IN}\|$ est le nombre d'enregistrements, c'est-à-dire $\|\mathrm{R}\| + \|\mathrm{S}\|$ pour une jointure $\mathrm{R} \bowtie_{\lambda} \mathrm{S}$ et $\|\Gamma\|$ dans le cas d'une auto-jointure [IM98]. Nous renvoyons le lecteur à l'état de l'art pour plus de détail concernant LSH et ses applications [Wan14b ; Jaf21].

Pour réduire l'espace de recherche de l'algorithme, il est courant de concaténer plusieurs fonctions indépendantes LSH. Formellement, supposons que $\mathcal{H}^{\mathcal{K}}$ soit la famille LSH correspondant à une fonction de hachage obtenue en concaténant $\mathcal{K} \geq 1$ fonctions de hachage choisies uniformément et indépendamment dans $\mathcal{H}$. Par conséquent, il tient que :

$$\mathsf{P}_{g^{\mathcal{K}} \in \mathcal{H}^{\mathcal{K}}}[g^{\mathcal{K}}(u) = g^{\mathcal{K}}(v)] = \mathsf{P}_{h \in \mathcal{H}}[h(u) = h(v)]^{\mathcal{K}} = p_1^{\mathcal{K}}$$

Pour garantir l'exhaustivité du résultat, la méthode est itérée plusieurs fois, c'est-à-dire que plusieurs fonctions de hachage indépendantes, concaténées ou non, sont appliquées à un seul objet. Plus généralement, chaque itération correspond à un attribut de jointure, et le résultat d'une fonction de hachage correspond à la valeur de l'attribut de jointure. Lorsque la probabilité de collision, $p_1$, est connue pour une famille de fonctions, il est possible de garantir que les couples d'objets proches apparaissent avec une probabilité constante dans les attributs de jointure en fixant le nombre d'itérations par :

$$\mathcal{Q} = \frac{1}{p_1^{\mathcal{K}}}$$

Nous présenterons plus tard d'autres façons de déterminer le nombre d'itérations et le nombre de concaténations pour optimiser la charge des nœuds de la grappe dans un contexte de BigData. Lorsque les probabilités de collision sont connues et suffisamment élevées, la concaténation de plusieurs fonctions LSH réduit considérablement le nombre de distances calculées par rapport aux méthodes exhaustives. Il convient de noter que les méthodes exhaustives ne peuvent généralement pas utiliser une approche similaire. En effet, le nombre de valeurs de l'attribut de jointure est généralement important et dépend de la dimension ou de la longueur des objets, ce qui signifie qu'ils ne peuvent être élevés à la puissance $\mathcal{K}$ qu'au prix de coûts de communication désastreux.

Nous passons en revue maintenant plusieurs familles de fonctions LSH pour traiter les distances étudiées dans cette thèse. Toutefois, il n'existe pas toujours de méthode efficace pour certaines distances. Plus particulièrement, les distances se calculant en temps quadratique présentent souvent des difficultés. Intuitivement, ces distances requièrent un alignement propre pour chaque couple d'objets qu'il est difficile de préserver lors du hachage. Les familles de fonctions LSH destinées à s'appliquer à ces distances ont généralement des imperfections, c'est-à-dire que les probabilités de collisions ne sont pas

constantes où sont trop faibles pour être utilisées dans un contexte de BigData.

### 3.3.2 MinHash : une famille LSH pour la distance de Jaccard

MinHash [Bro97 ; Bro98] est une famille de fonctions LSH qui estime la distance de Jaccard. Elle est définie en utilisant une permutation aléatoire $\pi$ de l'univers $\mathcal{U}$. Pour tout jeton $e$ de $\mathcal{U}$, notons $\pi(e)$ la position de $e$ dans la permutation de $\mathcal{U}$. La fonction de hachage est alors définie par $h(u) = \min_{e \in u} \pi(e)$. Il est facile de prouver que $\mathsf{P}[h(u) = h(v)] = \mathsf{Jaccard}(u, v) = 1 - \mathcal{D}_{\mathsf{Jaccard}}(u, v)$. La figure Fig. 3.8 illustre une itération de MinHash pour une permutation donnée.

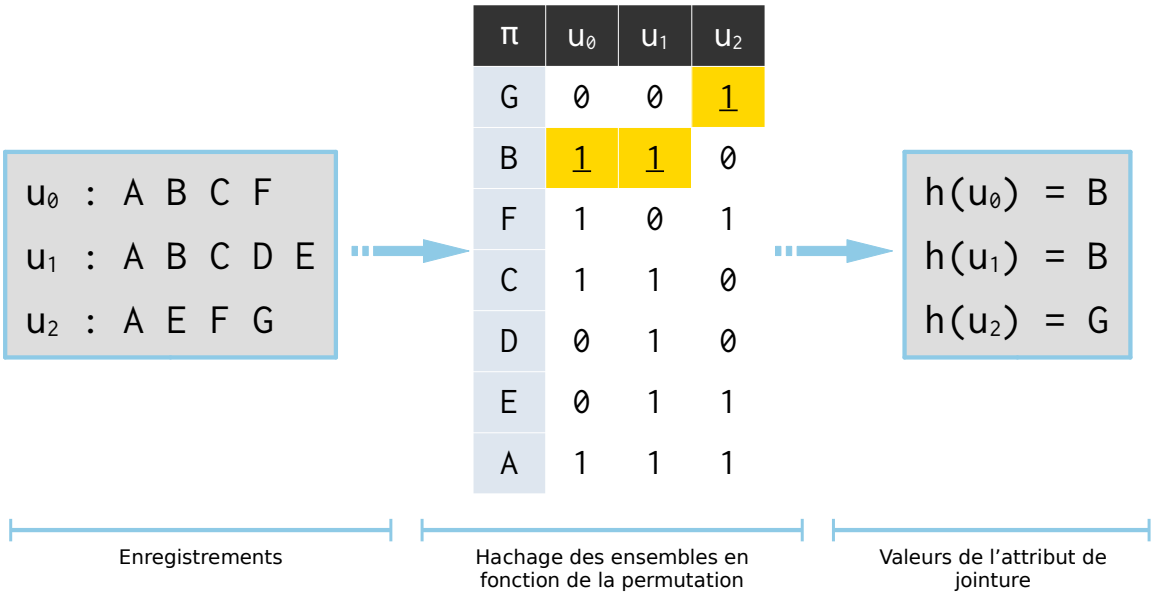



Fig 3.8 : Une itération de MinHash.

En pratique, on implémente la permutation de MinHash à l'aide d'une fonction de hachage. Nous utilisons le hachage de Zobrist [Zob90] dans cette thèse, car il a été démontré que ce hachage possède de bonnes propriétés pour l'implémentation de MinHash tout en offrant de bonnes performances en pratique [Tho17 ; DKT17].

One Permutation Hashing [LOZ12 ; SL14a ; SL14b ; Shr17] est une variante de MinHash qui calcule efficacement plusieurs fonctions de hachage indépendantes en utilisant une seule permutation, un exemple est présenté dans la figure Fig. 3.9. L'idée est de partitionner la permutation en $m$ bandes : $\mathrm{B}_1, \ldots, \mathrm{B}_m$ et d'utiliser $h_i(u) = \min_{e \in u \cap \mathrm{B}_i} \pi(e)$. Si $u \cap \mathrm{B}_i$ est vide, $h_i(u)$ est fixé à la première partie à droite (et de façon circulaire) $h_j(u)$ tel que $u \cap \mathrm{B}_j$ n'est pas vide. Le nombre de partitions est généralement défini par $m = \mathcal{Q} * \mathcal{K}$ pour calculer l'ensemble des valeurs des attributs de jointure en

une seule lecture de l'ensemble. Dans le reste de cette thèse, MinHash désignera One Permutation Hashing.

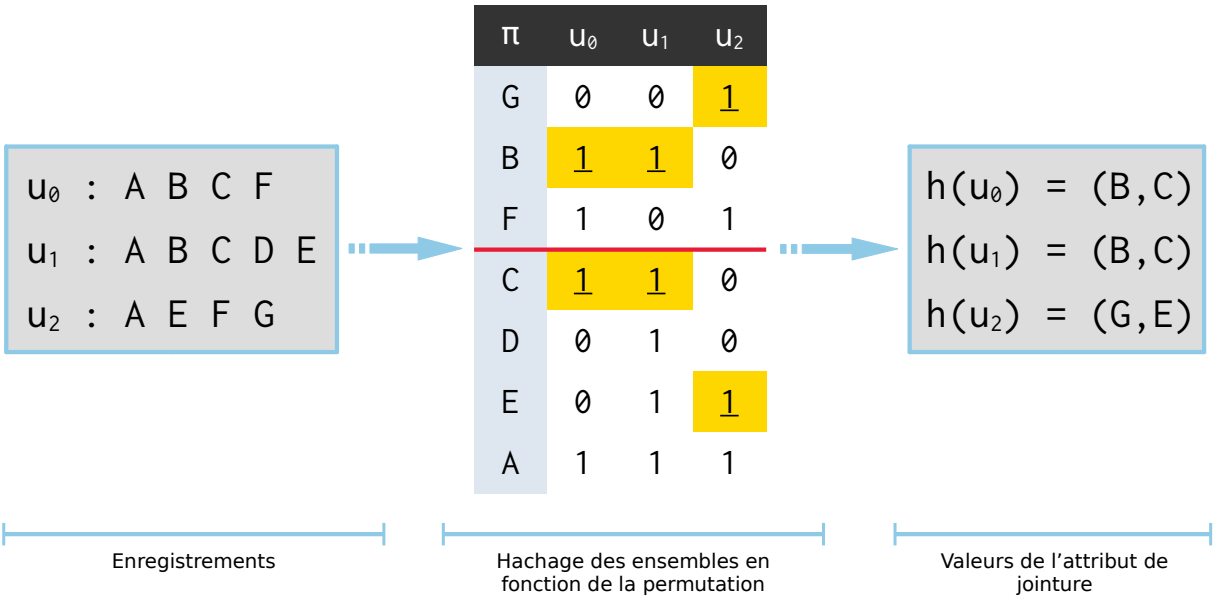



Fig 3.9 : Le fonctionnement de One Permutation Hashing en utilisant une permutation partitionnée en deux bandes.

### 3.3.3 Une famille de fonctions LSH pour la distance de Fréchet et la Déformation temporelle dynamique (DTW)

Plusieurs familles LSH ont été introduites dans la littérature pour la distance de Fréchet. La plus ancienne a été présentée par Indyk [Ind02]. Toutefois, cette famille LSH a une complexité en espace qui dépend exponentiellement de la longueur des trajectoires, ce qui signifie que le nombre d'attributs de jointure nécessaires est trop important. Cette famille de fonctions LSH est donc impraticable dans un contexte de BigData.

Une famille de fonctions LSH a récemment été introduite pour les trajectoires sous la distance Fréchet et la Déformation temporelle dynamique (DTW) [DS17 ; CDS19]. Cette famille utilise des grilles aléatoires pour transformer chaque trajectoire en une séquence de nœuds de grille. La figure Fig. 3.10 illustre le calcul des valeurs d'un attribut de jointure en se donnant une grille aléatoire.

Soit $d$ la dimension de l'espace considéré et soit $\sigma$ un réel désignant la résolution telle que $\sigma > 0$. Une grille est définie dans l'espace considéré grâce à son origine et à sa résolution.

**Définition 20.** Soit $t = (t_1 \dots t_d)$, tel que pour tout $1 \leq i \leq d$, $t_i$ désigne l'origine de la grille dans la dimension $d$, et satisfait $0 < t_i < \sigma$. Une grille est alors définie par

l'ensemble de ses nœuds :

$$\mathrm{G}_\sigma^t = \left\{ (g_1 \dots g_d) \in \mathbb{R}^d \;\mid\; \forall\, 1 \leq i \leq d,\; j \in \mathbb{N} : g_i = j * \sigma + t_i \right\}$$

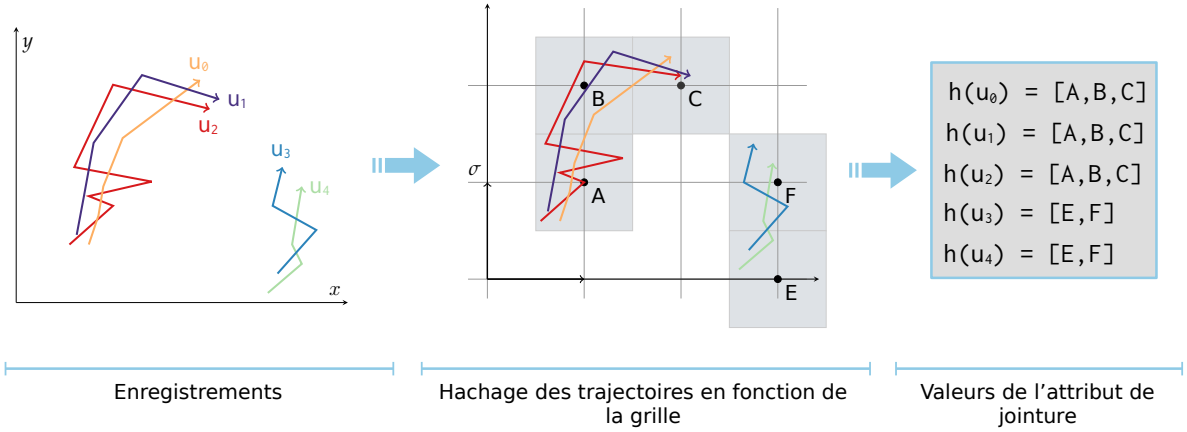



Fig 3.10 : Une famille LSH pour les trajectoires.

L'origine de la grille est choisie aléatoirement pour chaque itération, et la résolution de la grille est déterminée par $\sigma = 4 * d * \lambda$, conformément aux expériences réalisées [CDS19].

**Définition 21.** Soit $\mathcal{H}$ la famille de fonctions LSH telle que pour toute fonction $h \in \mathcal{H}$, une grille $\mathrm{G}_\sigma^t$ est sélectionnée aléatoirement et indépendamment dans l'ensemble des grilles noté $\mathcal{G}$, c'est-à-dire par construction :

$$\mathcal{G} = \{ \mathrm{G}_\sigma^t \mid t \in [0, \sigma[^d \}$$

Soit $u = u_1 \dots u_n$ une trajectoire appartenant à l'ensemble des trajectoires $\mathcal{T}$ et définie par ses sommets où chaque sommet appartient à $\mathbb{R}^d$, et soit une fonction de hachage quelconque $h \in \mathcal{H}$. La fonction transforme une trajectoire en une séquence de nœuds de grille de la façon suivante. Pour tout sommet $u_i$ avec $1 \leq i \leq n$ :

1. Calculer le nœud de grille le plus proche du sommet $u_i$,
2. Ajouter ce nœud à la séquence résultat s'il est différent du dernier nœud ajouté.

Plus précisément, le nœud de grille le plus proche d'un sommet peut être calculé simplement en connaissant la résolution de la grille et son origine.

**Définition 22.** Soit $u_i = (u_i^1 \dots u_i^d)$ un sommet d'une trajectoire appartenant à $\mathbb{R}^d$, et soit une grille $\mathrm{G}_\sigma^t$ quelconque de $\mathcal{G}$, le nœud le plus proche correspond à l'application de la fonction $f(u_i^j) = \lfloor (u_i^j - t_i)/\sigma \rceil$ sur toutes les dimensions du sommet.

Cette famille de fonctions LSH préserve l'alignement des points de trajectoire, cependant la résolution de la grille est définie globalement, contrairement à la définition originale où la résolution est définie pour chaque couple de trajectoires [DS17]. En conséquence, les probabilités de collision ne sont pas constantes et dépendent, pour chaque couple de trajectoires, de leurs nombres de sommets. En d'autres termes, plus les trajectoires sont longues, plus la probabilité de collision entre les trajectoires similaires est faible.

### 3.3.4 Les techniques d'approximation pour la distance Levenshtein

Le traitement efficace de la jointure par similarité approximative de séquences reste un problème ouvert pour la distance de Levenshtein lorsque les séquences sont longues ou que le seuil est important. Plusieurs approches ont été proposées dans la littérature, néanmoins ces techniques nécessitent un espace exponentiel [Ind04] ou possèdent de très larges facteurs d'approximations [Bar04 ; OR07], les rendant impraticables pour une application sur de très grands ensembles de données.

Nous passons en revue deux algorithmes séquentiels récents traitant la jointure par similarité approximative de séquences et présentant des résultats théoriques.

#### a) Le plongement de CGK [CGK16]

. La première méthode plonge la distance Levenshtein dans un espace de Hamming. L'idée est d'appliquer une transformation à toutes les séquences afin de les traiter dans un espace où il y a une famille de fonctions LSH efficace. Théoriquement, un plongement d'un espace métrique vers un autre est défini de la manière suivante.

**Définition 23.** Soient $\mathbf{M} = (\mathbf{E}, \mathcal{D})$ et $\mathbf{M}' = (\mathbf{E}', \mathcal{D}')$ deux espaces métriques. Soit $\alpha$ un facteur tel que $\alpha \geq 0$. Un plongement est défini par une fonction $\phi : \mathbf{M} \to \mathbf{M}'$, si la relation suivante est satisfaite pour tout $u, v \in \mathbf{E}$.

$$\mathcal{D}'(\phi(u), \phi(v)) \leq \alpha * \mathcal{D}(u, v)$$

Fondamentalement, plus le facteur $\alpha$ est petit, plus il est facile de faire la distinction entre les éléments similaires et éloignés dans le nouvel espace métrique.

**Théorème 3** ([CGK16 ; ZZ17])**.** *Pour tout entier $n > 0$, il existe un algorithme calculant l'application $\phi : \Sigma^n \to \Sigma^{3n}$ telle que pour deux séquences quelconques, $u, v \in \Sigma^n$, il tient*

*avec de fortes probabilités que :*

$$\mathcal{D}_{\mathsf{Hamming}}(u, v) \leq \mathcal{O}(\mathcal{D}_{\mathsf{Lev}}(u, v)^2)$$

Cette analyse a été améliorée et une implémentation séquentielle a été proposée [ZZ17], démontrant que les résultats pratiques sur plusieurs ensembles de données permettent de traiter la jointure par similarité pour de longues séquences. Toutefois, il demeure difficile de paramétrer convenablement l'algorithme en raison de la non-connaissance des probabilités de collision. Pour combler ce fossé, il a été proposé dans la littérature de combiner le plongement et le traitement dans l'espace de Hamming en une famille de fonctions LSH [McC21], cependant les probabilités sont trop faibles pour pouvoir être utilisées pour de très larges seuils. En effet, la probabilité de collision de deux séquences similaires dépend exponentiellement du seuil.

### b) L'approche MinHash Ordonné (OMH) [Mar19]

La seconde méthode utilise MinHash sur l'ensemble des $q$-grammes d'une séquence (cf. section 3.2.5.a) et définit une famille LSH préservant l'alignement des $q$-grammes dans les séquences originales. L'idée est de construire une valeur d'un attribut de jointure par la concaténation de plusieurs $q$-grammes ordonnés selon leur position dans la séquence originale. Toutefois, les probabilités de collision sont trop faibles pour être utilisées en pratique et la preuve ne prend pas en compte la multiplicité des $q$-grammes dans les séquences.

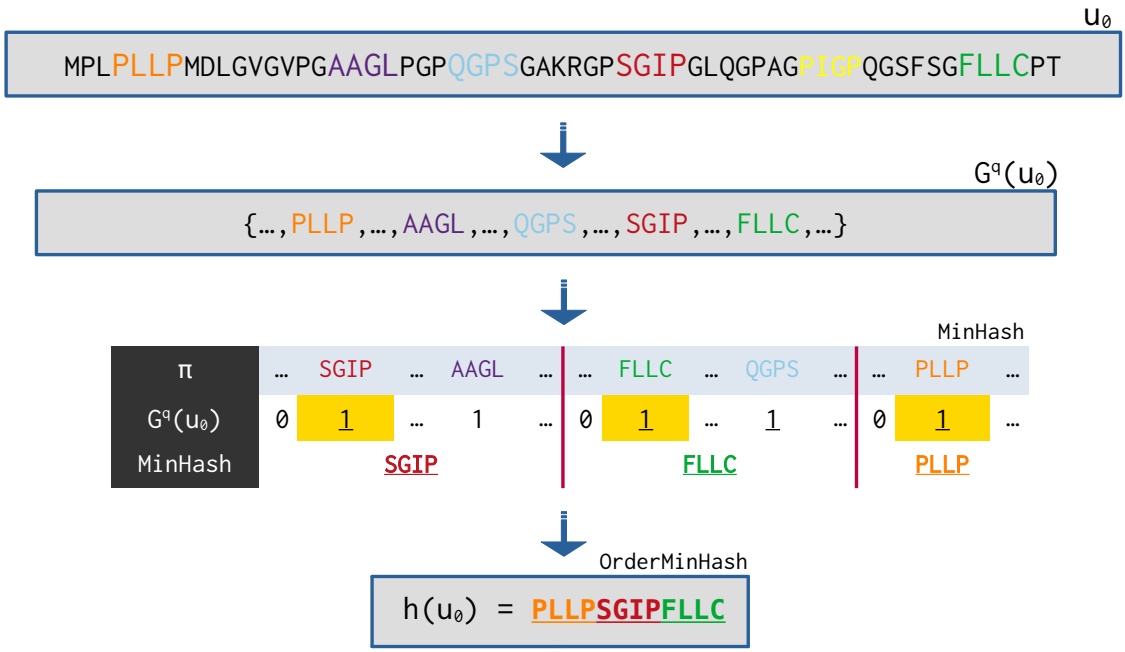



Fig 3.11 : Une famille LSH pour les séquences. Dans cet exemple, trois $q$-grammes sont sélectionnés sur la séquence originale et réordonnés en fonction de leur position dans la séquence originale.

### 3.3.5 La construction d'une esquisse d'objet

Les esquisses d'objets peuvent être considérées comme des empreintes. Lorsque les empreintes ne représentent que quelques octets et sont petites en termes d'espace comparé aux objets originaux, elles peuvent être rapidement générées et comparées. Ces empreintes sont générées à l'aide de LSH, permettant de faire une approximation de la distance considérée lors de la comparaison. L'idée est d'échanger un petit degré de complétude contre une réduction de l'espace de recherche importante.

**Définition 24** ([Chr19])**.** Soit $\mathcal{H}$ une famille de fonctions LSH et une constante $b \geq 1$. Pour tout $i \in 1, \ldots, b$ avec $h_i \in \mathcal{H}$ tel que $h_i : \mathrm{T} \to \mathrm{X}$ où T désigne un ensemble d'enregistrements et X désigne l'ensemble des valeurs des attributs de jointure construit en utilisant LSH. En prenant une fonction de hachage universelle supplémentaire $f : \mathrm{X} \to \{0, 1\}$, l'esquisse d'un objet $u \in \mathrm{T}$ est définie par $s(u) \in \mathrm{T} \to \{0, 1\}^b$ de sorte que chaque bit $i$ corresponde à $s(u)_i = f(h_i(u))$.

La probabilité qu'un bit $i$ soit identique pour un couple d'esquisses données dépend des valeurs de la fonction LSH $h_i$. Plus généralement, cette probabilité de collision est déterminée par la famille de fonctions LSH utilisée. Par exemple, pour les ensembles, la probabilité est de $(2 - \lambda)/2$ pour chaque bit [LK10]. Dès lors, pour décider si un

couple d'ensembles sont proches du point de vue des esquisses, la somme des collisions est calculée, c'est-à-dire $t = (\sum_{i=1}^{b} 1\ (s(u)_i = s(v)_i))/b$, et un couple d'ensembles est filtré si $t < (2 - c * \lambda)/2$.

Un aspect important du problème réside dans le choix de la taille des esquisses, c'est-à-dire le choix du paramètre $b$, de manière à garantir une certaine précision pour pouvoir faire la distinction entre les points proches et les points éloignés, tout en minimisant les données transmises. Il faut donc trouver un compromis entre la précision et l'efficacité lors du choix de la taille des esquisses. Dans cette thèse et dans le cas des ensembles, la construction des esquisses se fait en utilisant One Permutation Hashing afin de calculer les valeurs des $b$ fonctions de hachage LSH en une seule permutation, accélérant ainsi leur construction.

### 3.3.6 Le paramétrage de LSH dans un contexte de BigData

Pour l'instant, nous n'avons pas spécifié le nombre de concaténations, $\mathcal{K}$, nous passons principalement en revue les travaux antérieurs fournissant des garanties théoriques. Dans ce qui suit, nous supposons qu'il existe une famille LSH, $\mathcal{H}$, avec des probabilités constantes $p_1, p_2$, et $\sigma = c * \lambda$.

Indyk et Motwani ont proposé une approche pour déterminer le nombre de concaténations en fonction du nombre d'enregistrements en entrée [IM98]. En paramétrant le nombre de concaténations par $\mathcal{K} = \log_{1/p_2}(\|\text{IN}\|)$, il tient que pour toute fonction de hachage $g^{\mathcal{K}} \in \mathcal{H}^{\mathcal{K}}$ :

- Si $\mathcal{D}(u, v) \leq \lambda$ alors $\text{P}[g^{\mathcal{K}}(u) = g^{\mathcal{K}}(v)] \geq 1/\|\text{IN}\|^{\rho}$
- Si $\mathcal{D}(u, v) \geq \sigma$ alors $\text{P}[g^{\mathcal{K}}(u) = g^{\mathcal{K}}(v)] \leq 1/\|\text{IN}\|$

Cependant, l'utilisation de cette approche dans un contexte de BigData conduit à des coûts de communication désastreux en raison du nombre excessif d'itérations nécessaires pour maintenir un bon niveau d'exhaustivité. Il y a donc un compromis à trouver entre la réduction de l'espace de recherche et les coûts de communication. Ce problème est en partie résolu par l'approche de Hu et al. [HTY17 ; HYT19] qui garantit, dans le modèle de coût massivement parallèle (MPC) [KSV10], une charge espérée, c'est-à-dire une charge moyenne, sur les nœuds de la grappe. Pour obtenir cette garantie sur la charge des nœuds, le nombre de concaténations et d'itérations de LSH sont définis en fonction du nombre total de processeurs P chargés de calculer la jointure par similarité.

**Théorème 4** ([HTY17 ; HYT19])**.** *Il existe un algorithme de jointure par similarité distribué et utilisant une famille de fonctions LSH d'efficacité* $\rho$ *s'exécutant sur* P *processeurs en* $\mathcal{O}(1)$ *tour, c'est-à-dire en un job MapReduce, générant chaque couple du*

*résultat de la jointure par similarité avec au moins une probabilité constante et ayant la charge espérée suivante*[1] *:*

$$\tilde{\mathcal{O}}\left(\sqrt{\frac{\|\mathrm{R} \bowtie_{\lambda} \mathrm{S}\|}{\mathrm{P}^{1/(1+\rho)}}} + \sqrt{\frac{\|\mathrm{R} \bowtie_{\sigma} \mathrm{S}\|}{\mathrm{P}}} + \frac{\|\mathrm{IN}\|}{\mathrm{P}^{1/(1+\rho)}}\right).$$

Soit $f : \mathrm{X} \to \{0,1\}^{32}$ une fonction de hachage universelle. En fixant le nombre de concaténations à $\mathcal{K} = \left\lceil \log(p_1, 1/\mathrm{P}^{\frac{\rho}{1+\rho}}) \right\rceil$, l'algorithme nécessite un seul *job* MapReduce :

**1.a** Sélectionner aléatoirement et indépendamment $\mathcal{Q}$ fonctions de hachage $g_1^{\mathcal{K}}, \ldots, g_{\mathcal{Q}}^{\mathcal{K}}$ de $\mathcal{H}^{\mathcal{K}}$,

**1.b** Émettre un couple pour tout $i \in 1, \ldots, \mathcal{Q}$, un couple clé-valeur $((i, f(g_i^{\mathcal{K}}(u)), \mathrm{R}), u)$ (resp. $((i, f(g_i^{\mathcal{K}}(v)), \mathrm{S}), v)$) pour tout enregistrement $u$ (resp. $v$) provenant de l'ensemble R (resp. S),

**1.c** Calculer la jointure par similarité en utilisant $(i, f(g_i^{\mathcal{K}}(u)))$ comme une valeur d'attribut de jointure, c'est-à-dire qu'un enregistrement $u \in \mathrm{R}$ se joint avec un enregistrement $v \in \mathrm{S}$ s'ils ont une valeur en commun pour un attribut de jointure,

**1.d** Émettre un couple d'enregistrements $(u, v)$ en sortie si $\mathcal{D}(u, v) \leq \lambda$.

Le nombre d'itérations est donné par $\mathcal{Q} = \lceil p_1^{-\mathcal{K}} \rceil$ dans [HTY17], garantissant que chaque couple d'enregistrements similaires ait une probabilité constante d'être généré en sortie. Cependant, les utilisateurs peuvent vouloir produire le résultat complet de la jointure par similarité. En fixant $\mathcal{Q} = \lceil 3 * p_1^{-\mathcal{K}} * \ln(\|\mathrm{IN}\|) \rceil$, la probabilité de produire la jointure par similarité complète est de $1 - 1/\|\mathrm{IN}\|$ [HYT19]. Nous préférons laisser les utilisateurs spécifier la qualité du résultat attendus en fixant $\mathcal{Q} = \lceil \mathbb{E} * p_1^{-\mathcal{K}} \rceil$ avec $1 \leq \mathbb{E} \leq 3 * \ln(\|\mathrm{IN}\|)$. Par exemple, en utilisant MinHash et en fixant le seuil à $\lambda = 0.05$, le nombre de concaténations sera de $\mathcal{K} = 36$ avec $\rho = 0.5$ pour une grappe de calcul de 256 processeurs. Pour avoir une probabilité constante, le nombre d'itérations sera de $\mathcal{Q} = 12$. Pour une grappe de 24 processeurs, on fixera $\mathcal{K} = 21$ et $\mathcal{Q} = 3$. Du point de vue de l'information contenue dans les valeurs d'un attribut de jointure, LSH permet d'augmenter la quantité d'information en concaténant plusieurs composantes en comparaison aux algorithmes exhaustifs. Il faut noter que le paramètre $\rho$ est défini par l'utilisateur dans le cas de MinHash puisque les probabilités de collision sont continues sur l'intervalle $[0, 1]$ (cf. section 3.3.2). De plus, le choix du nombre de processeurs n'est pas clairement déterminé sur l'environnement Hadoop.

Un exemple d'exécution sur le modèle MapReduce est illustré dans la figure Fig. 3.12. Dans cet exemple, il n'y a pas de doublons dans les résultats de la jointure puisqu'il

[1] La notation $\tilde{\mathcal{O}}$ ignore les facteurs polylogarithmiques.

n'y a qu'une seule valeur commune dans les attributs de jointure. Cependant, dans le cas général, le traitement pour dédupliquer les résultats n'est pas clairement indiqué. De plus, de façon similaire à l'algorithme naïf de jointure (cf. section 2.3.2), lors de très grands ensembles de données, le traitement de la jointure par similarité peut être déséquilibré sur les nœuds de la grappe de calcul, conduisant à un traitement inefficace ou une incapacité à terminer le traitement de la jointure dans le pire des cas. Cette situation peut se produire lorsque le nombre de concaténations n'est pas suffisamment élevé. De plus, l'ensemble des enregistrements est transmis $\mathcal{Q}$ fois sur les *Reducers*, ce qui n'est pas très efficace et limite en pratique le nombre de concaténations.

Enfin, le compromis entre la réduction de l'espace de recherche et les coûts de communication dépend aussi de la complexité de la distance considérée, ce qui n'est pas pris en compte par cet algorithme. Pour les distances se calculant en temps quadratique, il est généralement préférable de réduire davantage l'espace de recherche. Tous ces compromis complexifient le paramétrage dans la pratique, ce qui complique les expériences en comparaisons aux algorithmes exhaustifs.

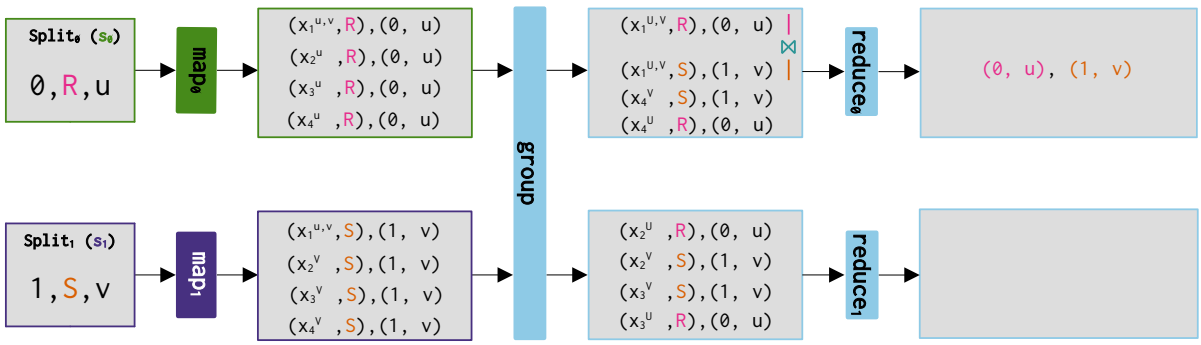



Fig 3.12 : Un exemple d'exécution de l'algorithme de Hu et al. avec le nombre d'itérations fixé à $\mathcal{Q} = 4$. La valeur $x_1^{u,v}$, issue du premier attribut de jointure, est la seule valeur commune des quatre attributs de jointure entre les deux enregistrements $u$ et $v$.

### 3.3.7 Les mesures de rappel et de précision

Pour pouvoir mesurer l'exhaustivité du résultat et la réduction de l'espace de recherche des algorithmes reposant sur LSH, nous mesurons le **rappel** et la **précision**, permettant de comparer la qualité du filtrage et des résultats produits par les algorithmes exhaustifs et approximatifs. Ces mesures de performances proviennent du domaine de l'apprentissage automatique, nous les définissons pour le cas de la jointure par similarité.

Le **rappel** permet de quantifier l'exhaustivité du résultat produit par les méthodes approximatives. Il correspond à la fraction du nombre de couples d'enregistrements similaires correctement générés par un algorithme approximatif par rapport au nombre

complet de couples d'enregistrements similaires.

**Définition 25.** Le **rappel** est défini de la manière suivante :

$$\textbf{rappel} = \frac{\text{Nombre de résultats générés}}{\text{Nombre total de résultats}}$$

Par définition, le **rappel** nécessite de connaître le résultat complet de la jointure par similarité, ce qui n'est pas toujours possible. En effet, les algorithmes exhaustifs échouent régulièrement à terminer le traitement de la jointure par similarité lorsque les ensembles de données sont volumineux.

La **précision** pour sa part permet de mesurer la qualité du filtrage. Elle correspond à la fraction du nombre de couples d'enregistrements similaires correctement générés par un algorithme approximatif par rapport au nombre de couples d'enregistrements identifiés comme similaires, ce qui correspond au nombre de distances calculées.

**Définition 26.** La **précision** est définie de la manière suivante :

$$\textbf{précision} = \frac{\text{Nombre de résultats générés}}{\text{Nombre de distances calculées}}$$

Nous évaluons ces deux mesures lors du calcul des distances, c'est-à-dire que les résultats produits en sortie ne contiennent pas de couple d'enregistrements ayant une distance supérieure au seuil donné.

## 3.4 Conclusion

Dans ce chapitre, nous avons présenté les principaux algorithmes et techniques permettant de traiter la jointure par similarité. Nous avons séparé ces algorithmes en deux catégories en fonction de la complétude des résultats. En résumé,

- Aucun algorithme exhaustif présenté dans la littérature ne permet de traiter la jointure par similarité de façon efficace dans le cas général. L'efficacité de ces approches dépend fortement de divers facteurs tels que la taille des enregistrements, leur dimension ou le seuil défini par l'utilisateur. Il est à noter que les performances des algorithmes distribués peuvent parfois être décevantes par rapport à leurs versions séquentielles, présentant une extensibilité extrêmement limitée. Ces limites ne semblent pas pouvoir être facilement surmontables, suggérant la nécessité d'adopter une approche différente pour aborder l'opération de jointure par similarité dans le contexte du BigData.

- Les performances des algorithmes approximatifs reposant sur LSH ne dépendent que des probabilités de collision, ce qui semble très efficace, cependant il n'y a pas d'études expérimentales pour le démontrer. De plus, les algorithmes présentés dans la littérature ne prennent pas en compte le déséquilibre des données, ce qui peut avoir un effet désastreux sur les performances.

- Les distances se calculant en temps quadratique nécessite un alignement propre à chaque couple d'objets, ce qui présente des difficultés pour LSH et la construction de famille de hachage praticables dans un contexte de BigData.

# Chapitre 4

# *MRS-join* : Un algorithme approximatif de jointure par similarité

Dans ce chapitre, nous introduisons un algorithme de jointure par similarité, appelé "MapReduce Similarity-Join" (*MRS-join*), reposant sur le modèle MapReduce, et sur des fonctions de hachage LSH pour regrouper les objets similaires. Nous présentons d'abord tous les prérequis à la compréhension de cet algorithme, puis nous exhibons les résultats théoriques et expérimentaux. Ce travail a conduit aux deux publications [Riv22a ; Riv22b].



## 4.1 Préliminaires

Afin d'optimiser la répartition des charges en fonction des caractéristiques des données, d'éviter les coûts de communication superflus et de prévenir les divers déséquilibres pouvant survenir lors de la jointure, nous adaptons l'algorithme *MRFA-join* [HBL14] (cf. section 2.3.4) initialement conçu pour le traitement de l'équi-jointure afin de traiter la jointure par similarité. Autrement dit, nous ajoutons une étape préalable à l'algorithme de Hu et al. [HTY17 ; HYT19] (cf. section 3.3.6) pour construire l'histogramme de la jointure par similarité.

Nous rappelons que nous utilisons une famille de fonctions LSH adaptée à la distance souhaitée pour regrouper les enregistrements similaires (cf. section 3.3.1). Une itération de LSH consiste à appliquer une fonction de hachage LSH sur les enregistrements. Une itération peut être considérée comme un partitionnement de l'union des ensembles de données, $\mathrm{R} \cup \mathrm{S}$, où chaque partie est associée à une valeur spécifique de la fonction de hachage. Nous calculons ensuite la distance au sein de chaque partie du partitionnement. Ainsi, le calcul de la distance entre deux enregistrements $u$ et $v$ appartenant à $\mathrm{R} \times \mathrm{S}$, n'est effectué que si les valeurs obtenues à partir de la fonction de hachage LSH sont les mêmes. Les résultats de l'application d'une fonction de hachage LSH lors d'une itération peuvent également être considérés comme valeurs d'un attribut de jointure et permettant de ne comparer que les enregistrements correspondant à ces mêmes valeurs.

Pour produire une large partie des résultats, il faut utiliser plusieurs itérations indépendantes. Toutefois, chaque itération augmente également le nombre de distances calculées, car des couples d'enregistrements dont la distance dépasse le seuil, $\lambda$, peuvent se retrouver dans une même partie d'un des partitionnements. Soit un seuil $\sigma$ satisfaisant $\sigma \geq \lambda$. Nous utilisons la notation $\mathrm{R} \bowtie_{\sigma} \mathrm{S}$ pour distinguer les couples candidats dont la distance devra être calculée du résultat final lui-même $\mathrm{R} \bowtie_{\lambda} \mathrm{S}$. Cela signifie que la notation $\mathrm{R} \bowtie_{\sigma} \mathrm{S}$ représente la jointure par similarité que l'algorithme devra traiter en utilisant une famille de fonction LSH quelconque.

### 4.1.1 La construction des attributs de jointure avec LSH

Nous définissons formellement les valeurs des attributs de jointure d'un ensemble de données en utilisant une famille de fonctions LSH quelconque. Soit $\mathcal{H}^{\mathcal{K}}$ une famille de fonctions LSH construite à partir de $\mathcal{K} \geq 1$ concaténations (cf. section 3.3.1). Soit $\mathcal{Q}$ le nombre d'itérations, déterminé en fonction des probabilités de la famille LSH et de la complétude du résultat désiré par l'utilisateur (cf. section 3.3.6).

**Définition 27.** Pour $i \in 1, \ldots, \mathcal{Q}$, une fonction LSH $g_i^{\mathcal{K}}$ est uniformément et indépendamment sélectionnée depuis la famille LSH $\mathcal{H}^{\mathcal{K}}$. En se donnant une fonction de hachage universelle $f$, l'ensemble des valeurs des attributs de jointure d'un ensemble de données $\mathrm{T} \in \{\mathrm{R}, \mathrm{S}\}$ est alors défini par :

$$\chi(\mathrm{T}) = \{(i, f(g_i^{\mathcal{K}}(u))) \mid \forall\ 1 \leq i \leq \mathcal{Q},\ u \in \mathrm{T}\}$$

En pratique, chaque valeur de l'attribut de jointure est représentée sur 64 bits de telle sorte que les 32 premiers bits correspondent à l'index de l'attribut et les derniers à la valeur résultant du hachage du résultat de la fonction LSH. L'idée est que nous n'avons plus besoin du résultat de la fonction LSH, mais seulement d'avoir une valeur pouvant

être utilisée rapidement pour rassembler les couples d'enregistrements probablement similaires tout en garantissant des propriétés d'équilibrage. Nous reviendrons lors de la présentation des expériences sur les paramètres et les familles LSH utilisés en fonction des objets d'études.

### 4.1.2 L'histogramme de la jointure : Adaptation à la jointure par similarité et redistribution

Il reste à étendre la définition de l'histogramme d'une équi-jointure (cf. section 2.2.1.c) pour le traitement de la jointure par similarité.

**Définition 28.** L'histogramme de la jointure, ne contenant que les données pertinentes, c'est-à-dire en supprimant les parties des partitionnements ne pouvant produire de résultats, est défini par :

$$\mathrm{Hist}(\mathrm{R} \bowtie_\sigma \mathrm{S}) = \{(x, \mathbf{f}_x^{\mathrm{R}}, \mathbf{f}_x^{\mathrm{S}}) \mid x \in \chi(\mathrm{R}) \cap \chi(\mathrm{S})\}.$$

Pour générer l'ensemble des résultats de la jointure par similarité, LSH nécessite plusieurs itérations. Cependant, la taille de l'histogramme est liée à ce nombre d'itérations, ce qui peut poser des problèmes au niveau de la mémoire puisque l'algorithme *MRFA-join* nécessite de stocker l'ensemble de l'histogramme en mémoire. Pour de très grands ensembles de données ou lorsque la complétude du résultat est requise avec de très forte probabilité, on peut s'attendre à ce que l'histogramme de la jointure ne tienne pas en mémoire.

Pour résoudre ce problème, il faut remarquer que lors de la jointure, les tâches **map** ne nécessitent qu'une partie de l'histogramme de la jointure. En effet, chaque tâche **map** n'a besoin que des entrées de l'histogramme correspondant aux valeurs de l'attribut de jointure présentes dans son *split* de données en entrée. Nous introduisons donc un schéma de redistribution basé sur l'occurrence des valeurs de l'attribut de jointure dans les différentes portions de l'entrée. Nous définissons une portion de l'entrée de deux façons par :

- un *split*, c'est-à-dire la portion de données traitée par une tâche **map** ;
- un *chunk*, c'est-à-dire un morceau d'un *split*.

#### a) La redistribution de l'histogramme d'une équi-jointure

Par souci de simplicité, nous présentons d'abord notre schéma de redistribution dans le cadre de l'équi-jointure en utilisant les *splits* comme unité de référence. Une illustration est présentée dans la figure Fig. 4.1.

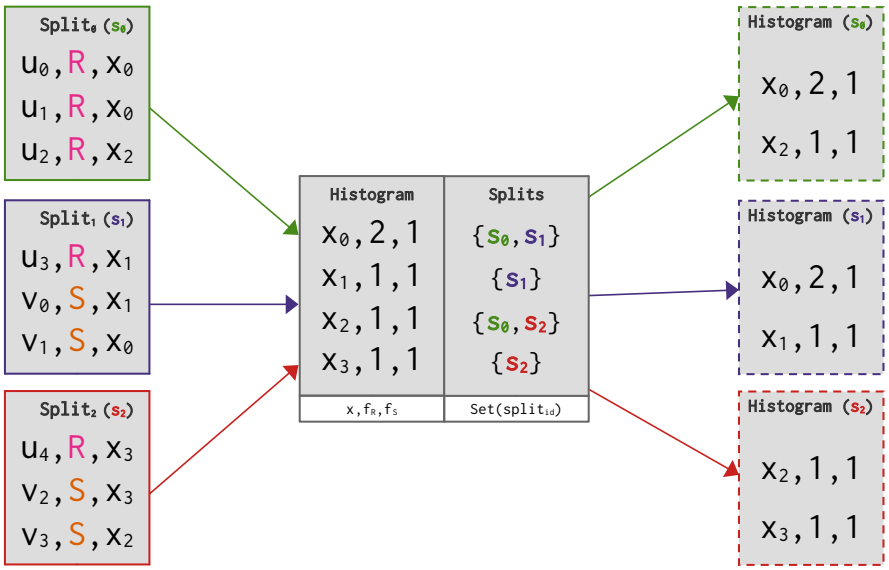



Fig 4.1 : Un exemple de redistribution de l'histogramme en fonction de l'occurrence des valeurs de l'attribut de jointure dans les *splits* de données. À la fin de la redistribution de l'histogramme, chaque *split* a l'histogramme lui correspondant stocké sur l'HDFS.

Dans le cas de la jointure, cette redistribution permet d'améliorer l'analyse de l'algorithme *MRFA-join* (cf. section 2.3.4.c). En effet, les tâches **map** de l'étape de jointure ne devront avoir en mémoire qu'un sous-ensemble de l'histogramme. Au maximum, la taille de ce sous-ensemble correspond au nombre d'enregistrements dans le *split* traité par la tâche : ceci est dû au fait qu'il n'y a qu'une valeur de l'attribut de jointure par enregistrement dans le cas de l'équi-jointure. En conséquence, en ajoutant une étape de redistribution à l'algorithme *MRFA-join*, cet algorithme devient asymptotiquement optimal. Bien entendu, il ne s'agit que d'un argument intuitif qui mériterait une analyse complète prenant en compte le coût de la redistribution de l'histogramme. Nous développerons cette analyse lors de la redistribution de l'histogramme dans le cadre de l'opération de jointure par similarité.

### b) La redistribution de l'histogramme dans les jointures par similarité

La redistribution de l'histogramme dans le cadre d'une jointure par similarité est illustrée par la figure Fig. 4.2. Fondamentalement, cette méthode permet d'assurer que la taille de l'information supplémentaire nécessaire au traitement d'une tâche **map** soit liée au nombre d'enregistrements traités par cette tâche. Cependant, contrairement à l'équi-jointure, le nombre d'attributs de jointure par enregistrement dépend de la famille de fonctions LSH et de ses probabilités.

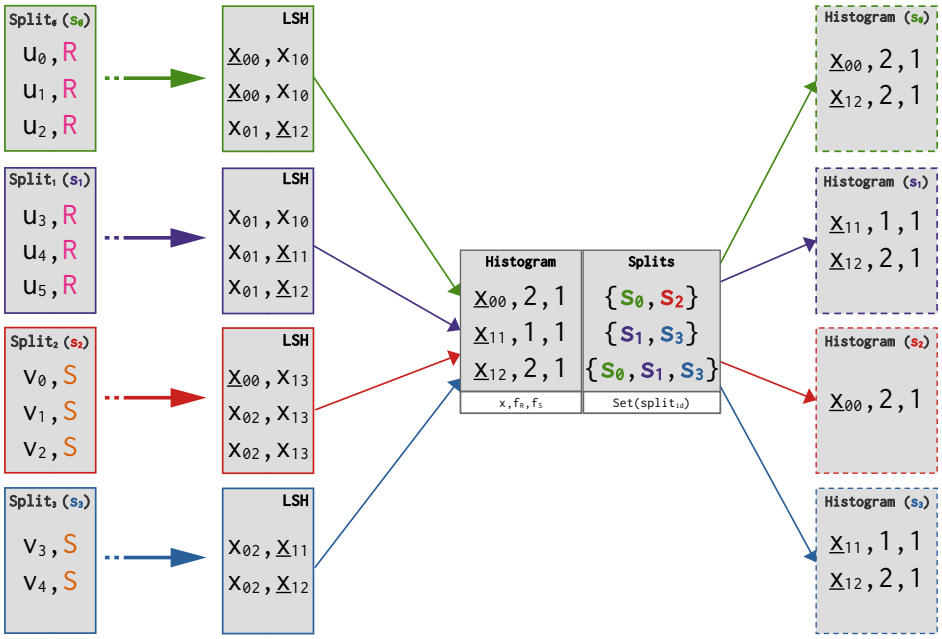



Fig 4.2 : Un exemple de redistribution de l'histogramme en fonction de l'occurrence des valeurs des deux attributs de jointure calculés dans les *splits* de données. Les valeurs soulignées font partie l'histogramme global puisqu'elles sont présentes dans les deux ensembles de données.

L'utilisation de *chunks* permet seulement d'avoir une gestion de la mémoire plus fine en garantissant qu'un nombre maximum de tuples de l'histogramme soit gardé en mémoire au même instant. Ce principe de redistribution à deux niveaux permet de garantir que les histogrammes distribués puissent tenir en mémoire, peu importe le nombre d'enregistrements par *split* et le nombre d'itérations de LSH. En effet, la taille d'un enregistrement peut grandement varier pour l'opération de jointure par similarité. En prenant l'exemple des séquences, leurs longueurs peuvent varier considérablement, s'étendant de quelques dizaines à plusieurs centaines de milliers de caractères. Cette variation dépend du type et de l'origine des séquences étudiées. Plus la taille des enregistrements est faible, plus il y aura d'enregistrements par *split*. Pour des ensembles de données possédant des enregistrements de tailles très hétérogènes, il est possible qu'un déséquilibre se produise dans la taille de l'histogramme à stocker en mémoire par *split*. Ce déséquilibre est exacerbé par l'utilisation de plusieurs attributs de jointure par enregistrement, et peut entraîner l'échec de l'algorithme en raison d'une allocation mémoire insuffisante. L'utilisation de *chunks* permet de prévenir ces cas en paramétrant la taille de l'histogramme allouée, et ce, peu importe les objets étudiés et leurs tailles.

Pour ce faire, un paramètre supplémentaire, noté $\mathbf{t}_{\max}$, est nécessaire. Ce paramètre est choisi de manière à ce que chaque *Mapper* puisse mettre en mémoire un histogramme distribué contenant au maximum $\mathcal{Q} * \mathbf{t}_{\max}$ tuples. La méthode consiste à construire

des groupes d'enregistrements consécutifs dans un *split* de telle sorte qu'un groupe soit composé d'au plus $\mathbf{t}_{\max}$ enregistrements. Une illustration de cette redistribution est présentée dans la figure Fig. 4.3.

Pour être plus précis, lorsque $\mathbf{t}_{\max}$ est strictement supérieur au nombre d'enregistrements par *split*, la redistribution par *chunks* est équivalente à une redistribution par *split*. En revanche, lorsque $\mathbf{t}_{\max} = 1$, l'histogramme distribué représente la liste, triée selon leur ordre dans le *split*, des tuples de l'histogramme, nécessaire à chaque enregistrement. Généralement, on cherche à maximiser le paramètre $\mathbf{t}_{\max}$ pour les deux raisons suivantes :

- La redistribution d'un tuple de l'histogramme nécessite de mémoriser les groupes dans lesquels il apparaît. Lorsque $\mathbf{t}_{\max} = 1$, le nombre de groupes possible correspond au nombre d'enregistrements total, ce qui pose des problèmes de mémoire. Ils peuvent être néanmoins gérés au prix d'un surcoût de communication ;
- Plus $\mathbf{t}_{\max}$ est grand, plus il est probable qu'il y ait des valeurs des attributs de jointure en commun dans un groupe, ce qui réduit la taille de l'histogramme nécessaire et amortit le coût des lectures/écritures sur les disques.

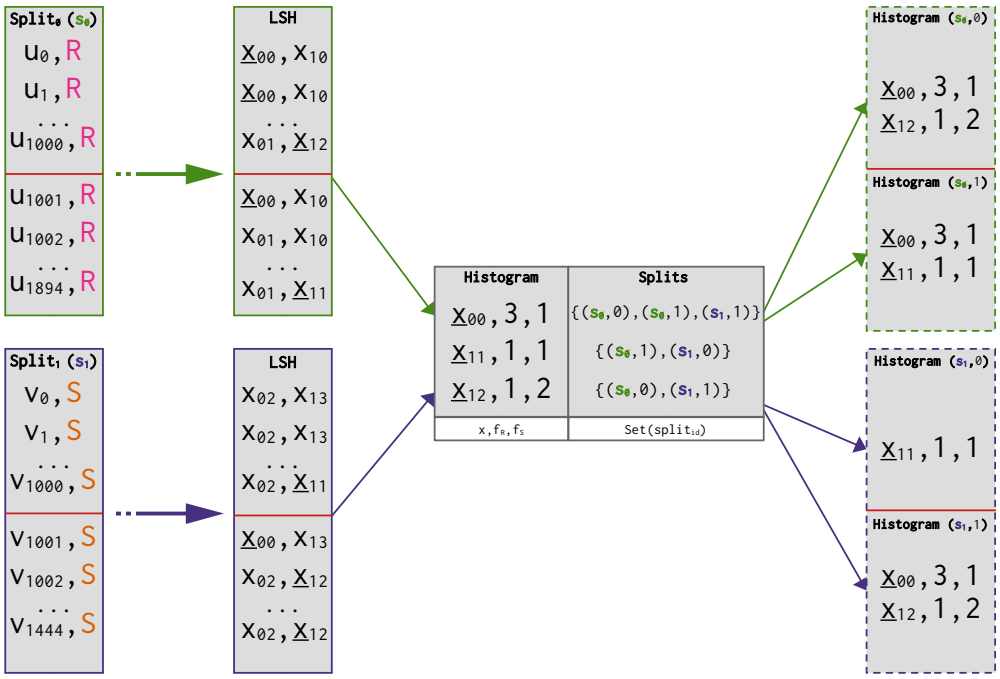



Fig 4.3 : Un exemple de redistribution de l'histogramme en fonction de l'occurrence des valeurs des deux attributs de jointure calculés dans les *splits* de données. Dans cet exemple, chaque *split* est divisé en deux groupes contenant au plus $\mathbf{t}_{\max} = 1000$ enregistrements. Les valeurs soulignées font partie l'histogramme global puisqu'elles sont présentes dans les deux ensembles de données.

En pratique, cette redistribution nécessite une phase de communication supplémentaire pour répartir chaque entrée de l'histogramme vers l'ensemble des *splits* (ou *chunks*) qui en ont besoin pour traiter la jointure. Cette phase de communication n'est possible que par l'ajout d'un *job* dans le modèle MapReduce.

Par souci de simplicité, nous décrivons par la suite les différentes étapes de l'algorithme *MRS-join* en utilisant un *split* comme unité de référence. L'analyse dans notre modèle de coût utilisera également cette hypothèse. Néanmoins, l'utilisation de *chunks* à la place des *splits* pour redistribuer l'histogramme n'ajoute aux modèles de coût qu'un facteur constant dépendant de $\mathbf{t}_{\max}$ et du nombre d'enregistrements moyen dans un *split*.

### 4.1.3 Les schémas de communication pour l'auto-jointure

Une fois l'histogramme distribué, il faut pouvoir l'utiliser pour gérer les valeurs des attributs de jointure posant des difficultés ; nous avons déjà présenté les schémas de communication pour la jointure entre deux ensembles de données (cf. section 2.3.4.a), nous les adaptons maintenant pour le cas de l'auto-jointure. Dans ce contexte et en prenant une valeur donnée $x$ de l'attribut de jointure, les schémas de communication redistribueront les blocs d'enregistrements en fonction de l'un des deux cas suivants :

a. $\mathbf{f}_x < \mathbf{f}_{max}$ : les enregistrements ayant $x$ comme valeur pour un attribut de jointure sont peu nombreux et ils peuvent être stockés en mémoire d'une seule tâche **reduce** ; ils sont donc partitionnés et distribués sur les *Reducers* en utilisant une simple méthode de hachage,

b. $\mathbf{f}_{max} \leqslant \mathbf{f}_x$ : la valeur $x$ est trop fréquente, ce sont généralement ces valeurs qui ont un effet important sur le déséquilibre de la charge. Les enregistrements correspondant à cette valeur sont distribués en utilisant des schémas de communication randomisés et spécifiques reposant sur une méthode de partition/réplication, comme l'illustre la figure Fig. 4.4.

De façon similaire à la jointure entre deux ensembles de données, la fonction **reduce** calcule la jointure en utilisant l'algorithme suivant lorsqu'une valeur est trop fréquente et que les schémas de communication sont utilisés :

1. Stocker en mémoire le bloc partitionné,
   c'est-à-dire le bloc de données correspondant à la ligne $\text{Row}_{\text{id}}$ 0 (cf. Fig. 4.4),

2. Calculer l'auto-jointure du bloc stocké,

3. Calculer la jointure entre le bloc stocké et les blocs répliqués,
   c'est-à-dire la jointure entre le bloc correspondant à la ligne $\text{Row}_{\text{id}}$ 0 et ceux des lignes $\text{Row}_{\text{id}}$ $i : i \geq 1$.

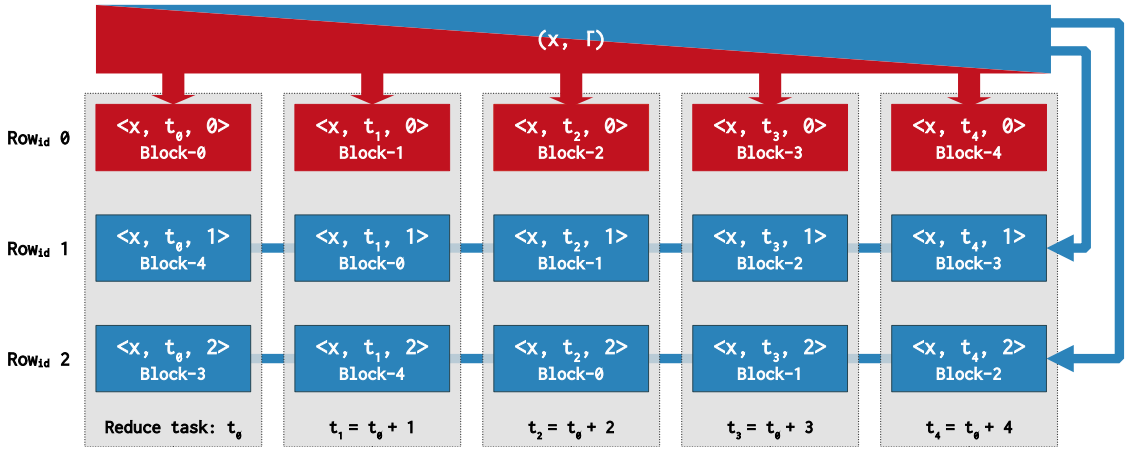



Fig 4.4 : Les schémas de communication dans le cas d'une auto-jointure où le nombre d'enregistrements correspondant à une valeur $x$ d'un attribut de jointure est trop important pour être traité au sein d'une seule tâche **reduce**. Dans cet exemple, les enregistrements correspondant à une valeur $x$ d'un attribut de jointure sont partitionnés en cinq blocs et transmis à cinq tâches différentes. Les blocs sont également répliqués deux fois sur l'ensemble des tâches. Les blocs partitionnés (rouge) sont stockés en mémoire pour calculer la jointure avec les blocs répliqués (bleu). Les schémas de communication générés sont rangés en lignes et en colonnes. Chaque cellule correspond à un bloc. Chaque colonne correspond aux données transmises à une tâche **reduce**. Ces tâches sont identifiées à partir d'un nombre entier aléatoire $t_0$ qui peut être dérivé de la valeur de l'attribut de jointure.

## 4.2 *MRS-join* : Les étapes de l'algorithme

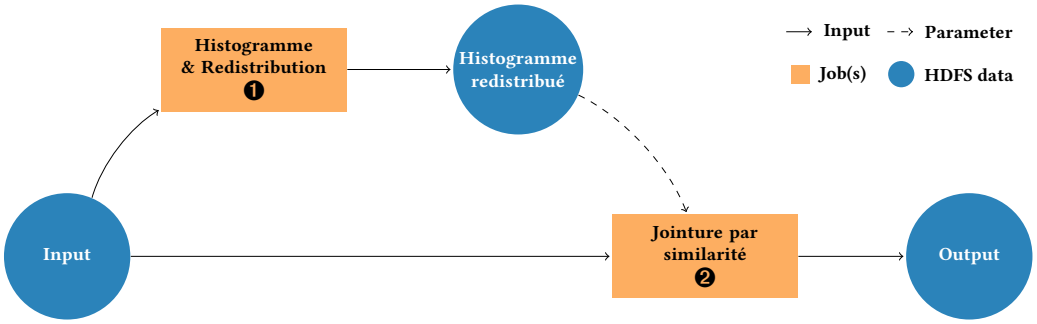



Fig 4.5 : *MRS-join* : les étapes de traitement de la jointure par similarité.

*MRS-join* est un algorithme de jointure par similarité reposant sur le paradigme MapReduce pour traiter de très grands ensembles de données. Afin de réduire drastiquement le nombre de comparaisons, l'algorithme exploite LSH pour calculer les valeurs des attributs de jointure. Pour assurer une charge de calcul équilibrée sur les nœuds de la grappe, des histogrammes distribués et des schémas de communication randomisés sont mises en œuvre lorsque les enregistrements correspondant à une valeur d'un attribut

de jointure sont trop nombreux pour être traités au sein d'une seule tâche **reduce**. L'algorithme se déroule en deux étapes d'une ou deux phases de MapReduce, comme illustré dans la figure Fig. 4.5, où :

❶.a L'histogramme de la jointure est calculé pour réduire le calcul uniquement aux données pertinentes tout en permettant d'identifier les valeurs pouvant conduire à des déséquilibres lors de l'étape de jointure, (c'est-à-dire que les valeurs des attributs de jointure sont calculées en utilisant LSH pour chaque enregistrement, ainsi que leurs fréquences correspondantes),

❶.b L'histogramme de la jointure est redistribué en fonction de l'occurrence des valeurs de l'attribut de jointure dans les différentes portions de l'entrée,

❷ En utilisant les histogrammes distribués, des schémas de communication efficaces et extensibles sont générés pour équilibrer la charge sur les nœuds. La distance est finalement calculée pour tous les couples ayant la même valeur pour un attribut de jointure. Plus précisément, les valeurs des attributs de jointure sont recalculées et, en utilisant les fréquences calculées dans l'étape précédente, les enregistrements sont équitablement répartis sur les nœuds pour calculer les résultats de la jointure par similarité.

Nous supposons que l'ensemble de données en entrée est divisé en *splits* de données, entreposés sur le Hadoop Distributed File System (HDFS) et répliqués sur plusieurs nœuds pour des raisons de fiabilité. Soit un ensemble de données $\mathrm{T} \in \{\mathrm{R}, \mathrm{S}\}$, nous décrivons les coûts de chaque étape de l'algorithme *MRS-join* en utilisant les notations suivantes :

| | |
|---|---|
| $\|\mathrm{T}\|$ | Le nombre d'enregistrements provenant de l'ensemble de données T, |
| $\lvert\mathrm{T}\rvert$ | Le nombre de *splits* de données de T, |
| $\mathrm{T}_i^{\mathrm{map}}$ | Le *fragment* (ensemble de *splits*) de T traité par le *Mapper* $i$, |
| $\mathcal{S}(\mathrm{T})$ | L'ensemble des identifiants de *splits* de T, |
| $\mathrm{T}_j^{\mathrm{split}} \mid j \in \mathcal{S}(\mathrm{T})$ | Le *split* correspondant à l'identifiant $j$, |
| $\mathrm{T}_i^{\mathrm{red}}$ | Le *fragment* de T traité par le *Reducer* $i$, |
| $\mathcal{M}$ | Le nombre de *Mappers*, |
| $\mathcal{R}$ | Le nombre de *Reducers*, |
| $\mathsf{c}_{\mathbf{r/w}}$ | Le coût de lecture/écriture d'une page/*split* de données depuis le Système de Fichiers Distribués (DFS), |
| $\mathsf{c}_{\mathbf{c}}$ | Le coût de communication par page de données, |

| | |
|---|---|
| $\mathcal{Q}$ | Le nombre d'itérations en utilisant une famille LSH, |
| $\mathcal{K}$ | Le nombre de concaténations utilisées par itération de LSH, |
| $\chi(\mathrm{T})$ | L'ensemble des valeurs des attributs de jointure de l'ensemble de données T (cf. Définition 27), |
| $\mathrm{Hist}(\mathrm{T}_i^{\mathrm{map}})$ | L'histogramme du *fragment* $\mathrm{T}_i^{\mathrm{map}}$ ; c'est-à-dire l'ensemble des tuples $(x, \mathbf{f}_x^{\mathrm{R}}, \mathbf{f}_x^{\mathrm{S}})$ correspondant à chaque valeur, $x$, des attributs de jointure, associée à leurs fréquences respectives dans le *fragment* $\mathrm{T}_i^{\mathrm{map}}$, |
| $\mathrm{Hist}_i^{\mathrm{map}}(\mathrm{T})$ | Le *fragment* de l'histogramme Hist(T) traité par le *Mapper* $i$, |
| $\mathrm{Hist}_i^{\mathrm{red}}(\mathrm{T})$ | Le *fragment* de l'histogramme Hist(T) traité par le *Reducer* $i$, |
| $\mathrm{Hist}(\mathrm{R} \bowtie_\sigma \mathrm{S})$ | L'histogramme de la jointure ne contenant que les données pertinentes (cf. Définition 28), |
| $\mathrm{Hist}_j^{\mathrm{split}}(\mathrm{R} \bowtie_\sigma \mathrm{S})$ | La partie de l'histogramme $\mathrm{Hist}(\mathrm{R} \bowtie_\sigma \mathrm{S})$ nécessaire à un *split* $\mathrm{T}_j^{\mathrm{split}}$, c'est-à-dire l'ensemble des tuples de l'histogramme de la jointure correspondant aux valeurs des attributs de jointure $\chi(\mathrm{R}) \cap \chi(\mathrm{S}) \cap \chi(\mathrm{T}_j^{\mathrm{split}})$, |
| $\overline{\mathrm{T}}$ | La partie de l'ensemble de données T ayant au moins une valeur des attributs de jointure faisant partie de l'histogramme de la jointure, |
| $\mathsf{c}_{\mathcal{H}}$ | Le coût de mise en mémoire de l'histogramme distribué dans une table de hachage, |
| $\mathsf{c}_{\mathcal{D}}$ | Le coût d'évaluation de la distance entre deux enregistrements en fonction du seuil $\lambda$ fixé par l'utilisateur, |
| $\overline{\mathrm{R}} \bowtie_\sigma \overline{\mathrm{S}}$ | La jointure traitée par l'algorithme, c'est-à-dire l'ensemble des couples ayant au moins une valeur commune et présente dans R et S pour un attribut de jointure, |
| $\overline{\mathrm{R}} \bowtie_\lambda \overline{\mathrm{S}}$ | Les résultats de la jointure par similarité produit par l'algorithme, cet ensemble est inclus à la fois dans $\overline{\mathrm{R}} \bowtie_\sigma \overline{\mathrm{S}}$ et dans $\mathrm{R} \bowtie_\lambda \mathrm{S}$. LSH étant une méthode approximative, les résultats de l'algorithme représentent une large part des résultats complets. |

### 4.2.1 L'étape de calcul de l'histogramme de la jointure

L'étape de calcul de l'histogramme est décrite dans l'Algorithme 1, et un exemple est illustré dans la figure Fig. 4.6. L'utilisation de LSH implique de partager des fonctions de hachage, ce qui signifie que des nombres aléatoires doivent être partagés entre les nœuds afin d'initialiser les fonctions de hachage LSH et de calculer les valeurs des attributs de jointure de manière déterministe. Par exemple, pour la famille de fonctions LSH sur les trajectoires (cf. section 3.3.3), une grille aléatoire est nécessaire. Nous supposons que ces nombres ont été entreposés sur le système de fichiers distribués, et que les fonctions LSH peuvent être instanciées dès l'initialisation d'une tâche **map**. En pratique, pour les expériences, nous utilisons des fichiers binaires provenant de www.random.org ce qui permet de rendre facilement les expériences reproductibles en partageant ces fichiers.

Par souci de clarté, les exemples ne montrent que deux enregistrements $u$ et $v$ appartenant respectivement à deux *splits* $s_0$ et $s_1$. Pour calculer la fréquence de chaque valeur des attributs de jointure, la phase **map** émet un couple clé-valeur contenant deux entiers et l'identifiant du *split* courant pour chaque enregistrement, et pour chaque valeur des attributs de jointure.

**Algorithme 1 :** Étape de calcul de l'histogramme global (❶.a)

**Map :** <id, R|S, u> → List(<$x_i$, (0|1, 0|1, $\text{split}_{\text{id}}$)>)

**init :**

Lire depuis le HDFS les fonctions LSH.

$\text{split}_{\text{id}} \leftarrow \text{getSplitId}()$ ;

Pour tout $i \in 1, \ldots, \mathcal{Q}$ : // $\mathcal{Q}$ étant le nombre d'itérations

Calculer la valeur $x_i$ correspondant au $i$-ième attribut de jointure de l'enregistrement $u$ en utilisant LSH.

Émettre un couple en fonction de l'ensemble de données d'origine :

- <$x_i$, (1, 0, $\text{split}_{\text{id}}$)> si $u$ appartient à R,
- <$x_i$, (0, 1, $\text{split}_{\text{id}}$)> sinon.

**Reduce :** <$x_i$, List(0|1, 0|1, $\text{split}_{\text{id}}$)> → <$x_i$, ($\mathbf{f}^{\text{R}}_{x_i}$, $\mathbf{f}^{\text{S}}_{x_i}$, Set($\text{split}_{\text{id}}$))>

Calculer les fréquences globales $\mathbf{f}^{\text{R}}_{x_i}$ et $\mathbf{f}^{\text{S}}_{x_i}$ de la valeur $x_i$.

Calculer l'ensemble des $\text{split}_{\text{id}}$ reçus.

Si $\mathbf{f}^{\text{R}}_{x_i} \geq 1$ et $\mathbf{f}^{\text{S}}_{x_i} \geq 1$ :

Émettre un couple clé-valeur <$x_i$, ($\mathbf{f}^{\text{R}}_{x_i}$, $\mathbf{f}^{\text{S}}_{x_i}$, Set($\text{split}_{\text{id}}$))>.

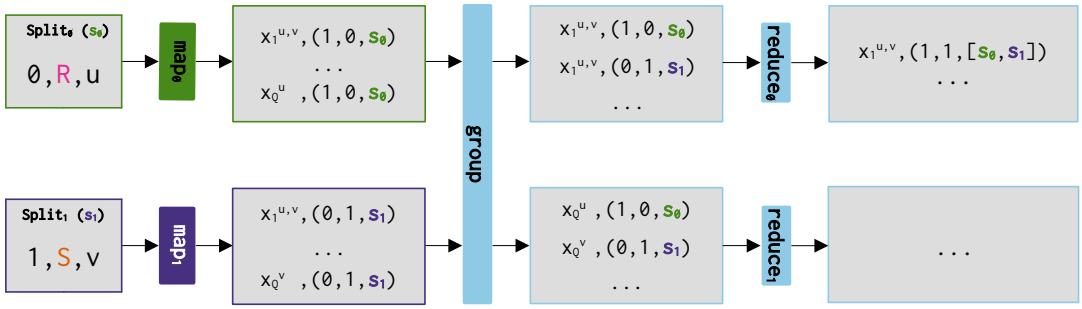



Fig 4.6 : Un exemple d'exécution de l'étape de calcul de l'histogramme : Les deux enregistrements d'entrée $u$ et $v$, identifiés respectivement par 0 et 1, et appartenant aux *splits* $s_0$ et $s_1$, émettent respectivement $\mathcal{Q}$ couples clé-valeur correspondant à leurs $\mathcal{Q}$ attributs de jointure. Chaque attribut de jointure correspond à une itération de LSH. Nous ajoutons en exposant les enregistrements ayant la même valeur. La valeur $x_1^{u,v}$ signifie donc que les enregistrements $u$ et $v$ ont la même valeur pour le premier attribut de jointure. Dans les exemples suivants, nous ne suivrons que cette valeur par simplicité.

### a) Analyse du coût de l'étape du calcul de l'histogramme

Sur le nœud $i$, le *Mapper* lit son fragment à un coût $\mathbf{c_{r/w}} * (|\mathrm{R}_i^{\mathrm{map}}| + |\mathrm{S}_i^{\mathrm{map}}|)$. Généralement, le coût de calcul des $\mathcal{Q}$ valeurs des attributs de jointure dépend de la longueur des enregistrements. En effet, plusieurs optimisations (cf. section 3.3.2) permettent de calculer plusieurs valeurs en une seule passe sur un enregistrement. Soit $\|\tilde{u}\|$ la longueur moyenne des enregistrements d'entrée, le terme $\mathcal{O}\left(\|\tilde{u}\| * (\|\mathrm{R}_i^{\mathrm{map}}\| + \|\mathrm{S}_i^{\mathrm{map}}\|)\right)$ représente le coût de traitement de tous les enregistrements dans le *fragment*. Les données intermédiaires sont ensuite triées sur le *Mapper* $i$, le coût de cette étape est représenté par le terme $\mathcal{O}\left((\mathcal{Q} * (\|\mathrm{R}_i^{\mathrm{map}}\| + \|\mathrm{S}_i^{\mathrm{map}}\|)) * \log\left(\mathcal{Q} * (\|\mathrm{R}_i^{\mathrm{map}}\| + \|\mathrm{S}_i^{\mathrm{map}}\|)\right)\right)$. Ensuite, les données intermédiaires sont envoyées aux *Reducers*, ce qui correspond au terme $\mathbf{c_c} * (|\mathrm{Hist}(\mathrm{R}_i^{\mathrm{map}})| + |\mathrm{Hist}(\mathrm{S}_i^{\mathrm{map}})|)$. Par conséquent, cette phase **map** nécessite au maximum le coût :

$$\begin{aligned}\mathsf{Time}(1.\mathrm{a.Mapper}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{M}} \mathbf{c_{r/w}} * (|\mathrm{R}_i^{\mathrm{map}}| + |\mathrm{S}_i^{\mathrm{map}}|) + \|\tilde{u}\| * (\|\mathrm{R}_i^{\mathrm{map}}\| + \|\mathrm{S}_i^{\mathrm{map}}\|) \\ + (\mathcal{Q} * (\|\mathrm{R}_i^{\mathrm{map}}\| + \|\mathrm{S}_i^{\mathrm{map}}\|)) * \log\left(\mathcal{Q} * (\|\mathrm{R}_i^{\mathrm{map}}\| + \|\mathrm{S}_i^{\mathrm{map}}\|)\right) \\ + \mathbf{c_c} * (|\mathrm{Hist}(\mathrm{R}_i^{\mathrm{map}})| + |\mathrm{Hist}(\mathrm{S}_i^{\mathrm{map}})|)\Big).\end{aligned}$$

Chaque nœud, $i$, reçoit son fragment de l'histogramme global et calcule la somme des fréquences ainsi que l'union des identifiants des *splits* reçus. Le coût de cette étape est représenté par le terme $\mathcal{O}\left(\|\mathrm{Hist}_i^{\mathrm{red}}(\mathrm{R})\| + \|\mathrm{Hist}_i^{\mathrm{red}}(\mathrm{S})\|\right)$. Chaque *Reducer*, $i$, produit et réplique sur les disques sa partie de l'histogramme de la jointure, ce coût correspond

au terme $\mathsf{c}_{\mathbf{r/w}} * |\mathrm{Hist}_i^{\mathrm{red}}(\mathrm{R} \bowtie_\sigma \mathrm{S})|$. Par conséquent, le coût de cette phase est :

$$\mathsf{Time}(1.\mathrm{a.Reducer}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{R}} \|\mathrm{Hist}_i^{\mathrm{red}}(\mathrm{R})\| + \|\mathrm{Hist}_i^{\mathrm{red}}(\mathrm{S})\| + \mathsf{c}_{\mathbf{r/w}} * |\mathrm{Hist}_i^{\mathrm{red}}(\mathrm{R} \bowtie_\sigma \mathrm{S})| \Big).$$

Finalement, cette étape de calcul de l'histogramme aura donc le coût suivant :

$$\mathsf{Time}(1.\mathrm{a.}) = \mathsf{Time}(1.\mathrm{a.Mapper}) + \mathsf{Time}(1.\mathrm{a.Reducer}).$$

### b) Analyse de l'utilisation de la mémoire dans *MRS-join*

L'algorithme stocke au plus $\mathcal{O}(|\mathrm{R}| + |\mathrm{S}|)$ identifiants de *splits*. Nous rappelons qu'un *split* correspond généralement à 128MB ou 256MB, cet ensemble d'identifiants reste donc très petit. Néanmoins, pour garantir que l'algorithme puisse traiter la jointure par similarité à très grande échelle, on peut observer que, si, pour une valeur d'un attribut de jointure donnée, les fréquences globales sont déjà calculées, l'ensemble des identifiants de *splits* n'a pas besoin de tenir en mémoire. Par conséquent, on peut étendre cet algorithme pour séparer le calcul des fréquences du calcul de l'union des identifiants ; le détail est présenté dans l'Algorithme 2. L'utilisation d'une clé composite permet de trier naturellement les valeurs contenant les fréquences de celles contenant les identifiants de *splits* lors de leur traitement par les *Reducers*. La spécification de la fonction **partition** permet de s'assurer que les clés composites ayant une même valeur soient envoyées à un même *Reducer*. Ce traitement permet d'éviter les problèmes de mémoire quelle que soit la taille des ensembles de données en entrée en utilisant des *splits* ou des *chunks* pour la redistribution de l'histogramme.

**Algorithme 2 :** Étape de calcul de l'histogramme global (❶.a)

**Map :** <id, R|S, u> → List(<$x_i$, (0|1|, 0|1|, |split$_{\text{id}}$)>)

**init :**

Lire depuis le HDFS les fonctions LSH.

split$_{\text{id}}$ ← getSplitId() ;

Pour tout $i \in 1, \ldots, \mathcal{Q}$ : // $\mathcal{Q}$ étant le nombre d'itérations

Calculer la valeur $x_i$ correspondant au $i$-ième attribut de jointure de l'enregistrement $u$ en utilisant LSH.

Émettre un couple en fonction de l'ensemble de données d'origine :

- <($x_i$, 0), (1, 0, ∅)> si $u$ appartient à R,
- <($x_i$, 0), (0, 1, ∅)> sinon.

Émettre un couple en fonction du *split* d'origine :

- <($x_i$, 1), (ε, ε, split$_{\text{id}}$)>

**Partition :** ($x_i$, 0|1) → $x_i$

Partitionner les clefs composites en fonction de la valeur $x_i$ uniquement.

**Reduce :** <($x_i$, 0), List(0|1, 0|1, ∅)>

Calculer les fréquences globales $\mathbf{f}^{\text{R}}_{x_i}$ et $\mathbf{f}^{\text{S}}_{x_i}$ de la valeur $x_i$.

**Reduce :** <($x_i$, 1), List(ε, ε, split$_{\text{id}}$)> → <$x_i$, ($\mathbf{f}^{\text{R}}_{x_i}$, $\mathbf{f}^{\text{S}}_{x_i}$, Set(split$_{\text{id}}$))>

Si $\mathbf{f}^{\text{R}}_{x_i} \geq 1$ et $\mathbf{f}^{\text{S}}_{x_i} \geq 1$ :

Calculer l'ensemble des split$_{\text{id}}$ reçus ; à chaque fois que la taille de l'ensemble dépasse une limite donnée, les instructions suivantes sont exécutées et l'ensemble est vidé.

Émettre un couple clé-valeur <$x_i$, ($\mathbf{f}^{\text{R}}_{x_i}$, $\mathbf{f}^{\text{S}}_{x_i}$, Set(split$_{\text{id}}$))>.

### 4.2.2 L'étape de redistribution de l'histogramme

L'histogramme de la jointure est ensuite redistribué en utilisant l'Algorithme 3. Un exemple d'exécution est illustré dans la figure Fig. 4.7. Pour un tuple de l'histogramme, la phase **map** envoie un couple clé-valeur pour chaque *split* dans lequel il apparaît. L'algorithme partitionne les couples clé-valeur en fonction de l'identifiant d'un *split*, ce qui permet d'assurer que la sortie de chaque tâche **reduce** corresponde à la partie de l'histogramme distribué nécessaire au traitement de la jointure par un *split* lors du calcul de la jointure. Il faut toutefois paramétrer le nombre de tâches **reduce** pour être au moins égal au nombre de *splits* des ensembles de données composant l'entrée, c'est-à-dire |R| + |S|.

**Algorithme 3 :** Étape de redistribution de l'histogramme (❶.b)

**Map :** $<x_i, (\mathbf{f}^{\mathrm{R}}_{x_i}, \mathbf{f}^{\mathrm{S}}_{x_i}, \mathrm{Set}(\mathrm{split}_{\mathrm{id}}))> \rightarrow \mathrm{List}(<\mathrm{split}_{\mathrm{id}}, (x_i, \mathbf{f}^{\mathrm{R}}_{x_i}, \mathbf{f}^{\mathrm{S}}_{x_i})>)$

Pour tout id $\in \mathrm{Set}(\mathrm{split}_{\mathrm{id}})$ :

Émettre un couple $<\mathrm{id}, (x_i, \mathbf{f}^{\mathrm{R}}_{x_i}, \mathbf{f}^{\mathrm{S}}_{x_i})>$.

**Reduce :** $<\mathrm{split}_{\mathrm{id}}, \mathrm{List}(x_i, \mathbf{f}^{\mathrm{R}}_{x_i}, \mathbf{f}^{\mathrm{S}}_{x_i})> \rightarrow \mathrm{Set}(<x_i, \mathbf{f}^{\mathrm{R}}_{x_i}, \mathbf{f}^{\mathrm{S}}_{x_i}>)$

Pour chaque tuple de l'histogramme reçu :

Émettre un couple $<x_i, \mathbf{f}^{\mathrm{R}}_{x_i}, \mathbf{f}^{\mathrm{S}}_{x_i}>$.

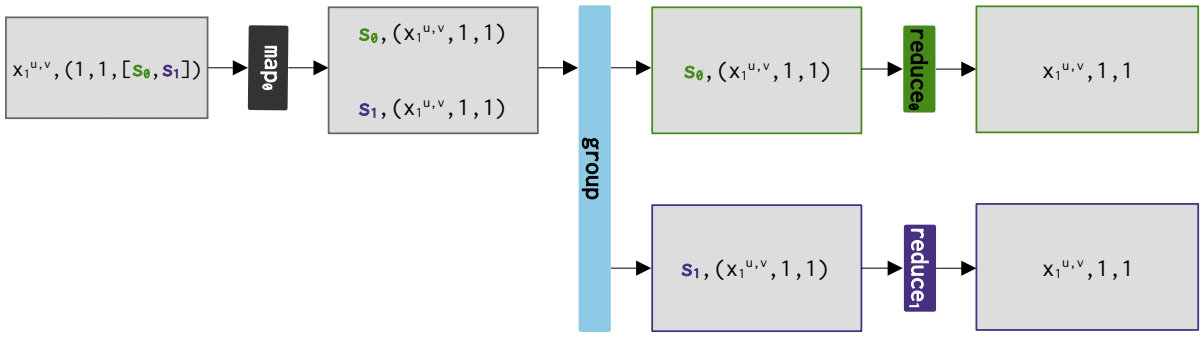



Fig 4.7 : Un exemple d'exécution de l'étape de redistribution de l'histogramme. La valeur $x_i^{u,v}$ était présente dans les *splits* $s_0$ et $s_1$, le tuple de l'histogramme est donc redistribué vers les tâches 0 et 1.

### a) Analyse du coût de l'étape de redistribution

Chaque *Mapper*, $i$, lit son fragment de l'histogramme pour un coût noté $\mathsf{c}_{\mathbf{r/w}} * |\mathrm{Hist}_i^{\mathrm{map}}(\mathrm{R} \bowtie_\sigma \mathrm{S})|$. Pour chaque tuple dans l'histogramme, au plus $(|\mathrm{R}| + |\mathrm{S}|)$ couples clé-valeur sont émis. Cette situation se présente lorsque la valeur d'un attribut de jointure apparaît dans l'ensemble des *splits*. En conséquence, le coût de cette étape correspond au terme $\mathcal{O}\left(\|\mathrm{Hist}_i^{\mathrm{map}}(\mathrm{R} \bowtie_\sigma \mathrm{S})\| * (|\mathrm{R}| + |\mathrm{S}|)\right)$. Les données sur le *Mapper* $i$ sont ensuite triées. Le terme $\mathcal{O}\left((\|\mathrm{Hist}_i^{\mathrm{map}}(\mathrm{R} \bowtie_\sigma \mathrm{S})\| * (|\mathrm{R}| + |\mathrm{S}|)) * \log\left(\|\mathrm{Hist}_i^{\mathrm{map}}(\mathrm{R} \bowtie_\sigma \mathrm{S})\| * (|\mathrm{R}| + |\mathrm{S}|)\right)\right)$ correspond au coût de cette étape. Les données sont ensuite transmises aux *Reducers*, ce qui représente le coût $\mathsf{c}_{\mathbf{c}} * |\mathrm{Hist}_i^{\mathrm{map}}(\mathrm{R} \bowtie_\sigma \mathrm{S})| * (|\mathrm{R}| + |\mathrm{S}|)$. Par conséquent, le coût de cette phase **map** nécessite au maximum :

$$\begin{aligned}\mathsf{Time}(1.\mathrm{b.Mapper}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{M}} \mathsf{c}_{\mathbf{r/w}} * |\mathrm{Hist}_i^{\mathrm{map}}(\mathrm{R} \bowtie_\sigma \mathrm{S})| + \|\mathrm{Hist}_i^{\mathrm{map}}(\mathrm{R} \bowtie_\sigma \mathrm{S})\| * (|\mathrm{R}| + |\mathrm{S}|) \\ + (\|\mathrm{Hist}_i^{\mathrm{map}}(\mathrm{R} \bowtie_\sigma \mathrm{S})\| * (|\mathrm{R}| + |\mathrm{S}|)) * \log\left(\|\mathrm{Hist}_i^{\mathrm{map}}(\mathrm{R} \bowtie_\sigma \mathrm{S})\| * (|\mathrm{R}| + |\mathrm{S}|)\right) \\ + \mathsf{c}_{\mathbf{c}} * |\mathrm{Hist}_i^{\mathrm{map}}(\mathrm{R} \bowtie_\sigma \mathrm{S})| * (|\mathrm{R}| + |\mathrm{S}|)\Big).\end{aligned}$$

Chaque tâche **reduce** traitée sur le nœud $i$ correspond à l'histogramme distribué nécessaire à un *split*. Étant donné que le nombre de tâches **reduce** est fixé pour être égal au nombre de *splits*, chacune de ces tâches a un identifiant $j$ tel que $j \in \mathcal{S}(\mathrm{R}) \cup \mathcal{S}(\mathrm{S})$. Par conséquent, le fragment assigné à un *Reducer* $i$ correspond à un sous-ensemble distinct de tâches **reduce** $\mathcal{S}_i^{\mathrm{red}}(\mathrm{R} \cup \mathrm{S}) \subseteq \mathcal{S}(\mathrm{R}) \cup \mathcal{S}(\mathrm{S})$.

Le terme $\mathcal{O}\left(\|\mathrm{Hist}_j^{\mathrm{split}}(\mathrm{R} \bowtie_\sigma \mathrm{S})\|\right)$ représente le coût de traitement d'une tâche **reduce** $j$. Le terme $\mathsf{c}_{\mathbf{r/w}} * |\mathrm{Hist}_j^{\mathrm{split}}(\mathrm{R} \bowtie_\sigma \mathrm{S})|$ représente le coût d'écriture des couples en sortie sur les disques. Par conséquent, le coût de cette phase est le suivant :

$$\mathsf{Time}(1.\mathrm{b.Reducer}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{R}} \sum_{j \in \mathcal{S}_i^{\mathrm{red}}(\mathrm{R} \cup \mathrm{S})} \|\mathrm{Hist}_j^{\mathrm{split}}(\mathrm{R} \bowtie_\sigma \mathrm{S})\| + \mathsf{c}_{\mathbf{r/w}} * |\mathrm{Hist}_j^{\mathrm{split}}(\mathrm{R} \bowtie_\sigma \mathrm{S})| \Big).$$

Cette étape de redistribution de l'histogramme aura donc le coût suivant :

$$\mathsf{Time}(1.\mathrm{b.}) = \mathsf{Time}(1.\mathrm{b.Mapper}) + \mathsf{Time}(1.\mathrm{b.Reducer}).$$

**b) Analyse de l'utilisation de la mémoire**

L'utilisation de l'Algorithme 2 permet d'éviter les problèmes de mémoire, cependant il peut engendrer des duplicatas dans l'histogramme distribué de chaque *split*. En conséquence, nous adaptons l'Algorithme 3 de redistribution pour les supprimer. Une façon simple de procéder est de calculer l'ensemble des tuples de l'histogramme reçu lors de la phase **reduce**. Cet ensemble doit tenir en mémoire naturellement, puisqu'il correspond à ce que les *Mappers* devront également mettre en mémoire lors de l'étape de jointure. Dans le cas contraire, l'utilisation de *chunks* le permettra.

### 4.2.3 L'étape de calcul de la jointure par similarité

Cette dernière étape permet de calculer la jointure par similarité en utilisant l'histogramme distribué pour adapter la charge des *Reducers* en utilisant des schémas de communication spécifiques (cf. sections. 2.3.4.a et 4.1.3) lorsqu'ils sont nécessaires. Les détails sont présentés dans l'Algorithme 4 et une illustration de l'exécution est fournie dans la figure Fig. 4.8.

Lors de la phase **map**, l'ensemble des valeurs des attributs de jointure sont calculées. Pour chacune de ces valeurs, on regarde l'histogramme distribué pour déterminer les schémas de communication à utiliser.

**Algorithme 4 :** Étape de calcul de la jointure par similarité (❷)

**Map :** <id, R|S, u> → <($x_i$, reduceId, rowId), (u, $\overline{\chi}(u)$)>

- **init :**
  - Lire depuis le HDFS les fonctions LSH.
  - Lire depuis le HDFS l'histogramme distribué $\text{Hist}_j^{\text{split}}(\text{R} \bowtie_\sigma \text{S})$ où $j$ correspond à l'identifiant du *split* courant.
  - Construire une table de hachage en utilisant les valeurs des attributs de jointure comme clé et les fréquences comme valeur.
- Initialiser un tableau $\overline{\chi}(u)$ de taille $\mathcal{Q}$.
- Pour tout $i \in 1, \ldots, \mathcal{Q}$ :
  - Calculer la valeur $x_i$ correspondant au $i$-ième attribut de jointure de l'enregistrement $u$ en utilisant LSH et la stocker dans le tableau $\overline{\chi}(u)$.
- Pour tout $i \in 1, \ldots, \mathcal{Q}$ :
  - Récupérer la valeur $x_i$ du tableau $\overline{\chi}(u)$
  - Si la valeur $x_i$ est présente dans l'histogramme distribué :
    - Émettre des couples clé-valeur conformément aux modèles de communication décrits précédemment (cf. section 2.3.4.a) et en accord avec les fréquences de l'histogramme.
    - Transmettre également le tableau $\overline{\chi}(u)$ pour chaque couple clé-valeur.

**Partition :** <($x_i$, reduceId, rowId), (u, $\overline{\chi}(u)$)> → Integer

- Rediriger chaque couple en fonction de la valeur $x_i$ ou de la tâche de destination "reduceId", générée aléatoirement durant la phase de **map**, en suivant les différents scénarios des schémas de communication.

**Reduce :** <($x_i$, reduceId, rowId), (u, $\overline{\chi}(u)$)> → ($\text{id}_u$, $\text{id}_v$)

- Calculer la jointure par similarité en utilisant les modèles de communication.
- Pour chaque couple d'enregistrements ($u$, $v$) distincts provenant de R × S :
  - Calculer les valeurs communes pour les différents attributs de jointure à l'aide des tableaux $\overline{\chi}(u)$ et $\overline{\chi}(v)$.
  - Si $x_i$ est la plus petite valeur commune :
    - Si $\mathcal{D}(u, v) \leq \lambda$ :
      - Émettre un couple ($\text{id}_u$, $\text{id}_v$).

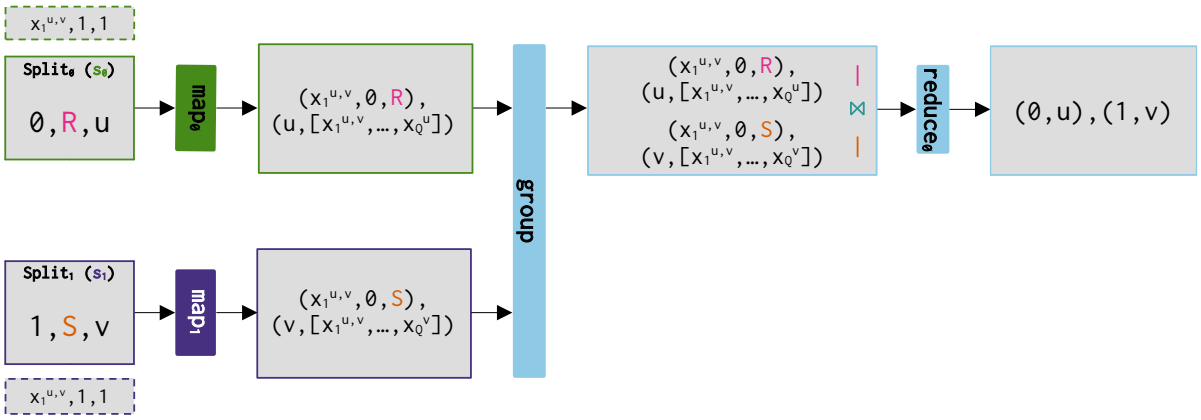



Fig 4.8 : Un exemple d'exécution de l'étape de calcul de la jointure par similarité. Les schémas de communication spécifiques ne sont pas utilisés dans cet exemple puisque la valeur $x_1^{u,v}$ n'apparaît que deux fois dans l'entrée. La valeur du 'reducerId' des clés transmises est donc mise à 0 et le 'rowId' prend la valeur de l'ensemble de données R ou S pour trier les données intermédiaires (cf. section 2.3.4.a).

Les distances sont enfin calculées entre les couples d'enregistrements ayant une valeur d'un attribut de jointure en commun, c'est-à-dire les couples identifiés comme similaires par une famille de fonctions LSH. L'utilisation d'heuristiques permet d'accélérer le coût de ces traitements, puisqu'il est uniquement nécessaire de décider si la distance d'un couple est inférieure au seuil $\lambda$. Il convient de noter que lors de la phase **reduce**, la distance entre deux enregistrements ne peut être calculée qu'une seule et unique fois. En effet, l'ensemble des valeurs des attributs de jointure est également transmis aux *Reducers* et associé à chaque enregistrement, ce qui permet de décider globalement, et sans communication supplémentaire, au sein de quelle tâche **reduce** s'effectuera le calcul.

Une autre implémentation a été proposée par Aumüller et al. [AC22]. L'idée est de recalculer ces valeurs durant la phase de **reduce** en réutilisant les fonctions LSH stockées sur le HDFS, ce qui permet réduire les coûts de communication. Cependant, nos expériences ont montré que cette implémentation s'avère trop coûteuse, et dégrade les performances, car elle nécessite de recalculer l'ensemble des valeurs jusqu'à $\mathcal{Q}$ fois.

### a) Analyse du coût de l'étape de jointure

Chaque *Mapper*, $i$, lit son fragment pour un coût $\mathsf{c}_{\mathbf{r/w}} * (|\mathrm{R}_i^{\mathrm{map}}| + |\mathrm{S}_i^{\mathrm{map}}|)$. Par souci de simplicité, nous notons le fragment du *Mapper* $i$ par $\mathrm{T}_i^{\mathrm{map}} = \mathrm{R}_i^{\mathrm{map}} + \mathrm{S}_i^{\mathrm{map}}$. Pour chaque *split* de son fragment, identifié par $j \in \mathcal{S}(\mathrm{T}_i^{\mathrm{map}})$, le *Mapper* lit l'histogramme distribué correspondant, ce qui correspond au coût $\mathsf{c}_{\mathbf{r/w}} * |\mathrm{Hist}_j^{\mathrm{split}}(\mathrm{R} \bowtie_\sigma \mathrm{S})|$. L'histogramme distribué est aussi mis en mémoire dans une table de hachage pour un coût représenté par le terme $\mathsf{c}_{\mathcal{H}} * \|\mathrm{Hist}_j^{\mathrm{split}}(\mathrm{R} \bowtie_\sigma \mathrm{S})\|$. Pour chaque enregistrement de son *fragment*, il

calcule les valeurs des attributs de jointure, ce qui correspond au coût $\mathcal{O}\left(\|\tilde{u}\| * \|\mathrm{T}_i^{\mathrm{map}}\|\right)$. Les données émises sont ensuite triées sur chaque *Mapper* ; le coût de cette étape correspond au terme $\mathcal{O}\left(\left(\mathcal{Q} * \|\overline{\mathrm{T}}_i^{\mathrm{map}}\|\right) * \log\left(\mathcal{Q} * \|\overline{\mathrm{T}}_i^{\mathrm{map}}\|\right)\right)$. Les données sont ensuite transmises aux *Reducers*, ce qui correspond au terme $\mathsf{c}_{\mathbf{c}} * |\overline{\mathrm{T}}_i^{\mathrm{map}}|$. Par conséquent, le coût de cette étape est le suivant :

$$\begin{aligned}\mathsf{Time}(\mathsf{2.Mapper}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{M}} \sum_{j \in \mathcal{S}(\mathrm{T}_i^{\mathrm{map}})} \Big(\mathsf{c}_{\mathbf{r/w}} * |\mathrm{Hist}_j^{\mathrm{split}}(\mathrm{R} \bowtie_\sigma \mathrm{S})| + \mathsf{c}_{\mathcal{H}} * \|\mathrm{Hist}_j^{\mathrm{split}}(\mathrm{R} \bowtie_\sigma \mathrm{S})\|\Big) \\ + \ \mathsf{c}_{\mathbf{r/w}} * |\mathrm{T}_i^{\mathrm{map}}| + \|\tilde{u}\| * \|\mathrm{T}_i^{\mathrm{map}}\| \\ + \left(\mathcal{Q} * \|\mathrm{T}_i^{\mathrm{map}}\|\right) * \log\left(\mathcal{Q} * \|\mathrm{T}_i^{\mathrm{map}}\|\right) + \mathcal{Q} * \mathsf{c}_{\mathbf{c}} * |\overline{\mathrm{T}}_i^{\mathrm{map}}| \Big).\end{aligned}$$

Chaque *Reducer*, $i$, reçoit son *fragment* de données et calcule la jointure par similarité, c'est-à-dire le calcul des distances. Ce coût est représenté par le terme $\mathsf{c}_{\mathcal{D}} * \|\overline{\mathrm{R}}_i^{\mathrm{red}} \bowtie_\sigma \overline{\mathrm{S}}_i^{\mathrm{red}}\|$. Les résultats de la jointure par similarité sont ensuite produits en sortie sur chaque *Reducer*, représenté par le terme $\mathsf{c}_{\mathbf{r/w}} * |\overline{\mathrm{R}}_i^{\mathrm{red}} \bowtie_\lambda \overline{\mathrm{S}}_i^{\mathrm{red}}|$. En conséquence, le coût de cette étape est le suivant :

$$\mathsf{Time}(\mathsf{2.Reducer}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{R}} \mathsf{c}_{\mathcal{D}} * \|\overline{\mathrm{R}}_i^{\mathrm{red}} \bowtie_\sigma \overline{\mathrm{S}}_i^{\mathrm{red}}\| + \mathsf{c}_{\mathbf{r/w}} * |\overline{\mathrm{R}}_i^{\mathrm{red}} \bowtie_\lambda \overline{\mathrm{S}}_i^{\mathrm{red}}| \Big).$$

Globalement, ce *job* aura le coût suivant :

$$\mathsf{Time}(\mathsf{2.}) = \mathsf{Time}(\mathsf{2.Mapper}) + \mathsf{Time}(\mathsf{2.Reducer}).$$

### b) Analyse du coût de l'algorithme *MRS-join*

Le coût global de l'algorithme *MRS-join* correspond donc à la somme des deux étapes précédentes, soit :

$$\mathsf{Time}(\textit{MRS-join}) = \mathsf{Time}(\mathsf{1.a.}) + \mathsf{Time}(\mathsf{1.b.}) + \mathsf{Time}(\mathsf{2.}).$$

En utilisant une famille de fonctions LSH, la jointure par similarité entre deux ensembles de données R et S nécessite au moins le coût suivant :

$$\mathbf{bound_{inf}} = \Omega\Bigg( \max_{i=0}^{\mathcal{M}} \Big( \Big(\mathsf{c_{r/w}} + \mathcal{Q} * \mathsf{c_c}\Big) * (|\mathrm{R}_i^{\mathrm{map}}| + |\mathrm{S}_i^{\mathrm{map}}|) + (\mathcal{Q} * (|\mathrm{R}_i^{\mathrm{map}}| + |\mathrm{S}_i^{\mathrm{map}}|)) * \log(\mathcal{Q} * (|\mathrm{R}_i^{\mathrm{map}}| + |\mathrm{S}_i^{\mathrm{map}}|)) \Big), \max_{i=0}^{\mathcal{R}} \Big( \mathsf{c}_{\mathcal{D}} * \|\mathrm{R}_i^{\mathrm{red}} \bowtie_{\sigma} \mathrm{S}_i^{\mathrm{red}}\| + \mathsf{c_{r/w}} * |\overline{\mathrm{R}}_i^{\mathrm{red}} \bowtie_{\lambda} \overline{\mathrm{S}}_i^{\mathrm{red}}| \Big) \Bigg).$$

Le terme $\Big(\mathsf{c_{r/w}} + \mathcal{Q} * \mathsf{c_c}\Big) * (|\mathrm{R}_i^{\mathrm{map}}| + |\mathrm{S}_i^{\mathrm{map}}|)$ représente le coût de lecture, de traitement en utilisant LSH et de transmission des données vers les *Reducers* pour le *Mapper i*. Le terme $\mathcal{O}\left((\mathcal{Q} * (|\mathrm{R}_i^{\mathrm{map}}| + |\mathrm{S}_i^{\mathrm{map}}|)) * \log(\mathcal{Q} * (|\mathrm{R}_i^{\mathrm{map}}| + |\mathrm{S}_i^{\mathrm{map}}|))\right)$ correspond à la phase de tri des données effectuée par le *Mapper i*. Du côté des *Reducers*, le coût des calculs de distances est représenté par le terme $\mathsf{c}_{\mathcal{D}} * \|\mathrm{R}_i^{\mathrm{red}} \bowtie_{\sigma} \mathrm{S}_i^{\mathrm{red}}\|$. Les résultats sont ensuite stockés sur l'HDFS ; ce coût correspond au terme $\mathsf{c_{r/w}} * |\overline{\mathrm{R}}_i^{\mathrm{red}} \bowtie_{\lambda} \overline{\mathrm{S}}_i^{\mathrm{red}}|$.

L'algorithme *MRS-join* est asymptotiquement optimal par rapport à un algorithme reposant sur LSH lorsque l'équation suivante est satisfaite :

$$\sum_{j \in \mathcal{S}(\mathrm{T}_i^{\mathrm{map}})} \mathsf{c_{r/w}} * |\mathrm{Hist}_j^{\mathrm{split}}(\mathrm{R} \bowtie_{\sigma} \mathrm{S})| \leq \max\Bigg( \max_{i=0}^{\mathcal{M}} \Big( \mathsf{c_{r/w}} * (|\mathrm{R}_i^{\mathrm{map}}| + |\mathrm{S}_i^{\mathrm{map}}|) + (\mathcal{Q} * \|\mathrm{T}_i^{\mathrm{map}}\|) * \log(\mathcal{Q} * \|\mathrm{T}_i^{\mathrm{map}}\|) \Big), \max_{i=0}^{\mathcal{R}} \Big( \mathsf{c}_{\mathcal{D}} * \|\mathrm{R}_i^{\mathrm{red}} \bowtie_{\sigma} \mathrm{S}_i^{\mathrm{red}}\| + \mathsf{c_{r/w}} * |\overline{\mathrm{R}}_i^{\mathrm{red}} \bowtie_{\lambda} \overline{\mathrm{S}}_i^{\mathrm{red}}| \Big) \Bigg).$$

Cette équation signifie que les surcoûts induits par les étapes de calcul de l'histogramme et de sa redistribution sont dominés par les coûts de traitement de la jointure par similarité en utilisant LSH lorsque la taille des histogrammes distribués est plus petite que la taille des enregistrements.

Cette inégalité tient puisque tous les autres termes sont dominés par ceux de $\mathbf{bound_{inf}}$. Plus précisément, le terme $\mathsf{c_{r/w}} * |\mathrm{Hist}_j^{\mathrm{split}}(\mathrm{R} \bowtie_{\sigma} \mathrm{S})|$ correspond au coût de lecture de l'histogramme distribué nécessaire au traitement d'un *split j*. Par définition, le coût de l'étape de l'histogramme est nécessairement dominé par le coût de l'étape de jointure, car seules les données des histogrammes sont redistribuées et non les données en entrées, et que le nombre d'itérations est le même pour ces deux étapes. Formellement, la taille de l'histogramme nécessaire à un *fragment* représente au plus $\mathcal{O}(\mathcal{Q} * (\|\mathrm{R}_i^{\mathrm{map}}\| + \|\mathrm{S}_i^{\mathrm{map}}\|))$ : ce terme se retrouve également dans la $\mathbf{bound_{inf}}$. Ce raisonnement s'applique aussi pour

l'étape de redistribution. Il reste cependant les coûts de lecture/écriture de l'histogramme sur les disques pour ces deux étapes qui ne peuvent pas être réduits puisque le nombre d'itérations lié à la famille LSH ne se retrouve pas sur un coût de lecture/écriture dans la $\mathbf{bound_{inf}}$. Ces coûts sont dominés par la somme $\sum_{j \in \mathcal{S}(\mathrm{T}_i^{\mathrm{map}})} \mathsf{c}_{\mathbf{r/w}} * |\mathrm{Hist}_j^{\mathrm{split}}(\mathrm{R} \bowtie_\sigma \mathrm{S})|$.

Plus généralement, cette équation est satisfaite pour les deux raisons suivantes :

- Le coût de lecture/écriture des histogrammes pour un *fragment* dépend principalement du nombre d'itérations LSH utilisé par enregistrement. Ce nombre étant généralement très petit, le coût de lecture/écriture des histogrammes est dominé par le coût de lecture/écriture du *fragment* ;
- Par nature, la taille de l'histogramme d'un *fragment* est nettement inférieure à celle de la jointure, c'est-à-dire aux données transmises aux *Reducers*, ainsi qu'aux résultats produits sur les disques.

### c) Les filtrages supplémentaires dans *MRS-join*

Pour éviter de calculer toutes les distances dans une partition LSH et réduire le nombre de comparaisons, plusieurs filtres additionnels peuvent être mis en œuvre. En général, ils sont basés soit sur la distance utilisée, soit sur LSH.

Dans le cas des ensembles et des séquences, un simple filtre sur la taille permet d'éliminer un très grand nombre de comparaisons à moindre coût. Pour qu'une stratégie soit efficace, le temps investi dans ce filtre doit être compensé par la réduction du nombre de distances calculées. Généralement, pour les distances se calculant en temps quadratique, toute heuristique est à considérer. Pour les ensembles, Mann et al. [MAB16] note que la vérification de la distance de Jaccard est si rapide que les filtres sophistiqués sont trop lents et non rentables.

Une deuxième catégorie de filtres repose sur LSH, ce qui engendre des coûts assez importants. Néanmoins, lorsqu'il s'agit d'une valeur d'un attribut de jointure très fréquente, ce coût peut être largement amorti. L'idée est de re-partitionner les blocs stockés en mémoire (cf. sections. 2.3.4.a et 4.1.3) au sein des tâches **reduce** en $\mathrm{F}_{\mathcal{Q}}$ parties à l'aide de la même famille de fonctions LSH. Le nombre de concaténations, noté $\mathrm{F}_{\mathcal{K}}$, est choisi à partir du nombre d'itérations et de l'espérance désirée par l'utilisateur $\mathrm{F}_{\mathbb{E}}$, c'est-à-dire formellement, $\mathrm{F}_{\mathcal{Q}} = \log_{1/p_1}(\mathrm{F}_{\mathcal{Q}}/\mathrm{F}_{\mathbb{E}})$. Lorsque $\mathcal{K} < \mathrm{F}_{\mathcal{K}}$, nous nous attendons à ce que le nombre de comparaisons soit considérablement réduit, ce qui entraîne un gain de temps. Cependant, ce gain de temps n'est garanti que si le coût de filtrage et de comparaison des enregistrements restants est inférieur au coût de comparaison de tous les enregistrements.

Une proposition similaire [AC22] a été présentée dans la littérature à la suite de cet algorithme, ce qui enrichit l'analyse théorique. Principalement, pour la qualité du résultat attendu, l'utilisateur ne choisi plus l'espérance $\mathbb{E}$ (cf. section 3.3.6), mais le **rappel** désiré, noté $\delta$, ce qui conduit à ajouter un facteur $\ln(1/(1-\delta))$ au nombre d'itérations, c'est-à-dire à $1/p_1^{\mathcal{K}}$. En ajoutant le filtrage précédent, le facteur devra être de $\ln\left(1/(1-\sqrt{\delta})\right)$ pour que la probabilité aux deux niveaux soit de $\sqrt{\delta}$.

De plus, Aumüller et al. [AC22] propose d'utiliser les esquisses d'objets (cf. section 3.3.5) pour décider rapidement si deux objets sont proches du point de vue de la distance et du seuil. Nous évaluons ce filtrage lors des expériences.

## 4.3 L'évaluation de l'algorithme *MRS-join*

Dans cette section, nous discutons de l'efficacité et de la robustesse de notre analyse théorique en expérimentant l'algorithme *MRS-join* à la fois sur des trajectoires et sur des ensembles. Les jeux de données proviennent du monde réel et de générateurs de données. Nous mesurons le **rappel** et la **précision** du filtrage (cf. section 3.3.7) ainsi que les performances en termes de temps d'exécution et de données transmises de l'algorithme *MRS-join* en comparaison à l'état de l'art. Les expériences en utilisant Apache Hadoop 3.3.4 sur une grappe de quatre machines. Chaque machine possède les caractéristiques suivantes : Intel(R) Xeon(R) CPU E5-2650 @2.60 GHz, 64 GB de mémoire et deux HDD d'une capacité de 1 TB chacun. Les nœuds sont interconnectés par un réseau de 1 Gb/s. La compression lors de la phase **map**, ainsi que la compression lors de la sortie en utilisant le codec Snappy.

### 4.3.1 Un générateur de données synthétiques

Le générateur de données permet de spécifier le nombre de résultats de la jointure par similarité à l'avance, ce qui permet d'éviter de devoir utiliser un algorithme exhaustif pour vérifier la complétude des résultats. Nous rappelons que les algorithmes exhaustifs échouent régulièrement lorsque les ensembles de données sont très volumineux.

### 4.3.2 La génération de trajectoires pour une jointure entre deux ensembles de données

Pour ce faire, le générateur prend en paramètres le nombre de clusters et leurs tailles dans les ensembles de données R et S, ainsi qu'un seuil donné. Pour chaque cluster, l'algorithme génère un modèle (une trajectoire) aléatoirement. Les éléments du cluster prennent ce modèle et le modifient en fonction du seuil donné. Le jeu de données est

complété par du bruit, c'est-à-dire des trajectoires aléatoires qui ne produiront aucun résultat de jointure par similarité avec une très forte probabilité.

Nous utilisons par la suite trois ensembles de données. Dans le plus petit ensemble de données (à savoir 5 GB), il y a 5 000 clusters de taille 200 répartis également dans R et S, et 2 000 000 de trajectoires aléatoires supplémentaires. Pour des ensembles de données plus grands, nous augmentons le nombre de clusters et le nombre de trajectoires aléatoires du même facteur de manière que le nombre de résultats produits suive le même facteur. Toutes les trajectoires générées sont bidimensionnelles et comprennent en moyenne 50 points espacés selon le seuil donné. Dans nos tests, nous avons utilisé trois ensembles de données distincts, à savoir : un ensemble de données "petit" de 5 GB avec 50 millions de trajectoires similaires, un ensemble de données "moyen" de 50 GB avec 500 millions de trajectoires similaires, et un ensemble de données "grand" de 100 GB avec 1 milliard de trajectoires similaires.

### 4.3.3 La création de jeux de données d'ensembles pour une auto-jointure

Nous évaluons le traitement de l'auto-jointure sur des ensembles en faisant varier la distance seuil, le générateur ne prend donc que deux paramètres, à savoir : le nombre de clusters et la taille des clusters. Nous n'ajoutons pas de bruit à ces jeux de données. À chaque nouveau cluster, un ensemble de taille 100 en moyenne est aléatoirement construit à partir d'un univers de taille $10^9$. Comme pour les trajectoires, les éléments du cluster prennent également ce modèle et le modifient. Cependant, le nombre de modifications n'est plus paramétré, mais dépend de leurs positions dans le cluster, c'est-à-dire que les $\lambda$ % premiers éléments du cluster auront une distance de Jaccard de $\lambda$ avec le modèle. Cela permet de pouvoir faire varier la distance seuil en gardant le même jeu de données. Nous utilisons principalement deux ensembles de données : le "petit" comprenant 1 000 clusters de taille 10 000, ce qui représente 10 GB, et le "grand" de 100 GB comprenant 10 000 clusters.

### 4.3.4 L'évaluation de la jointure par similarité entre deux ensembles de trajectoires

Pour les expériences sur les trajectoires, nous avons utilisé un ensemble de données provenant du monde réel pour tester la qualité des résultats de l'algorithme *MRS-join* et le générateur de données pour tester son extensibilité.

### a) Un ensemble de données de trajets de taxis

Le premier ensemble de données ECML / PKDD (Porto)[1] décrit les trajets de 442 taxis sur une année à Porto. L'ensemble de données contient 1,7 million de trajectoires. Chaque trajectoire correspond au voyage d'un passager avec l'un des taxis. Les emplacements GPS du taxi pendant un trajet ont été enregistrés toutes les 15 secondes au format WGS84. La longueur des trajectoires varie entre 1 km et 15 km. Pour simuler une jointure entre deux ensembles de données, une étiquette aléatoire (R ou S) a été attribuée à chaque trajectoire.

Le tableau Tab. 4.1 présente les performances de l'algorithme *MRS-join* sur l'ensemble de données ECML / PKDD en faisant varier le nombre d'itérations de la famille LSH (cf. section 3.3.3). Nous observons qu'une grande majorité ($\geq 95\%$) des trajectoires similaires ont été générées par l'algorithme *MRS-join*. Nous rappelons qu'il n'existe pas d'algorithme de jointure distribué permettant de traiter la distance de Fréchet dans la littérature (cf. section 3.2.4). Nous remarquons également que le nombre de distances calculées reste très faible en comparaison à algorithme naïf (cf. section 3.2.1) consistant à comparer l'ensemble des trajectoires des ensembles de données R et S et impliquant le calcul d'un produit Cartésien.

| Nombre d'itérations ($Q$) | | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|
| Nombre d'entrées | | 1 704 769 | 1 704 769 | 1 704 769 | 1 704 769 |
| Enregistrements transmis | [$\times 10^6$] | 11 | 22 | 43 | 87 |
| Nombre de résultats | | 17 719 253 | 18 317 307 | 18 467 773 | 18 479 419 |
| Distances calculées | [$\times 10^6$] | 746 | 1 010 | 1 267 | 1 521 |
| Temps d'exécution | [s] | 205 | 262 | 512 | 1 760 |
| Données transmises | [GB] | 6.6 | 14 | 33 | 84 |
| **rappel** | | 0.941 | 0.972 | 0.980 | 0.981 |
| **précision** | | 0.023 | 0.018 | 0.015 | 0.012 |

Tab. 4.1 : Les performances de la jointure entre deux ensembles de données de l'algorithme *MRS-join* sur l'ensemble de données ECML / PKDD (Porto) en faisant varier le nombre d'itérations de LSH et en fixant le seuil à $\lambda = 0.01$. Nous rappelons que le nombre de distances calculées avec un algorithme naïf est de l'ordre de $\sim 1.45 * 10^{12}$.

[1]Ensemble de données ECML / PKDD (Porto).
https://www.kaggle.com/c/pkdd-15-predict-taxi-service-trajectory-i

**b) Le traitement de la jointure par similarité sur des ensembles de données synthétiques**

Nous utilisons maintenant le générateur de données pour tester l'extensibilité de l'algorithme *MRS-join* sur une jointure entre deux ensembles de trajectoires. Le tableau Tab. 4.2 présente les performances en termes de temps d'exécution et de quantité de données transmises, ainsi que l'exhaustivité du résultat produit. Nous remarquons que les temps d'exécution et la quantité de données transmises dépend uniquement de la taille du résultat de la jointure par similarité. Cela est dû au fait que le générateur de données ne produit pas de trajectoires similaires en dehors des clusters. Néanmoins, l'algorithme *MRS-join* parvient à extraire ces clusters de l'ensemble de données, et ce, peu importe la taille de l'entrée, ce qui réduit considérablement le nombre de distances calculées par rapport à un algorithme naïf. Enfin, l'exhaustivité du résultat dépend fortement du nombre d'itérations. Par exemple, pour l'ensemble de données de 5 GB, avec 2 itérations, le **rappel** tombe à 73,7 %, tandis qu'avec 16 itérations, le **rappel** atteint 99,99 %.

| Dataset | [GB] | 5 | 10 | 50 | 100 | 200 |
|---|---|---|---|---|---|---|
| Nombre d'entrées | [$\times 10^6$] | 3 | 6 | 30 | 60 | 120 |
| Enregistrements transmis | [$\times 10^6$] | 7.7 | 15.5 | 77 | 155 | 311 |
| Nombre de résultats | [$\times 10^6$] | 49.7 | 99 | 497 | 993 | 1 987 |
| Temps d'exécution | [s] | 140 | 173 | 537 | 775 | 1523 |
| Données transmises | [GB] | 7 | 13 | 68 | 138 | 275 |
| **rappel** | | 0.9936 | 0.9935 | 0.9935 | 0.9935 | 0.9935 |

Tab. 4.2 : Les performances de la jointure par similarité entre deux ensembles de données de l'algorithme *MRS-join* en utilisant le générateur de données synthétiques. Le nombre d'itérations de LSH utilisées est de $\mathcal{Q} = 8$, le seuil étant fixé à $\lambda = 10$.

### 4.3.5 L'évaluation de l'auto-jointure par similarité d'ensembles

Dans cette sous-section, nous évaluons l'algorithme *MRS-join* en utilisant les paramètres présentés dans le tableau Tab. 4.3. Nous ne présentons pas d'expériences concernant ces paramètres pour plusieurs raisons, notamment parce que plusieurs de ces paramètres sont interdépendants et dépendent également de l'ensemble de données utilisé. Nous avons néanmoins sélectionné les paramètres de façon à avoir les meilleurs compromis en termes de performances et de qualité des résultats.

Le nombre de processeurs et le paramètre $\rho$ permettent de déterminer le nombre de concaténations (cf. section 3.3.6). Nous paramétrons le nombre de processeurs pour qu'il

soit égal au nombre de tâches pouvant s'exécuter sur la grappe en parallèle. En général, ce paramètre entraîne une augmentation des coûts communication, puisque le nombre de concaténations augmente, ainsi que le nombre d'itérations. Néanmoins, pour certains ensembles de données, l'augmentation des coûts de communication peut être compensé par une forte réduction du nombre de comparaisons, ce qui améliore les performances en termes de temps d'exécution de l'algorithme. Généralement, cela est possible lorsque l'ensemble de données contient une forte concentration d'enregistrements proches du seuil de similarité, c'est-à-dire dans l'intervalle $[\lambda, c * \lambda]$. Le tableau Tab. 4.3 présente également le nombre d'itérations et de concaténations de MinHash pour l'algorithme *MRS-join* résultant des paramètres fixés dans le tableau Tab. 4.3.

L'espérance de collision est fixée à 3 (cf. section 3.3.6). En réutilisant l'analyse précédente (cf. section 4.2.3.c), cela permet de garantir à l'algorithme un **rappel** de 0.95, ce qui signifie que 95 % des couples similaires sont retrouvés par l'algorithme *MRS-join*. La taille des blocs est un facteur d'équilibre entre la répartition des charges et l'ajout de coûts de communication. Dans la littérature de l'équi-jointure, ce paramètre est traditionnellement fixé entre 1 000 et 5 000 [HBL14 ; HB15]. Comme les distances ajoutent un surcoût significatif, nous préférons privilégier un meilleur équilibre dans la répartition des charges en le fixant à 1 000. La taille des esquisses est fixée à 320 bits, pour représenter environ 10 % des données transmises lorsque les enregistrements sont de tailles 100. De plus, le filtrage par esquisses n'est pas appliqué lorsque les enregistrements sont de taille inférieure à 30, puisque le calcul de la distance est plus rapide que le calcul des esquisses, le surcoût en termes de données transmises et leurs comparaisons.

| Paramètre | Notation | Valeur |
|---|---|---|
| Nombre de processeurs | P | 48 |
| $\log(p_1)/\log(p_2)$ | $\rho$ | 0.5 |
| Espérance de collision | $\mathbb{E}$ | 3 |
| Taille d'un bloc | $\mathbf{f}_{max}$ | 1 000 |
| Taille d'un *chunk* | $\mathbf{t}_{\max}$ | 1 000 000 |
| Taille des esquisses | $b$ | 320 bits |

| Seuil | Nombre d'itérations | Nombre de concaténations |
|---|---|---|
| $\lambda = 0.05$ | $\mathcal{Q} = 12$ | $\mathcal{K} = 26$ |
| $\lambda = 0.1$ | $\mathcal{Q} = 12$ | $\mathcal{K} = 13$ |
| $\lambda = 0.2$ | $\mathcal{Q} = 12$ | $\mathcal{K} = 6$ |
| $\lambda = 0.3$ | $\mathcal{Q} = 13$ | $\mathcal{K} = 4$ |
| $\lambda = 0.4$ | $\mathcal{Q} = 14$ | $\mathcal{K} = 3$ |

Tab. 4.3 : Les paramètres de l'algorithme *MRS-join*.

### a) L'évaluation des filtrages supplémentaires

Le tableau Tab. 4.4 présente l'évaluation des filtrages supplémentaires. Nous évaluons l'ajout d'un deuxième niveau de filtrage par LSH lorsque la valeur de l'attribut de jointure est fortement fréquente et que les enregistrements correspondants ont été redistribués

sur plusieurs tâches **reduce**. De plus, nous évaluons également le filtrage par esquisses. L'utilisation d'un deuxième niveau de filtrage en utilisant la stratégie de Aumüller et al. [AC22] implique d'augmenter le nombre d'itérations, ce qui augmente la quantité de données transmises et le temps d'exécution en comparaison à l'algorithme *MRS-join* sans filtrage. L'exhaustivité de l'algorithme est néanmoins améliorée et largement supérieure au **rappel** souhaité, c'est-à-dire $\delta = 0.95$.

Il est possible d'améliorer les performances de cette stratégie en faisant en sorte que le nombre d'itérations ne soit pas augmenté, et ce, tout en garantissant un certain **rappel**. En effet, il n'est pas nécessaire que les probabilités aux deux niveaux soient identiques (cf. section 4.2.3.c). Il est donc possible de configurer la stratégie de manière à ce que la probabilité au deuxième niveau soit proche de 0.99, ce qui théoriquement devrait améliorer les performances globales de l'algorithme. Nous ne présentons pas toutefois pas d'expériences démontrant l'efficacité de cette approche.

Le filtrage par esquisses permet d'améliorer les performances en termes de temps d'exécution et de **précision**. On peut remarquer que l'augmentation des données transmises dépasse les attentes. Cela est dû au fait que la compression semble être sous-optimale lorsque les esquisses sont envoyées. Pour la suite, nous désactivons les différents filtrages supplémentaires.

| Algorithme | | MRS | Filtrage LSH | Esquisses |
|---|---|---|---|---|
| Nombre d'entrées | $[\times 10^6]$ | 100 | 100 | 100 |
| Enregistrements transmis | | 298 818 291 | 378 453 592 | 298 818 291 |
| Nombre de résultats | $[\times 10^6]$ | 4 802 | 4 816 | 4 675 |
| Distances calculées | $[\times 10^6]$ | 28 225 | 22 949 | 8 886 |
| Temps d'exécution | [s] | 1600 | 2880 | 1416 |
| Données Transmises | [GB] | 107 | 141 | 137 |
| **rappel** | | 0.993 | 0.997 | 0.968 |
| **précision** du filtrage | | 0.17 | 0.21 | 0.53 |

Tab. 4.4 : Les performances des différents filtrages additionnels de l'étape de jointure de l'algorithme *MRS-join* en utilisant le grand jeu de données de 100 GB en fixant le seuil à $\lambda = 0.25$. Nous paramétrons le filtrage à deux niveaux de LSH avec l'espérance fixée à $\mathbb{E} = 4$. Le nombre d'itérations, de concaténations et l'espérance au deuxième niveau sont tels que $F_{\mathcal{Q}} = 32$, $F_{\mathcal{K}} = 8$, $F_{\mathbb{E}} = 3$. La quantité de données transmises présentée correspond à celle après compression.

### b) Les expériences sur des ensembles de données réels

Pour analyser la performance et la qualité des résultats générés par l'algorithme *MRS-join*, nous avons utilisé six ensembles de données provenant du monde réel et un ensemble de données synthétique en utilisant un générateur de la littérature. Les ensembles de données proviennent principalement des études de la littérature [MAB16 ; Fie18]. Nous avons ajouté un ensemble de données provenant du Web Table Corpus 2015 (WDC) [Leh16] ne contenant que les données en anglais. Les ensembles de données textuels sont pré-traités pour construire les ensembles à l'aide des outils de [MAB16]. Cette étape de prétraitement n'est pas mesurée dans nos expériences puisque nous nous concentrons sur l'étape de jointure par similarité. De plus, nous comparons *MRS-join* avec l'algorithme *VernicaJoin* (cf. section 3.2.3.a) en termes de temps d'exécution, de l'utilisation de réseau, et de qualité des résultats produits.

Le tableau Tab. 4.5 présente les caractéristiques des ensembles de données sélectionnés, indiquant le nombre d'enregistrements, la taille moyenne et maximale des ensembles, la taille de l'univers et la taille sur disque. Les enregistrements dans les ensembles de données AOL, LIVE, UNIFORM et WDC sont courts ($\leq 50$), ce qui favorise l'algorithme *VernicaJoin*.

La distribution des jetons suit une loi de Zipf [Zip49] pour la plupart des ensembles de données, ce qui signifie qu'une grande partie de l'univers est très peu fréquente. Les ensembles de données NETFLIX et UNIFORM sont des exceptions. L'univers de NETFLIX correspond à son catalogue de films et séries, et les moins regardés restent relativement fréquents. L'ensemble de données synthétique UNIFORM a été fabriqué en utilisant un générateur [MAB16] qui produit une distribution uniforme des jetons.

| Dataset | Nombre d'ensembles | Taille des ensembles | | Univers | Taille (B) |
|---|---|---|---|---|---|
| | $\cdot 10^5$ | max | avg | $\cdot 10^3$ | |
| AOL | 100 | 245 | 3 | 3900 | 396MB |
| ENRON | 2.5 | 3162 | 135 | 1100 | 254MB |
| LIVE | 31 | 300 | 36 | 7500 | 873MB |
| NETFLIX | 4.8 | 18000 | 210 | 18 | 576MB |
| ORKUT | 27 | 40000 | 120 | 8700 | 2.5GB |
| UNIFORM | 1 | 25 | 10 | 0.21 | 4.5MB |
| WDC | 414 | 17000 | 15 | 184644 | 5.8GB |

Tab. 4.5 : Les caractéristiques des ensembles de données.

Pour les expériences suivantes, les duplicatas dans les ensembles de données d'entrée ont été supprimés puisque le problème de l'équi-jointure est un problème différent de la

jointure par similarité. De plus, cela rend les résultats comparables aux études précédentes sur les ensembles [Fie18 ; MAB16].

Nous avons effectué une auto-jointure sur tous les ensembles de données précédents en variant le seuil de la distance de Jaccard pour tout $\lambda \in \{0.05, 0.1, 0.2, 0.2, 0.3\}$. Les performances en termes de temps de traitement sont présentées dans le tableau Tab. 4.6. Pour chaque exécution et pour des raisons pratiques, nous avons défini une limite d'une heure pour chaque *job*, ce qui représente plus de dix fois le temps d'exécution de *MRS-join* pour tous les ensembles de données. Le symbole "$\infty$" indique que cette limite de temps a été atteinte ou que l'exécution a échouée par manque de mémoire.

| Seuil ($\lambda$) | 0.05 | | 0.1 | | 0.2 | | 0.3 | | 0.4 | |
|---|---|---|---|---|---|---|---|---|---|---|
| Dataset | VJ | MRS | VJ | MRS | VJ | MRS | VJ | MRS | VJ | MRS |
| AOL | **131** | 245 | **131** | 158 | **136** | 143 | **151** | 159 | **171** | 220 |
| ENRON | **132** | 141 | 149 | **128** | 178 | **141** | 214 | **146** | 265 | **156** |
| LIVE | **155** | 171 | **163** | 176 | 185 | **179** | 203 | **171** | 307 | **197** |
| NETFLIX | 194 | **151** | 245 | **180** | 826 | **161** | 2450 | **177** | $\infty$ | **194** |
| ORKUT | 370 | **195** | 914 | **181** | 2362 | **175** | $\infty$ | **159** | $\infty$ | **186** |
| UNIFORM | **115** | 151 | **118** | 148 | **121** | 147 | **125** | 146 | **132** | 159 |
| WDC | $\infty$ | **490** | $\infty$ | **321** | $\infty$ | **323** | $\infty$ | **383** | $\infty$ | **705** |

Tab. 4.6 : La comparaison des temps d'exécution en secondes de l'algorithme *MRS-join* (MRS) et de l'algorithme *VernicaJoin* (VJ) en fonction du seuil donné par l'utilisateur pour la distance de Jaccard.

Les temps de traitement de l'algorithme *MRS-join* sont toujours inférieurs à ceux de l'algorithme *VernicaJoin*, à l'exception de l'ensemble de données AOL. Cependant, dans ce cas, les temps d'exécution sont du même ordre de grandeur. Dans les autres cas, *MRS-join* affiche un gain de vitesse de traitement pouvant dépasser un ordre de grandeur. En particulier, pour les ensembles de données NETFLIX, ORKUT et WDC, *VernicaJoin* échoue à calculer la jointure par similarité dans le temps imparti pour un ou plusieurs seuils.

Une analyse plus approfondie est présentée dans le tableau Tab. 4.7, qui compare les données transmises lors de la phase de communication de l'étape de jointure entre les algorithmes *MRS-join* et *VernicaJoin*. Il montre que *VernicaJoin* est inefficace en termes de données transmises pour les ensembles de données contenant en moyenne des enregistrements longs comme ENRON, NETFLIX et ORKUT. Cela est dû au fait que *VernicaJoin* utilise les préfixes (cf. section 3.2.3.a) pour calculer la jointure par similarité, ce qui le rend très sensible aux enregistrements longs et aux seuils élevés. Ce n'est pas le cas dans l'algorithme *MRS-join*, car il se repose sur LSH, qui est indépendant de la

dimensionnalité, pour identifier les enregistrements potentiellement similaires. De plus, nous rappelons que seules les données pertinentes sont transmises pour l'algorithme *MRS-join*, ce qui réduit considérablement les données transmises pendant la phase de communication de l'étape de jointure par similarité.

Dans le tableau Tab. 4.7, la quantité de données transmises pour l'ensemble de données WDC n'est pas présentée pour l'algorithme *VernicaJoin*, car il a échoué avant l'étape de jointure par similarité. En effet, pour calculer les préfixes en fonction de leurs fréquences, *VernicaJoin* construit l'histogramme de l'univers des jetons, qui est ensuite mis en mémoire. Pour les ensembles de données avec un très grand univers, comme WDC, *VernicaJoin* peut manquer de mémoire, ce qui limite son extensibilité. Cela ne peut pas se produire dans l'algorithme *MRS-join*, car l'histogramme des valeurs des attributs de jointure LSH est redistribué en fonction de leurs occurrences dans les *splits*. De plus, l'utilisation des *chunks* permet de garantir que la mémoire allouée soit toujours suffisante.

Enfin, *VernicaJoin* groupe les enregistrements en fonction de leurs jetons dans leur préfixe lors de l'étape de jointure. Le nombre d'enregistrements pour un jeton dépend de sa fréquence dans l'ensemble de données. Pour de très grands ensembles de données, un groupe peut ne pas tenir en mémoire d'une tâche **reduce**, ce qui limite l'extensibilité de l'algorithme. Même dans le cas de petits ensembles de données avec quelques jetons peu fréquents comme NETFLIX, cela limite significativement son efficacité, car les calculs de distances ne sont pas équitablement répartis sur l'ensemble des nœuds de traitement. Cela ne peut pas se produire dans l'algorithme *MRS-join*, puisque les fréquences de chaque valeur des attributs de jointure sont calculées, et que ces valeurs sont partitionnées en blocs et affectés à plusieurs tâches **reduce** distinctes et de manière aléatoire. Cela permet à l'algorithme *MRS-join* d'être extensible et insensible à la distribution des données tout en répartissant équitablement la charge sur l'ensemble des machines de la grappe.

| Seuil (λ) | 0.05 | | 0.1 | | 0.2 | | 0.3 | | 0.4 | |
|---|---|---|---|---|---|---|---|---|---|---|
| Dataset | VJ | MRS | VJ | MRS | VJ | MRS | VJ | MRS | VJ | MRS |
| AOL | 224MB | **95MB** | 273MB | **256MB** | **357MB** | 1006MB | **440MB** | 2GB | **490MB** | 5GB |
| ENRON | 2GB | **148MB** | 4GB | **237MB** | 8GB | **391MB** | 12GB | **661MB** | 17GB | **1GB** |
| LIVE | 2GB | **77MB** | 3GB | **147MB** | 7GB | **344MB** | 10GB | **804MB** | 13GB | **2GB** |
| NETFLIX | 10GB | **2MB** | 20GB | **8MB** | 40GB | **889MB** | 59GB | **2GB** | ∞ | **4GB** |
| ORKUT | 30GB | **21MB** | 59GB | **24MB** | 118GB | **41MB** | ∞ | **339MB** | ∞ | **2GB** |
| UNIFORM | 4MB | **110KB** | 5MB | **746KB** | **8MB** | 38MB | **10MB** | 129MB | **13MB** | 164MB |
| WDC | ∞ | **2GB** | ∞ | **4GB** | ∞ | **8GB** | ∞ | **15GB** | ∞ | **24GB** |

Tab. 4.7 : Les données transmises en octets durant l'étape de jointure de l'algorithme *MRS-join* (MRS) comparé à l'algorithme *VernicaJoin* (VJ) en fonction du seuil donné par l'utilisateur pour la distance de Jaccard.

Pour obtenir ces performances, l'algorithme *MRS-join* repose sur LSH et produit presque tous les résultats de la jointure par similarité. Le tableau Tab. 4.8 présente la qualité du filtrage résultant de l'application de LSH sur les ensembles de données précédents. La jointure par similarité complète est calculée en utilisant *VernicaJoin*, sauf pour WDC, où nous avons utilisé *MRS-join* avec l'espérance fixée à $\mathbb{E} = 3 * \ln(\|\text{WDC}\|)$ pour obtenir avec de très forte probabilité le résultat complet. Nous observons que *MRS-join* atteint au moins 90 % de **rappel** pour tous les ensembles de données. On peut remarquer les faibles valeurs de la **précision** dans les expériences. Ces valeurs ne sont pas mauvaises dans l'ensemble parce que la vérification de la distance de Jaccard est très performante. De plus, l'utilisation de filtrage par esquisses améliorerait ces valeurs pour les ensembles de données contenant des enregistrements de petites tailles, mais cela se ferait aux dépens des performances en termes de temps et de quantité de données transmises. De plus, il est important de rappeler qu'il n'existe pas de techniques de hachage ou de tri permettant de trouver des couples similaires, même pour ces valeurs de **précision**. Enfin, la comparaison du nombre de distances calculées présentée dans le tableau Tab. 4.9 montre que l'algorithme *MRS-join* les réduit considérablement par rapport à l'algorithme *VernicaJoin*, et ce même pour de très petits seuils où il est censé exceller.

| Seuil (λ) | 0.05 | | 0.1 | | 0.2 | | 0.3 | | 0.4 | |
|---|---|---|---|---|---|---|---|---|---|---|
| Dataset | rappel | précision | rappel | précision | rappel | précision | rappel | précision | rappel | précision |
| AOL | 0.96 | 0.002 | 0.96 | 0.004 | 0.95 | 0.007 | 0.97 | 0.002 | 0.94 | 0.002 |
| ENRON | 1.00 | 0.259 | 0.99 | 0.147 | 1.00 | 0.365 | 1.00 | 0.187 | 1.00 | 0.025 |
| LIVE | 0.99 | 0.340 | 0.99 | 0.060 | 0.99 | 0.009 | 0.98 | 0.004 | 0.96 | 0.003 |
| NETFLIX | 1.00 | 0.012 | 1.00 | 0.008 | 0.99 | 0.002 | 0.98 | 0.001 | 0.99 | 0.001 |
| ORKUT | 0.99 | 0.150 | 0.98 | 0.183 | 0.99 | 0.015 | 0.99 | 0.005 | 0.95 | 0.002 |
| UNIFORM | — | — | — | — | 0.94 | 0.001 | 0.97 | 0.001 | 0.95 | 0.001 |
| WDC | 1.00 | 0.071 | 1.00 | 0.024 | 1.00 | 0.012 | 0.95 | 0.034 | 0.95 | 0.261 |

Tab. 4.8 : La qualité du filtrage en termes de **rappel** et de **précision** de l'algorithme *MRS-join* en fonction du seuil donné par l'utilisateur pour la distance de Jaccard. Le **rappel** permet de mesurer l'exhaustivité du résultat alors que la **précision** mesure la réduction de l'espace de recherche par LSH. Les résultats pour UNIFORM ne sont pas indiqués pour les seuils 0.05 et 0.1, car il n'y a pas de couples similaires pour ces seuils.

| Seuil (λ) | 0.05 | | 0.1 | | 0.2 | | 0.3 | | 0.4 | |
|---|---|---|---|---|---|---|---|---|---|---|
| Dataset | VJ | MRS | VJ | MRS | VJ | MRS | VJ | MRS | VJ | MRS |
| AOL | 1E+08 | **2E+05** | 2E+08 | **2E+06** | 2E+09 | **7E+07** | 6E+09 | **1E+09** | 1.3E+10 | **1E+10** |
| ENRON | 2E+06 | **2E+05** | 1E+07 | **8E+05** | 1E+08 | **2E+06** | 6E+08 | **7E+06** | 2E+09 | **8E+07** |
| LIVE | 9E+06 | **3E+04** | 5E+07 | **4E+05** | 4E+08 | **2E+07** | 2E+09 | **2E+08** | 8E+09 | **2E+09** |
| NETFLIX | 5E+07 | **5E+02** | 5E+08 | **1E+04** | 5E+09 | **2E+06** | 2E+10 | **6E+07** | ∞ | **5E+08** |
| ORKUT | 2E+06 | **8E+02** | 1E+07 | **3E+03** | 2E+08 | **1E+05** | ∞ | **1E+06** | ∞ | **2E+07** |
| UNIFORM | 3E+07 | **3E+01** | 1E+08 | **2E+03** | 5E+08 | **8E+05** | 1E+09 | **2E+07** | 2E+09 | **1E+08** |
| WDC | ∞ | **2E+07** | ∞ | **2E+08** | ∞ | **3E+09** | ∞ | **5E+09** | ∞ | **2E+10** |

Tab. 4.9 : Le nombre de distances calculées par l'algorithme *MRS-join* (MRS) en comparaison à l'algorithme *VernicaJoin* (VJ) en fonction du seuil donné par l'utilisateur pour la distance de Jaccard.

### 4.3.6 L'extensibilité de l'algorithme *MRS-join*

Nous analysons maintenant l'extensibilité de l'algorithme *MRS-join* en utilisant le générateur de données. Les résultats sont présentés dans le tableau Tab. 4.10. Nous utilisons les deux ensembles de 10 GB et 100 GB. Les temps d'exécutions et les quantités de données transmises augmentent globalement d'un facteur 10 pour l'ensemble des seuils entre ces deux ensembles de données. Cela s'explique par le fait que l'algorithme *MRS-join* extrait les clusters de l'ensemble de données en ne transmettant que les données pertinentes, et qu'il ne dépend à aucun moment du produit Cartésien de l'ensemble des données d'entrée. En d'autres termes, les performances de l'algorithme *MRS-join* dépendent principalement de la taille des résultats de la jointure par similarité. Cela garantit l'extensibilité de l'algorithme *MRS-join* pour le traitement de la jointure par similarité.

Nous remarquons que les valeurs de la **précision** du filtrage sont globalement plus élevées que sur les ensembles de données provenant du monde réel. Cela s'explique par le fait que l'univers utilisé pour le générateur est plus grand que dans les ensembles de données du monde réel. Par conséquent, le nombre d'éléments partagés entre les modèles des différents clusters est très faible, ce qui implique que les ensembles au sein d'un cluster partagent en moyenne un nombre très faible d'éléments avec les ensembles en dehors de leurs clusters. Cette propriété nous permet également de déterminer le nombre de résultats sans recourir à un algorithme exhaustif. À titre d'exemple inverse, il résulte de l'ensemble de données NETFLIX, qui contient de longs enregistrements constitués à partir d'un univers de 18 000 jetons relativement fréquents, une faible précision. Nous avons remarqué dans ce cas que les performances s'améliorent en augmentant le nombre de concaténations, puisque l'augmentation théorique des coûts de communication est compensée par une forte réduction du nombre de comparaisons.

| Seuil ($\lambda$) | | 0.1 | | 0.25 | | 0.4 | | 0.5 | |
|---|---|---|---|---|---|---|---|---|---|
| Dataset | [GB] | 10 | 100 | 10 | 100 | 10 | 100 | 10 | 100 |
| Nombre d'entrées | [$\times 10^6$] | 9 | 99 | 9 | 99 | 9 | 99 | 9 | 99 |
| Enregistrements transmis | [$\times 10^6$] | 7.6 | 121 | 26.9 | 298 | 55 | 583 | 68 | 699 |
| Nombre de résultats | [$\times 10^6$] | 52 | 523 | 479 | 4 802 | 1 596 | 15 988 | 3 010 | 30 102 |
| Temps d'exécution | [s] | 177 | 802 | 282 | 1969 | 548 | 4799 | 862 | 7861 |
| Données transmises | [GB] | 2 | 49 | 7 | 107 | 15 | 195 | 18 | 223 |
| **rappel** | | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 |
| **précision** sans filtrage | | 0.14 | 0.14 | 0.16 | 0.17 | 0.20 | 0.20 | 0.22 | 0.22 |
| **précision** avec filtrage | | 0.50 | 0.51 | 0.49 | 0.53 | 0.53 | 0.53 | 0.52 | 0.51 |

Tab. 4.10 : L'extensibilité de l'auto-jointure de l'algorithme *MRS-join* en utilisant le générateur de données synthétiques en faisant varier la distance seuil. La **précision** est indiquée avec et sans le filtrage par esquisses des ensembles.

## 4.4 Conclusion

Dans ce chapitre, nous avons introduit, analysé et expérimenté l'algorithme *MRS-join*. Cet algorithme repose sur LSH pour réduire considérablement le nombre de comparaisons et sur l'utilisation de l'histogramme de la jointure pour déterminer les valeurs des attributs de jointure pouvant poser des problèmes. En résumé,

- La redistribution de l'histogramme permet de ne plus avoir besoin de stocker en mémoire l'ensemble de l'histogramme.
- L'analyse des coûts et les expérimentations sur l'algorithme *MRS-join*, utilisant des ensembles de données réels et synthétiques, démontrent que le surcoût lié à l'utilisation de l'histogramme distribué reste très faible par rapport aux gains de performances obtenus en réduisant la communication et le traitement des données uniquement aux données pertinentes, tout en évitant les différents **déséquilibres** tout au long des étapes de calcul de la jointure par similarité.
- Les performances de l'algorithme *MRS-join* montrent que l'utilisation de LSH permet de traiter efficacement la jointure par similarité tout en générant presque tous les résultats. Cela résout également les limitations que les approches exhaustives avaient sur de petits ensembles de données.
- L'algorithme *MRS-join* est extensible et permet de traiter des ensembles de données très volumineux, que ce soit pour les trajectoires ou pour les ensembles. Plus particulièrement, les performances de l'algorithme dépendent principalement de la taille du résultat.

# Chapitre 5

# *MRSF-join* : Un algorithme optimisé pour le traitement de la jointure par similarité

Dans ce chapitre, nous introduisons un algorithme, appelé “MapReduce Sequence Filtering” (*MRSF-join*), traitant la jointure par similarité de séquence avec la distance de Levenshtein. L’algorithme repose toujours à la fois sur des fonctions de hachage LSH pour regrouper les données potentiellement similaires et également sur l’utilisation de l’histogramme distribuée de la jointure pour éviter les effets des divers déséquilibres. Toutefois, une étape de filtrage additionnelle est ajoutée, permettant de réduire drastiquement les coûts de communication en n’envoyant les données qu’au dernier moment. Nous débutons par exposer les prérequis, suivis de la description de l’algorithme et d’une analyse des coûts. Enfin, nous concluons par les expériences menées sur différents ensembles de données. Ce travail a été soumis pour publication [Riv24].



## 5.1 Préliminaires

Nous présentons d’abord la construction des attributs de jointure pour la distance de Levenshtein et des esquisses utilisées pour réduire le nombre de distances calculées.

Nous nous concentrons sur cette distance, même si l'algorithme fonctionne sur d'autres types d'objets en changeant uniquement la famille de fonctions LSH.

### 5.1.1 La construction des attributs de jointure avec LSH pour la distance de Levenshtein

Nous rappelons qu'il n'y avait pas de famille de fonctions LSH pour la distance de Levenshtein utilisable pour le traitement de grands ensembles de données volumineux (cf. section 3.3.4). Cependant, la distance de Levenshtein peut être approximée en utilisant la distance Jaccard (cf. section 3.2.5.a) et nous avons constaté que MinHash est très efficace pour regrouper les ensembles similaires. Il est donc possible d'appliquer MinHash sur chaque multi-ensemble de $q$-grammes de chaque séquence en entrée.

Généralement, dans la littérature, la taille des $q$-grammes est assez grande ($q \geq 10$), car les algorithmes exhaustifs ne permettent pas de concaténer plusieurs $q$-grammes dans une valeur de l'attribut de jointure, ceci limite fortement l'efficacité de ces approches en raison de l'explosion de l'espace de recherche pour des ensembles de données volumineux. Fondamentalement, seul ce paramètre, $q > 0$, permet de réduire l'espace de recherche dans ces approches.

En utilisant MinHash (cf. section 3.3.2), le nombre de concaténations, $\mathcal{K} \geq 1$, joue également ce rôle ; nous avons donc deux paramètres. Il est alors possible choisir la taille des $q$-grammes de façon à ce que les seuils de la distance de Jaccard résultant des Théorème 2 et Corollaire 2.1 (cf. section 3.2.5.a) soient suffisamment élevés pour pouvoir calculer la jointure par similarité efficacement, tout en réduisant drastiquement l'espace de recherche. Par exemple, en fixant le seuil de la distance de Levenshtein normalisée à $\Lambda = 0.15$ et $q = 4$, il est possible de calculer une jointure par similarité sur des multi-ensembles en utilisant la distance de Jaccard, avec un seuil fixé à $\lambda = 0.75$ et en utilisant une seule concaténation par fonction de hachage LSH ($\mathcal{K} = 1$). Par conséquent, en ajoutant le nombre de concaténations, une valeur d'un attribut de jointure sera tirée à partir d'un domaine de taille $(\|\Sigma\|^q)^{\mathcal{K}}$ plutôt que simplement $\|\Sigma\|^q$ avec une approche exhaustive.

Nous rappelons que les distances se calculant en temps quadratique nécessitent un alignement propre pour chaque couple d'objets, comme c'est le cas pour la distance de Levenshtein. Pour réduire davantage l'espace de recherche, il faudrait donc que l'ordre des $\mathcal{K}$ $q$-grammes sélectionnés soient représentatifs de la séquence d'origine. Dans ce but, nous prenons une hypothèse forte sur la distribution des opérations de la distance de Levenshtein. Nous supposons que ces opérations sont uniformément réparties sur l'ensemble des deux séquences, ce qui nous permet de découper les séquences en $\mathcal{K}$ tranches et de partitionner les multi-ensembles de $q$-grammes en $\mathcal{K}$ parties en conséquence.

Nous supposons également que le nombre d'indels (insertions, délétions) est le même dans chaque partie. Bien que cette hypothèse semble très optimiste, le seuil de la distance Jaccard est défini dans le pire des cas, et le nombre moyen de $q$-grammes détruits, pour passer d'une séquence à l'autre, est en moyenne plus faible, ce qui garantit un certain degré d'exhaustivité.

**Définition 29.** Étant donné une séquence $u$ et supposons que le multi-ensemble lui correspondant soit de taille $\|\mathrm{G}^q(u)\| = \|u\| - q + 1 = \mathcal{K} * p$ pour un entier $p > 0$ désignant la longueur d'une tranche. Pour $j \in 1, \ldots, \mathcal{K}$, la $j$-ième tranche de la séquence, $u$, est définie par : $\mathrm{S}_j^{\mathcal{K}}(u) = u_{(j-1)*p+1} \cdots u_{j*p+q-1}$. Le multi-ensemble de $q$-grammes correspondant à la $j$-ième tranche est alors noté $\mathrm{G}_j^q(u) = \mathrm{G}^q(\mathrm{S}_j^{\mathcal{K}}(u))$ pour simplifier.

En pratique, lorsque la taille de la séquence n'est pas un multiple de $\mathcal{K}$, les $\|\mathrm{G}^q(u)\| \mod \mathcal{K}$ premières parties contiennent un $q$-gramme supplémentaire.

La procédure pour sélectionner un $q$-gramme parmi un multi-ensemble utilise MinHash, pour illustrer et définir formellement cette procédure, nous définissons les valeurs brutes des attributs de jointure, c'est-à-dire avant l'utilisation des fonctions de hachage universelles et de l'ajout du numéro de l'itération.

**Définition 30.** Soit $\mathcal{G}^*$ l'ensemble de tous les multi-ensembles de $q$-grammes. Supposons que nous disposions d'une fonction $\mathrm{MH} : \mathcal{G}^* \to \Sigma^q$ qui sélectionne un $q$-gramme à partir d'un multi-ensemble de $q$-grammes. Pour $i \in 1, \ldots, \mathcal{Q}$, la valeur brute du $i$-ième attribut de jointure est définie par :

$$x_i = \mathrm{MH}_1(\mathrm{G}_1^q(u)), \ldots, \mathrm{MH}_{\mathcal{K}}(\mathrm{G}_{\mathcal{K}}^q(u))$$

Par exemple, la figure Fig. 5.1 montre comment une valeur brute d'un attribut de jointure est construite en utilisant les $q$-grammes. Cette méthode de découpage permet donc de concaténer plusieurs $q$-grammes tout en préservant l'ordre de ces $q$-grammes dans la séquence originale, ce qui permet d'améliorer la qualité du filtrage et de réduire drastiquement les coûts de communication.

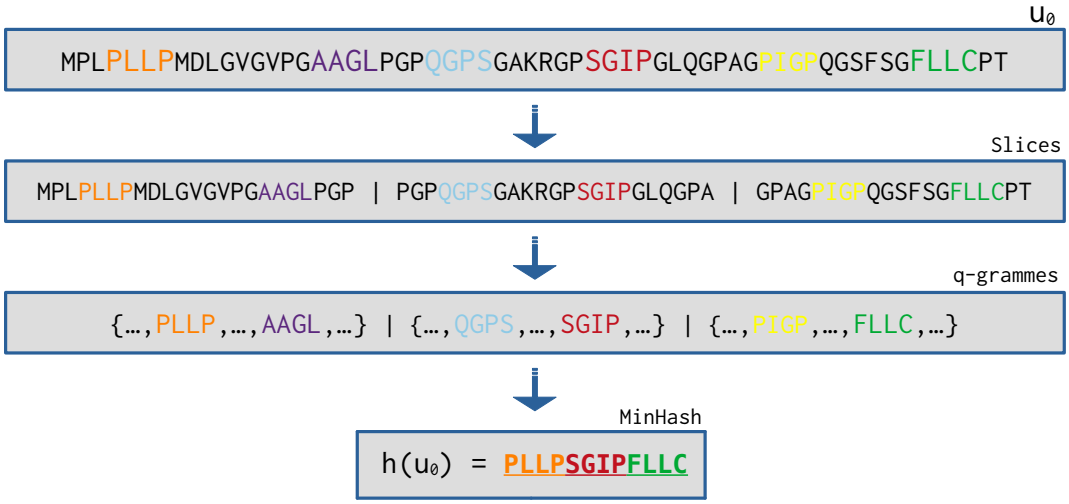



Fig 5.1 : Un exemple de valeur brute d'un attribut de jointure composée de trois 4-grammes concaténés provenant de trois tranches distinctes de la séquence originale.

Pour plus de précision sur la procédure de sélection des $q$-grammes, pour $j \in 1, \ldots, \mathcal{K}$, la fonction $\mathrm{MH}_j$ correspond à une seule fonction MinHash. Formellement, soit $\mathcal{H}^{\mathcal{K}}$ la famille LSH correspondant à une fonction de hachage obtenue en concaténant $\mathcal{K} \geq 1$ fonctions de hachage sélectionnées uniformément et indépendamment depuis $\mathcal{H}$, et appliquée à une tranche différente. Il tient sous l'hypothèse forte que l'égalité suivante est satisfaite :

$$\mathsf{P}_{g^{\mathcal{K}} \in \mathcal{H}^{\mathcal{K}}}[g^{\mathcal{K}}(u) = g^{\mathcal{K}}(v)] = \prod_{i=1}^{\mathcal{K}} \mathsf{P}[h_i(u) = h_i(v)], h_i \in \mathcal{H} = p_1^{\mathcal{K}}$$

Cela est dû au fait que les probabilités sont égales pour chaque tranche de la séquence sous cette hypothèse.

Pour distinguer les valeurs brutes des différents attributs de jointure, et pouvoir les utiliser rapidement, nous étendons la Définition 30 pour construire les valeurs des attributs de jointure en utilisant la même procédure que précédemment (cf. Définition 27). Cela signifie qu'une fonction de hachage universelle est utilisée sur les valeurs brutes et que le numéro d'itération est ajouté, permettant de représenter la valeur finale sur 64 bits.

Pour le moment, nous avons fait l'hypothèse que les $q$-grammes étaient uniques dans les multi-ensembles correspondant aux différentes tranches des séquences. Cependant, il suffit d'étendre les fonctions de MinHash de manière à retourner une position différente dans la permutation pour chaque occurrence d'un même $q$-gramme. Pour ce faire, il suffit de garder en mémoire le nombre d'occurrences de chaque $q$-gramme rencontré durant le processus de hachage et de retourner une valeur différente.

La taille de l'alphabet des séquences est un aspect important du problème. En effet, plus l'alphabet est grand, plus le filtrage est efficace, puisque chaque $q$-gramme sélectionné sur une tranche de la séquence sera tiré d'un domaine d'au moins $\|\Sigma\|^q$. Lorsque l'alphabet est de petite taille, disons 4 comme pour des séquences d'ADN, et que les séquences sont très longues, les multi-ensembles de $q$-grammes partageront un grand nombre de $q$-grammes, et ce, même si les séquences sont dissimilaires. Plus généralement, plus l'alphabet est grand, plus la distinction entre les séquences similaires et dissimilaires est facile à faire.

### 5.1.2 La construction des esquisses pour des séquences

Dans notre cas, les esquisses de séquences peuvent être considérées comme des esquisses des multi-ensembles de $q$-grammes des différentes tranches de la séquence d'origine. La taille de ces esquisses ne représente que quelques octets, elles peuvent donc être générées et comparées rapidement. Pour le traitement des séquences, cela permet de filtrer rapidement la plupart des couples de séquences dont la distance de Jaccard sur leurs multi-ensembles de $q$-grammes respectifs est supérieure au seuil $\lambda$. Nous étendons la définition des esquisses (cf. section 3.3.5) pour utiliser notre méthode de découpage en $\mathcal{K} \geq 1$ tranches. Notons $b$ la taille des esquisses, c'est-à-dire le nombre de bits d'une esquisse. L'idée est de représenter chacune des $\mathcal{K}$ tranches distinctement sur $b/\mathcal{K}$ bits, et à la suite dans l'esquisse d'une séquence.

**Définition 31.** Soit $\mathcal{H}$ une famille de fonctions LSH et soit $b = \mathcal{K} * l$ la longueur de l'esquisse pour un entier $l \geq 1$. Pour $i \in 1, \ldots, b$, une fonction de la famille LSH, $h_i : \mathcal{G}^* \to \Sigma^q$, est aléatoirement et indépendamment sélectionnée depuis $\mathcal{H}$. Pour une fonction de hachage universelle supplémentaire tel que $f : \Sigma^q \to \{0, 1\}$, l'esquisse d'une séquence $u \in \mathrm{R} \cup \mathrm{S}$ est définie comme $s(u) : \Gamma \to \{0, 1\}^b$, où chaque bit $i$ correspond à $s(u)_i = f(h_i(\mathrm{G}_k^q(u)))$ en prenant un entier $k$ satisfaisant $1 \leq k \leq \mathcal{K}$. Plus précisément, pour $i \in 1, \ldots, b$, la $i$-ième fonction de hachage est appliquée sur la $k$-ième tranche de la séquence d'origine, où $k = \lceil i/l \rceil$.

Cela signifie que les fonctions de hachage LSH sont appliquées sur les différents multi-ensembles de $q$-grammes correspondant aux différentes tranches de la séquence d'origine, comme représenté par la figure Fig. 5.2.

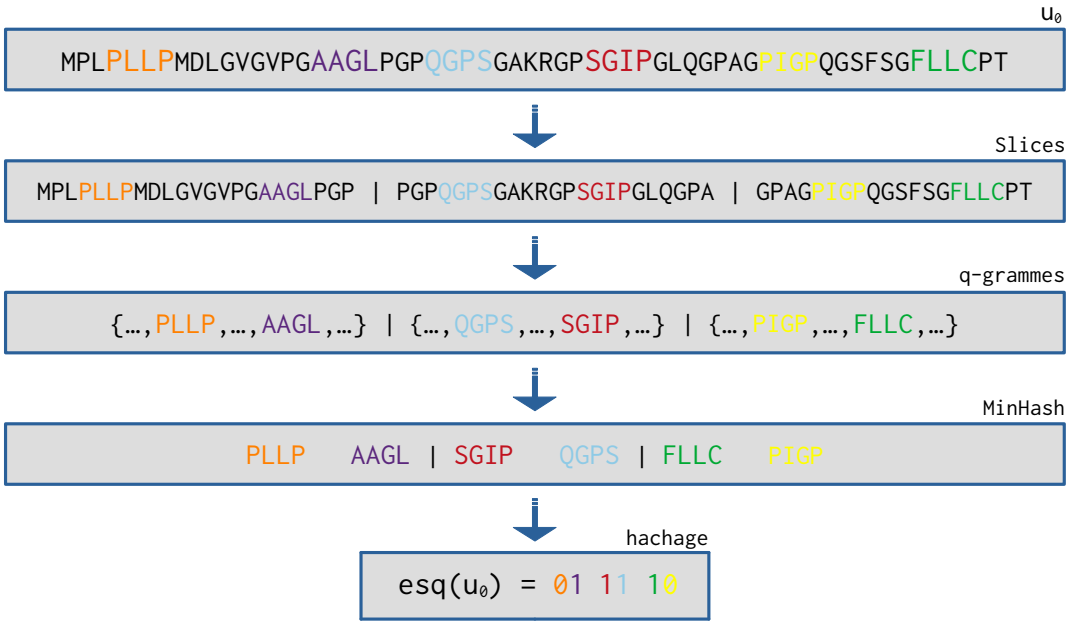



Fig 5.2 : Un exemple d'esquisse en utilisant trois tranches distinctes de la séquence originale.

Les probabilités de collisions sont les mêmes que précédemment pour les ensembles (cf. section 3.3.5), on élimine donc tous les couples de séquences dont la somme des collisions est strictement inférieure à $(2-\lambda)/2$. La somme des collisions est calculée globalement, et non localement sur la représentation de chaque tranche dans l'esquisse. Cela permet d'avoir un équilibre lorsque les opérations de la distance de Levenshtein ne sont pas distribuées uniformément sur la séquence.

### 5.1.3 Discussion sur la construction des fonctions LSH

Nous avons présenté le principe de découpage des séquences en $\mathcal{K}$ tranches pour utiliser MinHash sur des séquences et calculer la jointure par similarité en utilisant la distance de Levenshtein. Cela permet d'obtenir une famille de fonctions LSH ayant des probabilités constantes sous une hypothèse forte, et ce, peu importe la taille des séquences. Cette stratégie pourrait également être appliquée aux trajectoires pour approximer la distance de Fréchet dans le but d'avoir une famille de fonctions LSH avec des probabilités constantes sous hypothèse. L'idée serait alors de réutiliser la même représentation que la famille LSH (cf. section 3.3.3), c'est-à-dire l'utilisation d'une grille aléatoire, pour transformer une trajectoire en une séquence de nœuds de la grille. Cette séquence pourrait être découpée en tranches et un nœud de grille pourrait être sélectionné sur chaque tranche. Toutefois, nous ne fournissons ni preuve théorique ni résultats expérimentaux concernant l'utilisation de cette famille pour les trajectoires. Cette direction semble

seulement intéressante pour le traitement de très longues trajectoires.

## 5.2 *MRSF-join* : Les étapes de l'algorithme

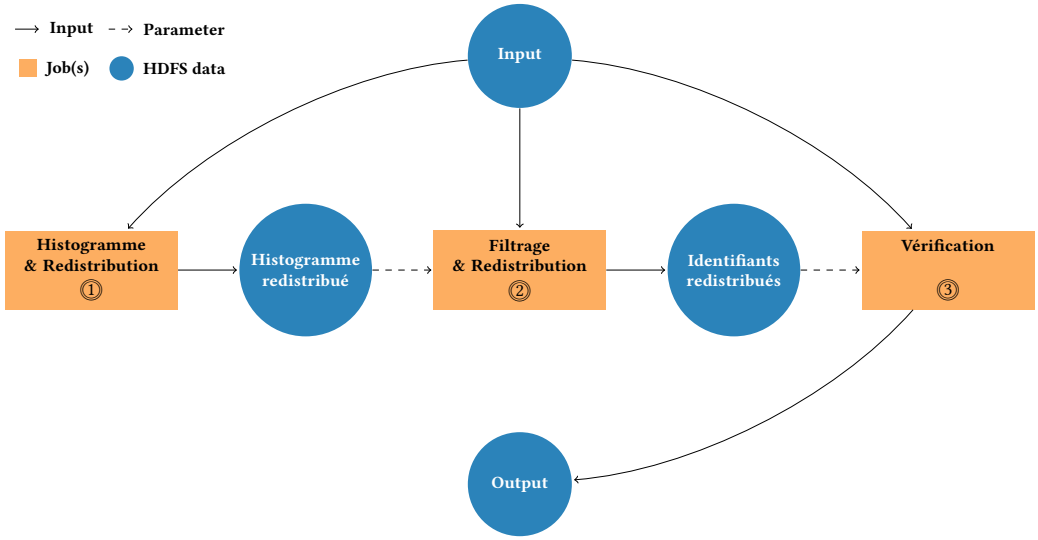



Fig 5.3 : *MRSF-join* : Étapes de traitement de la jointure par similarité.

L'algorithme *MRSF-join* se déroule en trois étapes, chacune comprenant un ou deux *jobs* MapReduce. La figure Fig. 5.3 montre les interactions entre les différentes étapes de l'algorithme :

①.a L'histogramme de la jointure est construit grâce aux valeurs des attributs de jointure, calculées en utilisant MinHash sur les $q$-grammes de chaque séquence découpée en $\mathcal{K}$ tranches,

①.b L'histogramme de la jointure est redistribué en fonction de l'occurrence des valeurs de l'attribut de jointure dans les différentes portions de l'entrée,

②.a En utilisant des histogrammes distribués, des schémas de communication randomisés, pour chaque valeur des attributs de jointure, sont générés pour éviter les effets des divers déséquilibres. Ensuite, la jointure est calculée et les couples d'identifiants de séquences sont filtrés en utilisant les esquisses, et finalement produits en sortie,

②.b Les couples d'identifiants de séquences sont redistribués en fonction de leur *split* d'origine,

③ En utilisant les couples d'identifiants de séquences redistribués, les séquences complètes sont transmises aux *Reducers*. Enfin, la distance de Levenshtein est calculée et les résultats de la jointure par similarité sont produits.

La principale différence avec l'algorithme *MRS-join* réside dans le fait qu'une étape de filtrage supplémentaire est introduite avant le calcul des distances et l'envoi des séquences. Ceci permet l'utilisation d'un nombre élevé d'itérations LSH tout en minimisant les coûts de communication en ne transmettant que des esquisses de séquences, et non les données des séquences. En résumé, l'étape ❷ de *MRS-join* est divisée en deux étapes, la première étant utilisée pour filtrer, générer et redistribuer les identifiants de couple de séquences ②, et la deuxième pour vérifier la distance de ces couples ③. Nous nous attendons à ce que l'exhaustivité de l'algorithme *MRSF-join*, mesuré par le **rappel**, soit légèrement plus faible par rapport à l'algorithme *MRS-join* en raison de cette étape de filtrage supplémentaire. Cependant, cette étape de filtrage devrait permettre d'améliorer la réduction de l'espace de recherche, mesuré par la **précision**.

Nous analysons les coûts associés aux différentes étapes de l'algorithme en utilisant les notations précédemment définies (cf. section 4.2). Nous ajoutons également les notations suivantes pour un ensemble de données $\mathrm{T} \in \{\mathrm{R}, \mathrm{S}\}$ :

| | |
|---|---|
| $\mathrm{Est}(\overline{\mathrm{T}}_i^{\mathrm{map}})$ | Information pour le filtrage provenant du *fragment* $\mathrm{T}_i^{\mathrm{map}}$, c'est-à-dire que chaque identifiant d'enregistrement qui apparaît via une valeur d'un attribut de jointure dans l'histogramme $\mathrm{Hist}(\mathrm{R} \bowtie_\sigma \mathrm{S})$ est associé à son esquisse, |
| $\mathrm{ID}(\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}})$ | Couple d'identifiants dont la distance doit être calculée, |
| $\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}}$ | Jointure par similarité évaluée par l'algorithme, c'est-à-dire l'ensemble des couples d'enregistrements dont la distance doit être calculée, |
| $\overline{\overline{\mathrm{R}}} \bowtie_\lambda \overline{\overline{\mathrm{S}}}$ | Les résultats de la jointure par similarité produite par l'algorithme *MRSF-join*. Cet ensemble est inclus à la fois dans $\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}}$ et dans $\mathrm{R} \bowtie_\lambda \mathrm{S}$. |

### 5.2.1 L'étape de calcul et de redistribution de l'histogramme de la jointure

L'étape de calcul de l'histogramme et de sa redistribution en fonction de l'occurrence des valeurs des attributs de jointure dans les différentes portions de l'entrée est la même que pour l'algorithme *MRS-join* (cf. aux sections. 4.2.1 et 4.2.2). Nous renvoyons donc le lecteur vers ces sections pour ce qui concerne les détails des implémentations, mais également pour l'analyse des coûts pour ces deux étapes.

### 5.2.2 L'étape de filtrage en utilisant les esquisses

L'étape de filtrage est détaillée dans l'Algorithme 5 et illustrée dans la figure Fig. 5.4. En utilisant les histogrammes distribués, la phase **map** envoie pour chaque enregistrement

et pour chaque valeur d'attribut de jointure restante un couple clé-valeur.

La clé est composée de la valeur de l'attribut de jointure, de l'identifiant de la tâche **reduce** et de l'identifiant de la ligne. Les deux dernières valeurs sont utilisées lorsque la valeur de l'attribut de jointure est très fréquente dans l'ensemble de données et que des schémas de communication spécifiques sont utilisés (cf. aux sections. 2.3.4.a et 4.1.3).

La valeur contient la localisation de l'enregistrement, c'est-à-dire l'identifiant du *split* et le décalage correspondant à la position de l'enregistrement dans le *split*, et des informations pour le filtrage. Généralement, il faut que ces informations pour le filtrage ne représentent que très peu d'espace, puisqu'ils peuvent être transmis au plus $\mathcal{Q}$ fois. Pour les séquences, ces informations contiennent la taille et une esquisse de la séquence (cf. aux sections. 3.3.5 et 5.1.2). La taille permet de filtrer rapidement et à moindre coût l'ensemble des couples de séquences qui ne peuvent pas être similaires, car leur ratio de tailles dépasse le seuil de la distance défini par l'utilisateur. Nous ne représentons pas dans les exemples l'envoi de la taille de séquence, puisque ce filtre n'est pas générique et n'est applicable que sur les séquences et les ensembles.

La phase **reduce** calcule la jointure et émet un couple clé-valeur, contenant les localisations des enregistrements pour chaque couple d'enregistrements qui n'a pas été filtré, et dont la distance entre les enregistrements d'origines doit être calculée. Il convient de souligner qu'un même couple d'identifiants peut être répété plusieurs fois dans les résultats, car il est trop coûteux d'envoyer tous les attributs de jointure pour chaque itération en raison du nombre important d'itérations. De plus, cette étape minimise la quantité de données transmises sur le réseau en réduisant drastiquement la taille de la jointure à évaluer sans transmettre les enregistrements.

### a) Analyse du coût de l'étape de filtrage dans *MRSF-join*

Par souci de simplicité, notons le fragment du *Mapper* $i$ par $\mathrm{T}_i^{\mathrm{map}} = \mathrm{R}_i^{\mathrm{map}} + \mathrm{S}_i^{\mathrm{map}}$. Le *Mapper* $i$ lit son *fragment* pour un coût correspondant au terme $\mathsf{c}_{\mathbf{r/w}} * (|\mathrm{R}_i^{\mathrm{map}}| + |\mathrm{S}_i^{\mathrm{map}}|)$. Pour chaque *split* de son *fragment*, identifié par un entier $j \in \mathcal{S}(\mathrm{T}_i^{\mathrm{map}})$, le *Mapper* $i$ lit l'histogramme distribué correspondant et le met en mémoire dans une table de hachage, ce qui correspond au terme $\mathsf{c}_{\mathbf{r/w}} * |\mathrm{Hist}_j^{\mathrm{split}}(\mathrm{R} \bowtie_\sigma \mathrm{S})| + \mathsf{c}_{\mathcal{H}} * \|\mathrm{Hist}_j^{\mathrm{split}}(\mathrm{R} \bowtie_\sigma \mathrm{S})\|$. Le terme $\mathcal{O}\left(\|\tilde{u}\| * \|\mathrm{T}_i^{\mathrm{map}}\|\right)$ représente le coût de calcul des valeurs des attributs de jointure et de l'esquisse pour chaque enregistrement dans le *fragment*, $\|\tilde{u}\|$ étant la longueur moyenne des enregistrements. Les données sont ensuite triées sur le *Mapper* $i$, ce qui est représenté par le terme $\mathcal{O}\left(\left(\mathcal{Q} * \|\overline{\mathrm{T}}_i^{\mathrm{map}}\|\right) * \log\left(\mathcal{Q} * \|\overline{\mathrm{T}}_i^{\mathrm{map}}\|\right)\right)$. Finalement, les données sont envoyées aux *Reducers*, ce qui est représenté par le terme $\mathcal{Q} * \mathsf{c}_{\mathbf{c}} * |\mathrm{Est}(\overline{\mathrm{T}}_i^{\mathrm{map}})|$. Par conséquent, le coût de cette étape est le suivant :

**Algorithme 5 :** Étape de filtrage (②.a)

**Map :** <id, u> → List(<($x_i$, $\text{reducer}_{\text{id}}$, $\text{row}_{\text{id}}$), (localisation, id, esquisse)>)

**Init :**

Lire depuis le HDFS les fonctions LSH.

Lire depuis le HDFS l'histogramme distribué $\text{Hist}_j^{\text{split}}(\text{R} \bowtie_\sigma \text{S})$ où $j$ correspond à l'identifiant du *split* courant.

Construire une table de hachage en utilisant les valeurs des attributs de jointure comme clé et les fréquences comme valeur.

localisation ← getLocalisation()

esquisse ← getEsquisse(u) (cf. section 5.1.2)

Pour tout $i \in 1, \dots, \mathcal{Q}$ :

Calculer la valeur $x_i$ correspondant au $i$-ième attribut de jointure de l'enregistrement $u$ en utilisant LSH.

Si la valeur $x_i$ est présente dans l'histogramme distribué :

Émettre des couples clé-valeur conformément aux modèles de communication décrits précédemment (cf. section 2.3.4.a) et en accord avec les fréquences de l'histogramme.

**Partition :** <($x_i$, $\text{reducer}_{\text{id}}$, $\text{row}_{\text{id}}$), (localisation, id, esquisse)> → Integer

Rediriger chaque couple en fonction de la valeur $x_i$ ou de la tâche de destination "reduceId", générée aléatoirement durant la phase de **map**, en suivant les différents scénarios des schémas de communication.

**Reduce :** <($x_i$, $\text{reducer}_{\text{id}}$, $\text{row}_{\text{id}}$), List(localisation, id, esquisse)> → List(<($\text{id}_\text{u}$, $\text{localisation}_\text{u}$), ($\text{id}_\text{v}$, $\text{localisation}_\text{v}$)>)

Calculer la jointure en utilisant les modèles de communication.

Pour chaque couple d'enregistrements ($u$, $v$) distincts provenant de R × S :

Si le couple passe l'ensemble des filtres :

Émettre un couple <($\text{id}_\text{u}$, $\text{localisation}_\text{u}$), ($\text{id}_\text{v}$, $\text{localisation}_\text{v}$)>.

$$
\begin{aligned}
\mathsf{Time}(\text{2.a.Mapper}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{M}} \sum_{j \in \mathcal{S}(\text{T}_i^{\text{map}})} \Big( \mathsf{c}_{\mathbf{r/w}} * |\text{Hist}_j^{\text{split}}(\text{R} \bowtie_\sigma \text{S})| + \mathsf{c}_{\mathcal{H}} * \|\text{Hist}_j^{\text{split}}(\text{R} \bowtie_\sigma \text{S})\| \Big) \\
+ \ \mathsf{c}_{\mathbf{r/w}} * (|\text{R}_i^{\text{map}}| + |\text{S}_i^{\text{map}}|) + \|\tilde{u}\| * \|\text{T}_i^{\text{map}}\| \\
+ \Big( \mathcal{Q} * \|\overline{\text{T}}_i^{\text{map}}\| \Big) \ * \ \log \Big( \mathcal{Q} * \|\overline{\text{T}}_i^{\text{map}}\| \Big) + \mathcal{Q} * \mathsf{c}_{\mathbf{c}} * |\text{Est}(\overline{\text{T}}_i^{\text{map}})| \Big).
\end{aligned}
$$

Le coût de calcul de la jointure et du filtrage des enregistrements dissimilaires en utilisant les esquisses est représenté par le terme $\mathcal{O}\left( \|\text{Est}(\overline{\text{R}}_i^{\text{red}}) \bowtie \text{Est}(\overline{\text{S}}_i^{\text{red}})\| \right)$, tandis que le coût pour l'écriture les résultats du *Reducer* $i$ correspond au terme $\mathsf{c}_{\mathbf{r/w}} * |\text{ID}(\overline{\overline{\text{R}}} \bowtie_\sigma \overline{\overline{\text{S}}})|$.

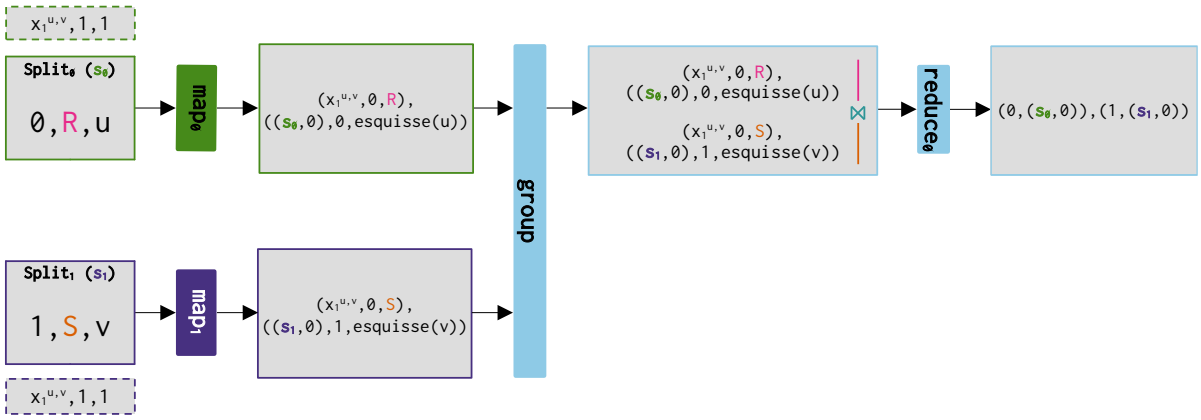



Fig 5.4 : Un exemple d'exécution de l'étape de filtrage : Les deux enregistrements d'entrée $u$ et $v$, identifiés respectivement par 0 et 1, et appartenant aux *splits* $s_0$ et $s_1$, émettent respectivement au plus $\mathcal{Q}$ couples clé-valeur correspondant à leurs $\mathcal{Q}$ attributs de jointure. Nous supposons que seule la valeur $x_1^{u,v}$ est présente dans l'histogramme, ce qui signifie que les enregistrements $u$ et $v$ ont la même valeur pour le premier attribut de jointure, comme pour les exemples de l'algorithme *MRS-join* (cf. section 4.2.2). Les schémas de communication spécifiques ne sont pas utilisés dans cet exemple puisque la valeur $x_1^{u,v}$ n'apparaît que deux fois dans l'entrée. La valeur du 'reducerId' des clés transmises est donc mise à 0 et la valeur du 'rowId' prend l'ensemble de données d'entrées pour trier les données intermédiaires. Nous représentons dans cet exemple la localisation par les tuples $(s_0, 0)$ et $(s_1, 0)$ pour les enregistrements $u$ et $v$. Ce qui signifie que l'enregistrement $u$ est dans le *split* $s_0$ à la position 0. Lors du calcul de la jointure, les esquisses des enregistrements sont comparées. Dans cet exemple, nous supposons que les esquisses de $u$ et $v$ sont similaires et que la distance nécessite d'être vérifiée.

Par conséquent,

$$\mathsf{Time}(\text{2.a.Reducer}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{R}} \|\mathrm{Est}(\overline{\mathrm{R}}_i^{\mathrm{red}}) \bowtie \mathrm{Est}(\overline{\mathrm{S}}_i^{\mathrm{red}})\| + \mathsf{c}_{\mathbf{r/w}} * |\mathrm{ID}(\overline{\overline{\mathrm{R}}} \bowtie_{\sigma} \overline{\overline{\mathrm{S}}})| \Big).$$

Ainsi, cette étape de filtrage aura le coût suivant :

$$\mathsf{Time}(\text{2.a.}) = \mathsf{Time}(\text{2.a.Mapper}) + \mathsf{Time}(\text{2.a.Reducer}).$$

### 5.2.3 L'étape de redistribution des sous-problèmes

Cette étape permet de redistribuer les couples d'identifiants potentiellement similaires vers leurs *splits* d'origine. Les doublons sont également supprimés. Les détails sont présentés dans l'Algorithme 6 et illustrés dans la figure Fig. 5.5. L'idée de l'algorithme est globalement la même que la redistribution de l'histogramme.

**Algorithme 6 :** Étape de redistribution des identifiants (②.b)

**Map :** <(id$_u$, localisation$_u$), (id$_v$, localisation$_v$)> → List(<localisation, id$_u$>)

Émettre un couple clé-valeur <localisation$_u$, id$_u$>.

Émettre un couple clé-valeur <localisation$_v$, id$_u$>.

**Partition :** <localisation, id$_u$> → split$_{\text{id}}$

Partitionner les clefs composites en fonction de l'identifiant du *split* uniquement, cet identifiant provenant de la valeur 'localisation'.

**Reduce :** <localisation, List(id$_u$)> → <localisation, Set(id$_u$)>

Calculer l'ensemble des valeurs reçues pour éliminer les doublons.

Émettre un couple clé-valeur <localisation, Set(id$_u$)>.

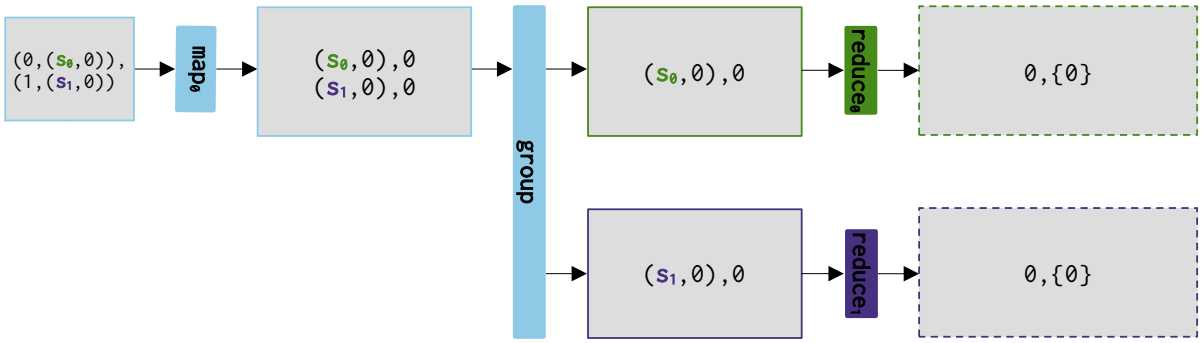



Fig 5.5 : Un exemple d'exécution de l'étape de redistribution des identifiants. L'enregistrement $u$ étant dans le *split* $s_0$ à la position 0. Il reste maintenant à transmettre l'enregistrement $u$ sur la clef 0. Pour l'enregistrement $v$ qui est dans le *split* $s_1$ à la position 0, il faudra également le transmettre sur la clef 0 pour pouvoir calculer la distance entre les deux enregistrements.

Pour des ensembles de données très volumineux, l'ensemble calculé durant la phase de **reduce** peut ne pas tenir en mémoire. Cependant, il est possible de modifier l'algorithme de sorte que l'identifiant envoyé dans la valeur soit inclus dans la clef. De cette manière, les doublons sont éliminés durant la phase de communication et l'ensemble des identifiants reçu peut être émis en plusieurs fois.

### a) Analyse du coût de l'étape de redistribution des sous-problèmes

Le terme $\mathbf{c_{r/w}} * |\mathrm{ID}_i^{\mathrm{map}}(\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}})| * \mathcal{Q}$ correspond au coût de lecture des couples d'identifiants probablement similaires par le *Mapper* $i$. Le terme $\mathcal{Q}$ est dû au fait que les couples d'identifiants potentiellement similaires peuvent être dupliquées. Les données sont ensuite triées. Le terme $\mathcal{O}\left(\left(\mathcal{Q} * \|\mathrm{ID}_i^{\mathrm{map}}(\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}})\|\right) * \log\left(\mathcal{Q} * \|\mathrm{ID}_i^{\mathrm{map}}(\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}})\|\right)\right)$

représente le coût de cette étape. Enfin, les données sont transmises aux *Reducers*. Le coût est représenté par le terme $\mathsf{c_c} * |\mathrm{ID}_i^{\mathrm{map}}(\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}})| * \mathcal{Q}$. En conséquence,

$$\mathsf{Time(2.b.Mapper)} = \mathcal{O}\Big( \max_{i=0}^{\mathcal{M}} \mathsf{c_{r/w}} * |\mathrm{ID}_i^{\mathrm{map}}(\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}})| * \mathcal{Q} + \mathsf{c_c} * |\mathrm{ID}_i^{\mathrm{map}}(\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}})| * \mathcal{Q} + \Big( \mathcal{Q} * \|\mathrm{ID}_i^{\mathrm{map}}(\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}})\| \Big) * \log\Big( \mathcal{Q} * \|\mathrm{ID}_i^{\mathrm{map}}(\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}})\| \Big) \Big).$$

De façon semblable à l'étape de redistribution de l'histogramme, le nombre de tâches **reduce** de ce *job* est paramétré pour être égal au nombre de *splits*. Lors de la phase **reduce**, une tâche correspond donc aux données nécessaires à un *split* identifié par un entier $j$ tel que $j \in \mathcal{S}(\mathrm{R}) \cup \mathcal{S}(\mathrm{S})$. Sur le *Reducer* $i$, son *fragment* correspond à un sous-ensemble distinct de tâches **reduce** $\mathcal{S}_i^{\mathrm{red}}(\mathrm{R} \cup \mathrm{S}) \subseteq \mathcal{S}(\mathrm{R}) \cup \mathcal{S}(\mathrm{S})$. Pour chacune de ces tâches, l'algorithme élimine les doublons. Le coût de cette étape correspond au terme $\mathcal{O}\Big( \mathcal{Q} * \|\mathrm{ID}_j^{\mathrm{split}}(\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}})\| \Big)$. Enfin, les résultats sont produits et stockés sur les disques, ce qui correspond au terme $\mathsf{c_{r/w}} * |\mathrm{ID}_j^{\mathrm{split}}(\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}})|$.

$$\mathsf{Time(2.b.Reducer)} = \mathcal{O}\Big( \max_{i=0}^{\mathcal{R}} \sum_{j \in \mathcal{S}_i^{\mathrm{red}}(\mathrm{R} \cup \mathrm{S})} \mathcal{Q} * \|\mathrm{ID}_j^{\mathrm{split}}(\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}})\| + \mathsf{c_{r/w}} * |\mathrm{ID}_j^{\mathrm{split}}(\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}})| \Big).$$

Par conséquent, cette étape de redistribution des couples d'identifiants aura le coût suivant :

$$\mathsf{Time(2.b.)} = \mathsf{Time(2.b.Mapper)} + \mathsf{Time(2.b.Reducer)}.$$

### 5.2.4 L'étape de vérification et de calcul des distances

Les détails de la dernière étape sont présentés dans l'Algorithme 7 et illustrés dans la figure Fig. 5.6. La phase **map** envoie un couple clé-valeur pour chaque identifiant redistribué. La phase **reduce** calcule les distances pour chaque couple d'enregistrements potentiellement similaires. Les résultats de la jointure par similarité sont finalement produits et stockés sur les disques de la grappe.

**Algorithme 7 :** Étape de vérification (③)

**Map :** <id, u> → List(<($s$, 0|1), (id, u)>)

Lire les identifiants d'enregistrements potentiellement similaires correspondant à l'enregistrement courant s'il y en a.

Pour chaque identifiant $id_v$ :

Si ($id_v$ == id) :

Émettre un couple clé-valeur <(id, 0), (id, u)>.

Sinon :

Émettre un couple clé-valeur <($id_v$, 1), (id, u)>.

**Partition :** <(id, 0|1), (id, u)> → id

Partitionner les clefs composites en fonction de l'identifiant 'id' uniquement.

**Reduce :** <($id_u$, 0), (id, u)>

Stocker en mémoire l'enregistrement (id, u).

**Reduce :** <($id_u$, 1), List(<$id_v$, v>)> → List(<($id_u$, u), ($id_v$, v)>)

Pour chaque enregistrement reçu :

Calculer la distance avec l'enregistrement stocké en mémoire.

Si $\mathcal{D}(u, v) \leq \lambda$ :

Émettre un couple clé-valeur <($id_u$, u), ($id_v$, v)>.

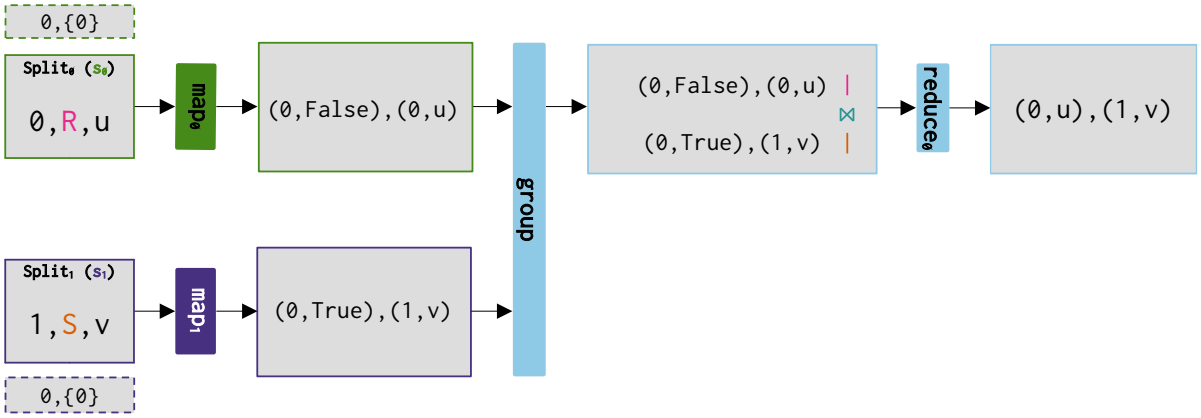



Fig 5.6 : Un exemple d'exécution de l'étape de vérification. En reprenant les identifiants redistribués de l'étape précédente, les deux enregistrements sont envoyés sur la même clef, et la distance est calculée et comparée au seuil fixé par l'utilisateur. Dans cet exemple, nous supposons que la distance calculée est inférieure au seuil, ce qui entraîne la génération d'un couple clé-valeur contenant les deux enregistrements.

### a) Analyse du coût de l'étape de vérification

Le coût de la phase **map** de l'étape de vérification est le suivant :

$$\mathsf{Time}(3.\mathsf{Mapper}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{M}} \mathsf{c}_{\mathbf{r/w}} * |\mathrm{T}_i^{\mathrm{map}}| + \sum_{j \in \mathcal{S}(\mathrm{T}_i^{\mathrm{map}})} \mathsf{c}_{\mathbf{r/w}} * |\mathrm{ID}_j^{\mathrm{split}}(\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}})| + \|\mathrm{ID}_j^{\mathrm{split}}(\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}})\| \Big).$$

Le terme $\mathsf{c}_{\mathbf{r/w}} * |\mathrm{T}_i^{\mathrm{map}}|$ représente le coût de lecture du *fragment* assigné au *Mapper* $i$. Pour chaque *split* $j$, les identifiants redistribués d'enregistrements similaires sont lus et un couple clé-valeur est émis pour chaque identifiant, ce qui correspond au terme $\mathsf{c}_{\mathbf{r/w}} * |\mathrm{ID}_j^{\mathrm{split}}(\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}})| + \|\mathrm{ID}_j^{\mathrm{split}}(\overline{\overline{\mathrm{R}}} \bowtie_\sigma \overline{\overline{\mathrm{S}}})\|$.

Le coût de la phase **reduce** de l'étape de vérification est le suivant :

$$\mathsf{Time}(3.\mathsf{Reducer}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{R}} \mathsf{c}_{\mathbf{c}} * |\overline{\overline{\mathrm{R}}}_i^{\mathrm{red}} \bowtie_\sigma \overline{\overline{\mathrm{S}}}_i^{\mathrm{red}}| + \left( \|\overline{\overline{\mathrm{R}}}_i^{\mathrm{red}} \bowtie_\sigma \overline{\overline{\mathrm{S}}}_i^{\mathrm{red}}\| \right) * \log\left( \|\overline{\overline{\mathrm{R}}}_i^{\mathrm{red}} \bowtie_\sigma \overline{\overline{\mathrm{S}}}_i^{\mathrm{red}}\| \right) + \mathsf{c}_{\mathcal{D}} * \|\overline{\overline{\mathrm{R}}}_i^{\mathrm{red}} \bowtie_\sigma \overline{\overline{\mathrm{S}}}_i^{\mathrm{red}}\| + \mathsf{c}_{\mathbf{r/w}} * |\overline{\overline{\mathrm{R}}}_i^{\mathrm{red}} \bowtie_\lambda \overline{\overline{\mathrm{S}}}_i^{\mathrm{red}}| \Big).$$

Le terme $\mathsf{c}_{\mathbf{c}} * |\overline{\overline{\mathrm{R}}}_i^{\mathrm{red}} \bowtie_\sigma \overline{\overline{\mathrm{S}}}_i^{\mathrm{red}}|$ correspond au coût de communication des données intermédiaires entre les *Mappers* et le *Reducer* $i$. Le coût engendré par le tri des données intermédiaires correspond au coût $\mathcal{O}\left( \left( \|\overline{\overline{\mathrm{R}}}_i^{\mathrm{red}} \bowtie_\sigma \overline{\overline{\mathrm{S}}}_i^{\mathrm{red}}\| \right) * \log\left( \|\overline{\overline{\mathrm{R}}}_i^{\mathrm{red}} \bowtie_\sigma \overline{\overline{\mathrm{S}}}_i^{\mathrm{red}}\| \right) \right)$. Les distances sont finalement calculées pour chaque couple d'enregistrements transmis, ce qui correspond au terme $\mathsf{c}_{\mathcal{D}} * \|\overline{\overline{\mathrm{R}}}_i^{\mathrm{red}} \bowtie_\sigma \overline{\overline{\mathrm{S}}}_i^{\mathrm{red}}\|$. Le terme $\mathsf{c}_{\mathbf{r/w}} * |\overline{\overline{\mathrm{R}}}_i^{\mathrm{red}} \bowtie_\lambda \overline{\overline{\mathrm{S}}}_i^{\mathrm{red}}|$ représente le coût pour stocker les résultats sur les disques.

En conséquence, le coût de cette étape de vérification est :

$$\mathsf{Time}(3.) = \mathsf{Time}(3.\mathsf{Mapper}) + \mathsf{Time}(3.\mathsf{Reducer}).$$

**b) Analyse du coût de l'algorithme *MRSF-join***

Le coût de l'algorithme *MRSF-join* est par conséquent la somme de toutes les étapes précédentes, ce qui peut être réécrit par :

$$
\begin{aligned}
\mathsf{Time}(\textit{MRSF-join}) = \mathcal{O}\Bigg( & \max_{i=0}^{\mathcal{M}} \Big( \mathsf{c_{r/w}} * (|\mathrm{T}_i^{\mathrm{map}}|) + \|\widetilde{u}\| * \|\mathrm{T}_i^{\mathrm{map}}\| + \mathsf{c_c} * |\mathrm{Hist}(\mathrm{T}_i^{\mathrm{map}})| \\
& \quad + (\mathcal{Q} * \|\mathrm{T}_i^{\mathrm{map}}\|) \ * \ \log\left(\mathcal{Q} * \|\mathrm{T}_i^{\mathrm{map}}\|\right) + \mathsf{c_c} * |\mathrm{Est}(\overline{\mathrm{T}}_i^{\mathrm{map}})| \Big), \\
& \max_{i=0}^{\mathcal{R}} \Big( \|\mathrm{Hist}_i^{\mathrm{red}}(\mathrm{R})\| + \|\mathrm{Hist}_i^{\mathrm{red}}(\mathrm{S})\| + \|\mathrm{Est}(\overline{\mathrm{R}}_i^{\mathrm{red}}) \bowtie \mathrm{Est}(\overline{\mathrm{S}}_i^{\mathrm{red}})\| \\
& \quad + \mathsf{c_c} * |\overline{\overline{\mathrm{R}}}_i^{\mathrm{red}} \ \bowtie_\sigma \ \overline{\overline{\mathrm{S}}}_i^{\mathrm{red}}| \\
& \quad + \left( \|\overline{\overline{\mathrm{R}}}_i^{\mathrm{red}} \ \bowtie_\sigma \ \overline{\overline{\mathrm{S}}}_i^{\mathrm{red}}\| \right) * \log\left( \|\overline{\overline{\mathrm{R}}}_i^{\mathrm{red}} \ \bowtie_\sigma \ \overline{\overline{\mathrm{S}}}_i^{\mathrm{red}}\| \right) \\
& \quad + \mathsf{c}_{\mathcal{D}} * \|\overline{\overline{\mathrm{R}}}_i^{\mathrm{red}} \ \bowtie_\sigma \ \overline{\overline{\mathrm{S}}}_i^{\mathrm{red}}\| + \mathsf{c_{r/w}} * |\overline{\overline{\mathrm{R}}}_i^{\mathrm{red}} \ \bowtie_\lambda \ \overline{\overline{\mathrm{S}}}_i^{\mathrm{red}}| \Big) \Bigg).
\end{aligned}
$$

Cela tient puisque l'histogramme de la jointure et sa redistribution est par définition lié à l'histogramme de chaque *split*. De plus, le calcul et la redistribution des couples d'identifiants est lié à la taille de la jointure par similarité évaluée par l'algorithme.

Lorsque les résultats de jointure par similarité sont nombreux, l'algorithme *MRSF-join* est dominé par les calculs de distances, ce qui correspond à l'étape de vérification de l'algorithme *MRSF-join*. Cependant, dans le cas contraire, le coût de l'algorithme *MRSF-join* est soit proportionnel à la taille de l'entrée, soit à la jointure évaluée lors du filtrage lorsque les résultats sont peu nombreux.

## 5.3 L'évaluation de l'algorithme *MRSF-join*

Dans cette section, nous discutons de l'efficacité et de la solidité de notre analyse théorique en expérimentant l'algorithme *MRSF-join* sur des ensembles de données réels et synthétiques. Nous avons mesuré l'efficacité de l'algorithme *MRSF-join* en termes de temps d'exécution et de coûts de communication de données. L'exhaustivité et la réduction de l'espace de recherche sont mesurés par le **rappel** et par la **précision** (cf. section 3.3.7). Nous évaluons ces deux mesures durant l'étape de vérification des distances, c'est-à-dire que les résultats de la jointure par similarité ne contiennent aucun couples de séquences avec une distance supérieure au seuil donné.

Les expériences ont été réalisées à l'aide du framework Hadoop 3.3.4 sur un cluster hétérogène de six machines. Quatre machines ont les spécifications suivantes : Intel(R) Xeon(R) CPU E5-2650 @2.60 GHz, 64 GB de mémoire et deux disques durs HDD d'une

capacité de 1 To chacun. Deux autres machines sont équipées d'un processeur Intel(R) Xeon(R) Gold 6248R @3.00GHz, de 256 GB de mémoire et de deux disques durs HDD de 8 To. Les nœuds de la grappe sont connectés par un réseau Ethernet de 1 Gbit/s. La compression en sortie de la phase **map** est activée à l'aide du codec Snappy, à l'exception des étapes de filtrage et de vérification qui utilisent le codec Gzip. La compression lors de l'écriture des résultats est réalisée avec le codec Gzip uniquement. La configuration de la mémoire pour les tâches **map** et **reduce** a été fixée à 2 GB et 6 GB respectivement.

### 5.3.1 Les paramètres de l'algorithme *MRSF-join*

L'algorithme *MRSF-join* peut calculer la jointure par similarité en utilisant plusieurs distances ; nous nous concentrons sur la distance de Levenshtein. Les expériences ont été réalisées en utilisant la variante normalisée de la distance Levenshtein, car les longueurs des séquences peuvent varier significativement dans les ensembles de données provenant du monde réels. De plus, nous expérimentons l'algorithme *MRSF-join* sur de longues séquences et en utilisant des valeurs de seuil élevées. Nous rappelons que les techniques de l'état de l'art ne permettent pas de traiter ces ensembles de données efficacement (cf. section 3.2.5). Plus précisément, l'ensemble des expériences ont été réalisées en fixant le seuil à 15 %. De plus, la taille de l'alphabet des séquences est fixée à 20, comme c'est le cas pour les protéines.

Jusqu'à présent, nous avons présenté l'algorithme traitant la jointure entre deux ensembles de données R et S. Dans nos expériences, afin de maximiser la taille des résultats, nous calculons une auto-jointure sur les différents ensembles de données. En comparaison aux expériences de l'algorithme *MRS-join*, la taille des *chunks* est réduite, car le nombre d'itérations est plus élevé. La taille des esquisses est également réduite, de sorte à représenter 10 % des données transmises lorsque les séquences sont de tailles 250.

| Paramètre | Notation | Valeur |
|---|---|---|
| Seuil de la distance normalisée de Levenshtein | $\Lambda$ | 15 % |
| Taille de l'alphabet | $\|\Sigma\|$ | 20 |
| Taille des $q$-grammes | $q$ | 4 |
| Nombre de concaténations | $\mathcal{K}$ | 3 |
| Nombre d'itérations | $\mathcal{Q}$ | 64 |
| Taille d'un bloc | $\mathbf{f}_{max}$ | 1 000 |
| Taille d'un *chunk* | $\mathbf{t}_{\max}$ | 50 000 |
| Taille des esquisses | $b$ | 192 bits |

Tab. 5.1 : Les paramètres de l'algorithme *MRSF-join*.

### 5.3.2 Le générateur de données synthétiques de séquences

Afin d'utiliser des ensembles de données de différentes tailles pour les expériences, nous avons utilisé un générateur de données synthétiques permettant de spécifier la taille moyenne des séquences, ainsi que le nombre de résultats de la jointure par similarité. Ce générateur est dans la même idée que le générateur de trajectoires (cf. section 4.3.1). Plus précisément, le générateur prend cinq paramètres, à savoir : la taille des séquences, le nombre de clusters, la taille des clusters, le seuil de la distance de Levenshtein normalisée et la taille de l'alphabet. Pour les expériences, les clusters générés ont une distance de Levenshtein normalisée comprise entre 10 % et 20 %. Les ensembles de données sont complétés par du bruit, c'est-à-dire un nombre de séquences aléatoires qui ne produisent avec de très forte probabilité aucun résultat.

### 5.3.3 L'évaluation du filtrage en variant la longueur des séquences

Pour analyser la qualité du filtrage de la famille de fonctions LSH pour de longues séquences, nous avons préparé plusieurs petits ensembles de données en faisant varier la longueur des séquences de 25 à 10 000. Pour chaque ensemble de données, nous avons généré 1 000 clusters de taille 10. Ces ensembles sont complétés par 1 000 séquences aléatoires.

Les résultats sont présentés dans le tableau Tab. 5.2. Nous observons que l'algorithme *MRSF-join* est très efficace et permet d'obtenir un **rappel** d'au moins 90 % pour des séquences de plus de 50 caractères. En revanche, pour les séquences plus courtes, les performances en termes d'exhaustivité sont relativement faibles et ne permettent pas de produire l'ensemble des résultats : ce biais peut s'expliquer par l'utilisation d'une seule fonction MinHash pour calculer l'ensemble des valeurs des attributs de jointure. De plus, bien que l'utilisation de trois concaténations fournisse un filtrage efficace, comme le reflètent les valeurs de **précision**, cela ne permet pas de traiter correctement les petites séquences. En effet, nous rappelons que l'algorithme *MRSF-join* ne transmet que des esquisses de séquences durant l'étape de filtrage, ce qui implique, pour les séquences très courtes, d'envoyer autant de données que les séquences d'origines. Nous suggérons donc de traiter ce cas différemment.

| Longueur | 25 | 50 | 75 | 100 | 250 | 500 | 750 | 1000 | 2500 | 5000 | 7500 | 10000 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Nombre d'entrées | 11000 | 11000 | 11000 | 11000 | 11000 | 11000 | 11000 | 11000 | 11000 | 11000 | 11000 | 11000 |
| Nombre de résultats | 19910 | 22475 | 22003 | 19126 | 20479 | 20890 | 20933 | 19979 | 20164 | 19680 | 20575 | 20853 |
| Distances calculées | 22924 | 30566 | 29146 | 29919 | 32275 | 33067 | 33397 | 32931 | 33235 | 33385 | 33640 | 32964 |
| **rappel** | 0.74 | 0.91 | 0.93 | 0.94 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 |
| **précision** du filtrage | 0.87 | 0.74 | 0.75 | 0.67 | 0.63 | 0.63 | 0.63 | 0.61 | 0.61 | 0.59 | 0.61 | 0.63 |

Tab. 5.2 : La qualité de filtrage de l'algorithme *MRSF-join* en termes de **rappel**, de **précision** et de distances calculées en faisant varier la longueur des séquences.

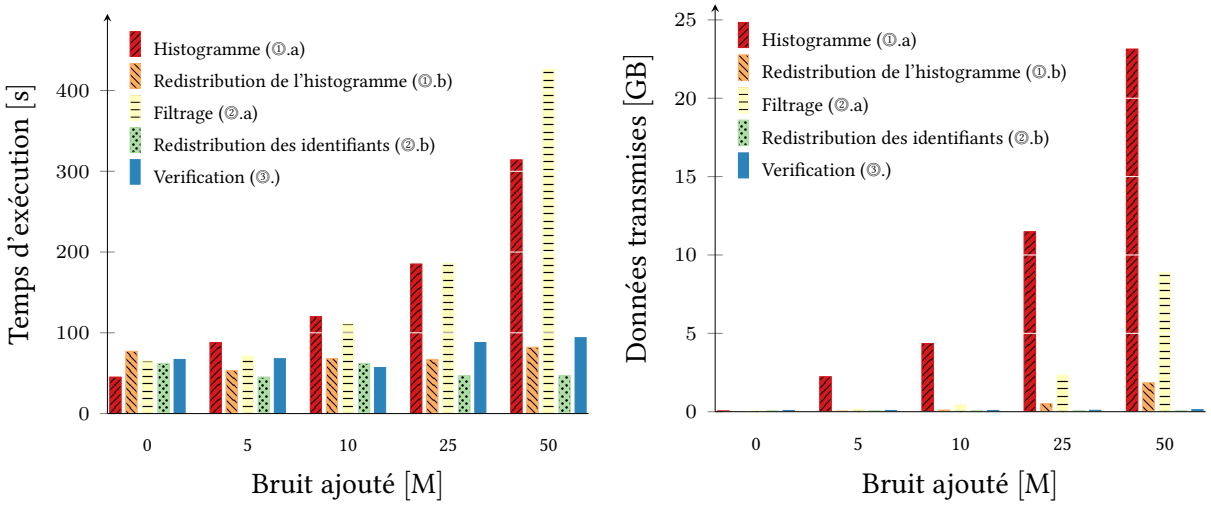
### 5.3.4 L'évaluation des performances sur des données synthétiques

Pour illustrer l'efficacité du filtrage sur de grands ensembles de données, nous avons préparé plusieurs ensembles de données en faisant varier uniquement la quantité de bruit. L'idée est de montrer que l'algorithme est capable d'extraire les clusters de séquences similaires provenant d'un ensemble de données très volumineux, et ce, en évitant le calcul d'un produit Cartésien. Dans ce but, tous les ensembles de données contiennent exactement les mêmes clusters de séquences. Il y a 1 000 clusters de taille 100. Toutes les séquences générées ont une longueur moyenne de 1 000, ce qui représente environ 1 GB sur disque par million de séquences. Afin de permettre une comparaison équitable, et de ne comparer que l'ajout de bruit, les graines aléatoires des fonctions de hachage LSH sont identiques entre les différentes exécutions.

La figure Fig. 5.7 présente les performances en termes de temps d'exécution et de coûts de communication des différentes étapes de l'algorithme *MRSF-join*. Les temps de traitement restent très faibles, et ce, même pour le plus grand ensemble de données contenant 50 millions de séquences aléatoires. Cela s'explique par le fait que, bien que les données d'entrée soient lues trois fois, seules les données pertinentes, c'est-à-dire celles pouvant produire un résultat, sont transmises durant les phases de communication.

Nous observons également que la quantité de données transmises lors de l'étape calculant l'histogramme est liée à la taille de l'ensemble de données. Pour l'étape de filtrage, les temps de traitement dépendent essentiellement du nombre de valeurs des attributs de jointure présentes dans l'histogramme. Étant donné que le nombre de

résultats de la jointure par similarité est très petit pour ces ensembles de données, la majeure partie du temps de traitement est consacrée à ces deux étapes. Enfin, les temps de traitement et les coûts de communication de l'étape de vérification restent constants. Cela s'explique par le fait que MinHash et l'étape de filtrage ont permis de réduire drastiquement le nombre de comparaisons, et de limiter au maximum les calculs de distances aux séquences faisant partie des clusters.




(a) Temps d'exécution de *MRSF-join*.

(b) Données transmises de *MRSF-join*.

Fig 5.7 : Les performances de l'algorithme *MRSF-join* en ajoutant du bruit, c'est-à-dire des séquences aléatoires, sur un ensemble de données synthétique.

Le tableau Tab. 5.3 présente une analyse plus détaillée de la qualité des résultats et du filtrage de l'algorithme *MRSF-join*. Nous observons que l'algorithme *MRSF-join* produit 99 % des résultats de la jointure par similarité, tout en réduisant drastiquement le nombre de distances calculées, comme le reflètent la **précision** du filtrage, qui atteint en moyenne 48 %. Cela signifie qu'environ la moitié des distances calculées sont pertinentes et figurent dans le résultat de la jointure par similarité. Pour donner une idée de l'efficacité de l'algorithme *MRSF-join*, comparons-le à un algorithme naïf de jointure par similarité. Ce dernier devrait effectuer une comparaison deux à deux de toutes les séquences de l'ensemble de données, ce qui représenterait jusqu'à $\sim 1.25 * 10^{15}$ distances calculées pour le plus grand ensemble de données, ce qui aurait un effet désastreux sur les performances. De plus, il convient de rappeler que la distance de Levenshtein est décidée en temps quadratique (cf. section 3.1.1.d).

| Bruit ajouté [M] | 0 | 5 | 10 | 25 | 50 |
|---|---|---|---|---|---|
| Nombre d'entrées | 100,000 | 5,100,000 | 10,100,000 | 25,100,000 | 50,100,000 |
| Esquisses filtrées | 976,236 | 2,284,439 | 6,112,586 | 32,697,631 | 127,353,919 |
| Nombre de résultats | 923,916 | 923,916 | 923,916 | 923,916 | 923,916 |
| Distances calculées | 1,915,057 | 1,915,567 | 1,916,983 | 1,926,660 | 1,960,962 |
| **rappel** | 0.995 | 0.995 | 0.995 | 0.995 | 0.995 |
| **précision** du filtrage | 0.48 | 0.48 | 0.48 | 0.48 | 0.47 |

Tab. 5.3 : La qualité de filtrage de l'algorithme *MRSF-join* en termes d'esquisses filtrées, de **rappel**, de **précision** et du nombre de distances calculées, en variant la quantité de bruit ajouté. Les esquisses filtrées font référence au nombre de paires de séquences éliminées pendant l'étape de filtrage. Il est important de noter que ce nombre comprend certains doublons, car la déduplication des couples d'identifiants similaires est effectuée durant l'étape suivante de redistribution.

### 5.3.5 Une comparaison avec l'algorithme *MRS-join*

Pour démontrer l'efficacité de l'algorithme *MRSF-join* par rapport à l'algorithme *MRS-join* sans filtrage supplémentaire, nous comparons le nombre de distances calculées, le temps de traitement et les coûts de communication en utilisant le plus grand ensemble de données synthétiques. Le tableau Tab. 5.4 présente une comparaison des performances. Tout d'abord, l'algorithme *MRSF-join* produit légèrement moins de résultats de jointure par similarité, ce qui est dû au filtrage par esquisse qui a été effectué. Cependant, ce filtrage permet de réduire le nombre de distances calculées d'un facteur 60 tout en minimisant les coûts de communication. Deuxièmement, le temps de traitement de l'algorithme *MRSF-join* est réduit d'un facteur 3, car les données transmises sont considérablement réduites, en considérant la somme des données transmises pour les étapes de l'algorithme *MRSF-join*. En bref, bien que *MRS-join* ne puisse pas échouer à traiter la jointure par similarité, l'algorithme *MRSF-join* offre de meilleures performances, car les couples de séquences dissimilaires sont filtrés avant d'être envoyés.

| Algorithme | | *MRS-join* | *MRSF-join* |
|---|---|---|---|
| Nombre d'entrées | | 50,100,000 | 50,100,000 |
| Nombre de résultats | | 927,113 | 923,916 |
| Distances calculées | | 129,030,002 | 1,964,159 |
| **rappel** | | 0.998 | 0.995 |
| **précision** du filtrage | | 0.007 | 0.472 |
| Temps d'exécution | [s] | 3,447 | 1,049 |
| Données transmises | [GB] | 242 | 10 + 0.2 + 2 ($\sim$ 12.2 GB) |

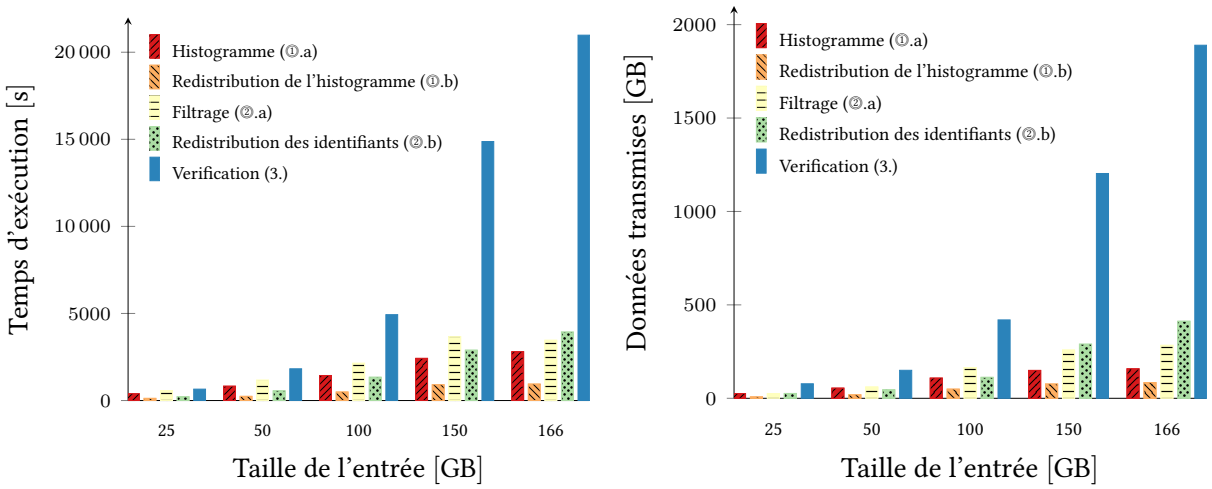
Tab. 5.4 : Une comparaison entre les algorithmes *MRS-join* et *MRSF-join* en termes de nombre de distances calculées, de temps de traitement et des coûts de communication. L'étape de calcul de l'histogramme et sa redistribution (❶, ①) étant la même pour les deux algorithmes, seule l'étape de jointure pour l'algorithme *MRS-join* (❷) et les étapes de filtrage, de redistribution (②) et de vérification (③) pour l'algorithme *MRSF-join* sont comparées pour les données transmises.

### 5.3.6 Les expériences sur un ensemble de données de protéines

Pour démontrer la capacité de l'algorithme *MRSF-join* à traiter la jointure par similarité sur des ensembles de données très volumineux, nous l'avons expérimenté sur le jeu de données metaclust [SS18]. L'ensemble de données original est composé de 1,59 milliard de fragments de séquences protéiques prédits par Prodigal [Hya10] provenant de divers ensembles de données métagénomiques et métatranscriptomiques. Pour les expériences suivantes, les séquences dupliquées ont été supprimées, car l'équi-jointure est un problème différent. De plus, comme l'objectif était de traiter la jointure par similarité sur de longues séquences, nous avons écarté les séquences de moins de 100 caractères. L'ensemble de données résultant est composé de 680 millions de séquences d'une longueur moyenne de 220, ce qui représente 166 GB de données sur disque. Nous découpons cet ensemble de données en tranche de 25 GB, de manière à tester l'extensibilité de l'algorithme.

La figure Fig. 5.8 présente les performances de l'algorithme *MRSF-join* sur cet ensemble de données. Nous remarquons que la majeure partie du temps de traitement est consacrée à l'étape de vérification, en particulier pour les jeux de données les plus volumineux, contrairement aux ensembles synthétiques précédents. Cela est dû au fait que les résultats de la jointure par similarité sont considérablement plus nombreux, comme le révèle le tableau Tab. 5.5, ce qui conduit à transmettre de grands volumes de données. Toutefois, les valeurs de la **précision** sont également élevées, y compris pour les ensembles de données les plus volumineux, ce qui signifie qu'une grande partie des données transmises correspond aux résultats. Par conséquent, il ne faut pas s'attendre à

une réduction significative des coûts de communication si les séquences d'origine sont envoyées. Finalement, l'algorithme *MRSF-join* produit 99 % des résultats de la jointure par similarité, tout en réduisant drastiquement l'espace de recherche, ce qui permet de traiter des ensembles de données très volumineux.




(a) Temps d'exécution de *MRSF-join*.

(b) Données transmises de *MRSF-join*.

Fig 5.8 : Les performances de l'algorithme *MRSF-join* en variant la taille de l'ensemble de données METACLUST en entrée.

| DATASET | [GB] | 25 | 50 | 75 | 100 | 125 | 150 | 166 |
|---|---|---|---|---|---|---|---|---|
| NOMBRE D'ENTRÉES | $[\times 10^6]$ | 111 | 231 | 360 | 468 | 580 | 643 | 680 |
| NOMBRE DE RÉSULTATS | $[\times 10^6]$ | 227 | 347 | 558 | 789 | 1065 | 4365 | 7408 |
| DISTANCES CALCULÉES | $[\times 10^6]$ | 1112 | 1952 | 3306 | 4936 | 6770 | 13715 | 20633 |
| **RAPPEL*** | | 0.99* | 0.99* | 0.99* | 0.99* | 0.99* | 0.99* | 0.99* |
| **PRÉCISION** DU FILTRAGE | | 0.2 | 0.18 | 0.17 | 0.16 | 0.16 | 0.32 | 0.36 |

TAB. 5.5 : La qualité de filtrage de l'algorithme *MRSF-join* en variant la taille de l'entrée de l'ensemble de données METACLUST : le **rappel** a été mesuré sur des échantillons du jeu de données, contrairement aux autres mesures.

## 5.4 Conclusion

Dans ce chapitre, nous avons présenté une famille de fonctions LSH permettant traiter des ensembles de données très volumineux de longues séquences et pour des seuils importants de la distance de Levenshtein. Nous avons introduit et analysé avec un modèle de coût l'algorithme *MRSF-join*. Finalement, nous avons présenté des expériences sur des ensembles de données synthétiques et provenant du monde réel, validant notre analyse théorique et démontrant l'efficacité de notre approche. En résumé,

- L'analyse des coûts et les expérimentations sur l'algorithme *MRSF-join* montrent que le surcoût lié aux étapes de calcul de l'histogramme et de filtrage, ainsi que leurs redistributions, reste très faible en comparaison à l'étape de vérification. De plus, ces étapes permettent de réduire drastiquement le nombre de distances calculées et les coûts de communication, ce qui permet de traiter efficacement la jointure par similarité pour des ensembles de données très volumineux, tout en ayant d'excellentes valeurs de **rappel** et de **précision**.
- L'algorithme *MRSF-join* offre de meilleures performances que l'algorithme *MRS-join*, en réduisant le nombre de comparaisons avant d'envoyer les séquences d'origines.
- L'algorithme *MRSF-join* permet de traiter des ensembles de données volumineux contenant de longues séquences et avec de larges seuils, ce qui résout les limitations des approches existantes dans la littérature.

# Chapitre 6

# Solution générique et squelette algorithmique de la jointure par similarité

Dans ce chapitre, nous introduisons une solution générique pour le traitement de la jointure par similarité. Nous présentons d'abord notre implémentation reposant sur le framework Apache Hadoop et son modèle de programmation MapReduce dans un cadre de BigData, puis nous présentons une implémentation de cette solution dans un contexte de calcul haute performance (HPC). Nous évaluons enfin ces solutions en comparant leurs performances sur des ensembles de données de différentes tailles. Ce travail a conduit à la publication [Ton23].



## 6.1 Une solution générique dans un cadre de mégadonnées

Nous avons expérimenté nos algorithmes sur des ensembles, des trajectoires et des séquences en utilisant plusieurs distances, et pour différents seuils. Toutefois, nous voulons qu'elle puisse prendre en charge d'autres types d'objets et de distances. Pour répondre à ce besoin, nous avons développé une solution générique et extensible permettant

aux utilisateurs de traiter l'opération de jointure par similarité sans avoir à modifier l'implémentation de l'algorithme. Cette approche permet de masquer les détails de bas niveau aux utilisateurs non expérimentés, ce qui leur permet de se concentrer sur la requête à exécuter plutôt que sur les aspects techniques du traitement distribué, tels que la gestion des divers facteurs de déséquilibres, la gestion de la mémoire et les différentes étapes du traitement de la jointure par similarité. Cette approche permet également de pouvoir adapter la solution à son propre type de données en fonction des besoins spécifiques des utilisateurs et de leur domaine d'expertise.

Pour traiter la jointure par similarité de manière générique, trois composants sont nécessaires :

- Une spécification de l'entrée permettant d'extraire et de lire les enregistrements de l'entrée ;
- Une distance pour comparer deux enregistrements par rapport à un seuil fixé à l'avance par l'utilisateur ;
- Une famille de fonctions LSH pour partitionner les enregistrements en fonction de leur similarité. Une famille de fonctions LSH est liée à la distance utilisée. Elle peut par ailleurs être utilisée pour construire des esquisses des objets.

Généralement, la distance et le seuil associé ne peuvent être prédéfinis pour un type de données : il dépend uniquement des données d'entrées. L'utilisateur, étant le mieux informé sur ses propres données, est donc le plus apte, de par son expérience, de déterminer la distance et le seuil appropriés. Cependant, une distance et un seuil ne sont qu'une approximation de ce que l'on attend comme résultats : ils peuvent donc être sujets à interprétation. Ils doivent être adaptés au besoin spécifique de l'application, en fonction des caractéristiques des données à traiter et des objectifs de la requête.

Dans la littérature, il n'existe pas toujours de famille de fonctions LSH efficaces avec des probabilités constantes pour toutes les distances, ce qui rend ce composant complexe à implémenter dans la pratique. Cependant, il n'y a pas de limite théorique à leur existence, du moins sous certaines hypothèses.

Notre solution générique, implémentée sur Apache Hadoop, utilise une interface Java pour ces trois composants. L'utilisateur peut facilement adapter notre solution générique à ses besoins spécifiques en écrivant une classe Java qui implémente l'interface correspondant à l'un de ces composants. La solution prend également en entrée un fichier XML, permettant de spécifier l'ensemble des paramètres de l'algorithme. Les composants peuvent être spécifiés dans ce fichier de configuration XML en fournissant les chemins d'accès vers les classes Java souhaitées. Il est aussi possible de réutiliser les classes de

composants de notre bibliothèque déjà développées pour les trajectoires, les ensembles et les séquences. Une illustration de la solution est présentée dans la figure Fig. 6.1.

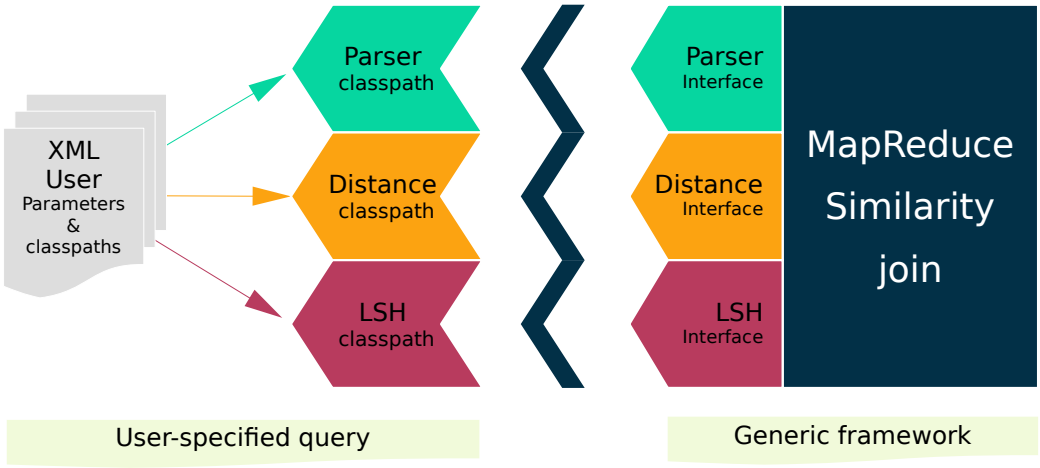



Fig 6.1 : La solution générique au traitement de la jointure par similarité : L'utilisateur fournit dans un fichier XML, les paramètres et les chemins vers les classes des composants, permettant la lecture, la comparaison et le partitionnement des enregistrements de son ensemble de données.

D'autres paramètres peuvent par ailleurs être spécifiés dans ce fichier XML tels que le seuil de la distance, les probabilités de la famille de fonctions LSH ou encore le **rappel** souhaité. Si les probabilités ne sont pas spécifiées, le nombre d'itérations et le nombre de concaténations doivent être fournis par l'utilisateur. De plus, ce fichier permet de remplacer les valeurs par défaut telles que la taille des blocs lors de l'utilisation des schémas de communication spécifiques ou la taille des *chunks* (cf. aux tableaux Tab. 4.3 et Tab. 5.1). Un exemple de ce fichier XML est présenté dans en Annexe A.1. L'utilisateur peut également étendre facilement les paramètres existants pour ses propres composants. Finalement, notre solution offre une grande flexibilité à l'utilisateur, qui peut choisir d'exécuter l'algorithme *MRS-join* ou l'algorithme *MRSF-join* pour le traitement de la jointure par similarité. Une interface graphique permettant de faciliter le paramétrage et le suivi de l'exécution de la jointure par similarité fera l'objet d'un développement futur.

## 6.2 Une solution générique dans un contexte de Calcul Haute Performance (HPC)

Les modèles de programmation parallèles tels que MapReduce ont simplifié le travail des scientifiques et des industriels dans de nombreux domaines d'application en réduisant les temps de développement pour obtenir des solutions extensibles à de très grands

ensembles de données sur des architectures distribuées. Généralement, ces modèles masquent les mécanismes de bas niveau et les complexités induites par la programmation parallèle et distribuée, comme la communication entre les nœuds, la tolérance aux pannes et la gestion des ressources disponibles sur la grappe, ce qui permet de se concentrer sur la logique algorithmique plutôt que sur les détails de bas niveau lié à la parallélisation. Les solutions implémentées sur ces modèles peuvent être exécutées sur n'importe quelle infrastructure matérielle, ce qui les rend portables, mais également plus faciles à déployer. Néanmoins, ils ne permettent pas toujours d'atteindre les meilleures performances sur l'architecture cible.

Lorsque l'on traite des ensembles de données de petites tailles pouvant tenir en mémoire, et que le temps de traitement est critique, l'utilisation de Hadoop et de son modèle MapReduce n'est pas forcément la solution la plus adaptée. En effet, Hadoop est conçu pour être tolérant aux pannes, ce qui signifie qu'il réplique les données sur plusieurs machines de la grappe pour garantir la disponibilité des données en cas de défaillance d'un nœud, ce qui induit un surcoût. De plus, le seul moyen d'utiliser une phase de communication au sein du modèle MapReduce est de relancer un *job*, c'est-à-dire une phase de **map** et de **reduce**. Nos algorithmes *MRS-join* et *MRSF-join*, qui sont respectivement composés de trois et cinq étapes, nécessitent de lire l'ensemble de données d'entrée deux et trois fois, ce qui induit là aussi un surcoût. Les autres étapes étant nécessaires pour la redistribution de l'histogramme ou des couples d'identifiants probablement similaires. Dans un contexte de calcul haute performance, ces surcoûts peuvent être évités pour obtenir les meilleures performances possibles en utilisant un autre modèle de programmation parallèle.

### 6.2.1 Une implémentation générique reposant sur *FastFlow*

Dans ce but, nous avons proposé, en collaboration avec l'Université de Pise (Italie), un squelette algorithmique de haut niveau traitant la jointure par similarité et reposant sur la bibliothèque *FastFlow* [Tor19 ; Ald17]. Cette bibliothèque permet de construire, à haut niveau, une topologie de communication en utilisant des blocs de flux de données. Ces blocs, appelés **Building Blocks**, sont des composants réutilisables et modulaires qui peuvent être combinés et imbriqués pour créer des applications de traitement de données complexes. Cette approche permet de simplifier la conception et le développement d'applications de traitement de données, en fournissant des composants de base prêts à l'emploi qui peuvent être facilement assemblés pour créer des flux de données personnalisés. Cette bibliothèque est développée en C++, ce qui permet également d'exploiter au maximum les performances des machines de la grappe. La bibliothèque *FastFlow* permet par ailleurs, avec peu de changements dans le squelette algorithmique,

de cibler à la fois des systèmes multicœurs et distribués pour le traitement de la jointure par similarité.

Le squelette algorithmique est similaire à notre solution générique, ce qui signifie que l'utilisateur doit fournir des fonctions pour extraire, comparer et partitionner les enregistrements. Le squelette repose sur un seul **Building Block**, appelé *All-to-All*, permettant de définir deux groupes de machines : gauche et droite. Les machines du groupe gauche peuvent communiquer avec toutes les machines du groupe droit. En termes de traitement, les machines du groupe gauche peuvent être considérées comme des *Mappers*, tandis que celles du groupe droit comme les *Reducers*. Des canaux de retour d'informations peuvent également être mis en œuvre afin que les *Reducers* puissent communiquer directement avec les *Mappers*. Cependant, cela suppose que les *Mappers* restent actifs pendant toute la durée du traitement et conservent en mémoire l'ensemble des données d'entrée. Cette contrainte impose une limite sur la taille des ensembles de données pouvant être traités en mémoire. Pour lever cette limite, *FastFlow* permet l'utilisation de mécanisme exploitant de la mémoire persistante (disques ou NVRAM) pour gérer les ensembles de données ne tenant pas en mémoire.

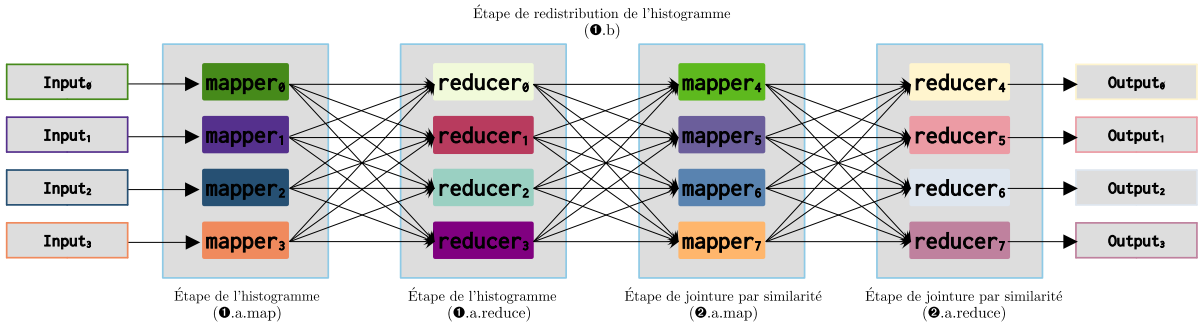



Fig 6.2 : La solution générique du traitement de la jointure par similarité reposant sur *FastFlow*. Les entrées sont lues et mises en mémoire au lancement des phases **map** des deux étapes.

L'implémentation reprend les étapes de l'algorithme *MRS-join*, comme illustrée par la figure Fig. 6.2. Le pipeline comporte deux blocs *All-to-All*. Le premier pour calculer l'histogramme de la jointure et le second pour traiter la jointure par similarité. Les phases **map** et **reduce** de ces deux étapes (cf. aux sections. 4.2.1 et 4.2.3) sont identiques à l'implémentation Hadoop. Les canaux d'informations connectant ces deux blocs permettent de redistribuer l'histogramme en fonction de l'occurrence des valeurs des attributs de jointure dans les différentes portions de l'entrée. La topologie du flux de données est déterminée au lancement de l'application, ce qui signifie que les *Mappers* et *Reducers* sont assignés à une machine de la grappe au début de l'exécution, ce qui diffère de Hadoop, où les tâches sont assignées dynamiquement en fonction de la disponibilité

des ressources. Bien que cette approche ne permette pas d'être tolérant aux pannes, elle offre de meilleures performances en évitant les surcoûts liés au stockage et à la relecture des données à chaque étape. Cependant, les *Mappers* et les *Reducers* de l'étape de calcul de l'histogramme restent actifs jusqu'à la fin du traitement de la jointure par similarité, ce qui utilise des ressources de la grappe inutilement. De plus, *FastFlow* ne permet pas de grouper horizontalement les traitements des *Mappers* et des *Reducers* de deux blocs *All-to-All* sur la même machine, ce qui signifie que les entrées doivent être lues et mises en mémoire deux fois.

La figure Fig. 6.3 présente une version plus optimisée du traitement de la jointure par similarité en utilisant un seul bloc *All-to-All* et des canaux de retours d'informations. Les mêmes *Mappers* et les *Reducers* traitent à la fois l'étape du calcul de l'histogramme et l'étape de jointure par similarité, ce qui améliore l'utilisation des ressources de la grappe. L'étape ❶.b (cf. section 4.2.2) est quant à elle implémentée en utilisant un canal de retour d'informations en multicast, permettant de redistribuer l'histogramme nécessaire à chaque *Mapper*.

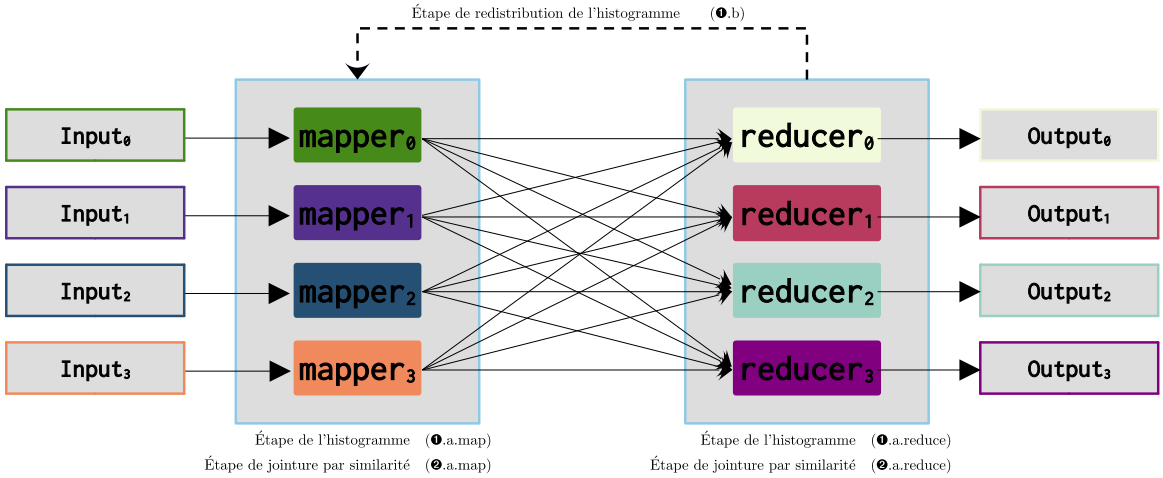



Fig 6.3 : La solution générique et optimisée du traitement de la jointure par similarité reposant sur *FastFlow*.

### 6.2.2 Discussion sur la mise en œuvre d'une version en streaming

La bibliothèque *FastFlow* permet également de gérer un flux continu de données et sans gestion de micro-lots. Nous considérons l'auto-jointure par similarité en flux continu, ce qui signifie que l'ensemble des données en entrée sont susceptibles d'être jointes entre elles. Lorsque le flux de données est important et que la répartition des données est fortement déséquilibrée, l'utilisation des schémas de communication est cruciale pour

répartir la charge de calcul sur l'ensemble des machines de la grappe. Cependant, dans ce contexte, l'ensemble de l'information n'est pas disponible dès le départ, ce qui peut poser des problèmes lors de la construction de l'histogramme et des schémas de communication à utiliser pour redistribuer les enregistrements sur les machines de la grappe. Nous ne présentons pas l'implémentation d'une solution, mais seulement des observations permettant une meilleure compréhension du problème.

La première implémentation, présentée dans la figure Fig. 6.2, semble plus simple à adapter. Tout d'abord, il est nécessaire de conserver l'ensemble des valeurs des attributs de jointure, même si elles ne peuvent pas conduire à un résultat immédiat, afin d'être en mesure d'incrémenter les fréquences lorsque les valeurs correspondant aux attributs de jointure de nouveaux enregistrements sont calculées. Il est également nécessaire de mettre en place des mécanismes de redistribution dynamiques des enregistrements correspondant à une valeur d'un attribut de jointure lorsque la fréquence dépasse un certain seuil, afin de répartir équitablement la charge sur l'ensemble des machines. Enfin, il est important d'attendre les mises à jour de l'histogramme correspondant aux valeurs des attributs de jointure d'un nouvel enregistrement et leurs possibles redistributions avant de calculer la jointure.

L'implémentation optimisée, présentée par la figure Fig. 6.3, semble plus complexe à adapter, car les *Mappers* peuvent recevoir au même moment l'entrée et une partie de l'histogramme distribué nécessaire au traitement de la partie de son entrée. Si l'on exécute en priorité la phase de l'histogramme sur les données en entrées, il est possible que la jointure ne soit jamais calculée, car l'accès en lecture doit être protégé des mises à jour de l'histogramme. Cela peut poser un problème de famine si le flux de données est important. À l'inverse, si l'étape de jointure est calculée en priorité, le résultat complet pour un nouvel enregistrement peut être très long à obtenir. Ces problèmes peuvent être évités en utilisant une solution de traitement en flux continu avec gestion de micro-lots avec état, telle qu'Apache Flink [Fli].

### 6.2.3 L'évaluation de l'implémentation *FastFlow*

Les expériences ont été réalisées à l'aide du framework Hadoop 3.3.4 sur un cluster de deux machines. Chaque machine est équipée d'un processeur Intel(R) Xeon(R) Gold 6248R @3.00GHz comprenant 24 cœurs physiques (48 threads), de 256 GB de mémoire et d'un disque NVMe dédié pour le stockage local. Les nœuds de la grappe sont connectés par un réseau Ethernet de 1 Gbit/s. La compression en sortie des phases **map** et **reduce** est désactivée afin de garantir une comparaison équitable, et de ne comparer que l'implémentation du même algorithme sur deux modèles de programmation différents. La configuration de la mémoire pour les tâches **map** et **reduce** a été fixée à 2 GB et 4

GB respectivement. La version basée sur *FastFlow* est configurée avec 24 *Mappers* et 24 *Reducers*, sans limitation de mémoire. Les mécanismes de gestion de mémoire persistante sont désactivées, ce qui signifie que tout le traitement de la jointure par similarité est effectué en mémoire principale.

| Dataset size | *Hadoop version* Temps [s] | | *FastFlow version* Amélioration | |
|---|---|---|---|---|
| | 1 | 2 | 1 | 2 |
| 5 GB | 133 | 140 | 7.8X | 5.6X |
| 50 GB | 507 | 537 | 1.8X | 2.4X |
| 100 GB | 1040 | 775 | — | 1.7X |

Tab. 6.1 : Tableau des temps d'exécution en secondes de l'implémentation Hadoop en utilisant un et deux nœuds, ainsi que le facteur d'amélioration de l'implémentation reposant sur *FastFlow* sur ces mêmes nœuds pour le traitement d'une jointure par similarité de trajectoires en utilisant l'algorithme *MRS-join*. L'ensemble de données de 100 GB ne peut pas être exécuté sur une seule machine par la version basée sur *FastFlow* en raison d'un manque de mémoire.

Pour l'expérience, nous avons utilisé les ensembles de données de trajectoires générés durant la section 4.3.4.b. Les résultats de l'expérience sont présentés dans le tableau Tab. 6.1. Nous observons que la version reposant sur *FastFlow* permet d'améliorer les performances sur les petits ensembles de données. En revanche, pour de grands ensembles de données, Hadoop peut s'exécuter sur un seul nœud avec une mémoire contrainte (192 GB) et obtenir de bonnes performances. De plus, il est difficile de dimensionner correctement le nombre de *Reducers* pour la solution reposant sur *FastFlow*, car la taille de la jointure par similarité n'est pas connue à l'avance, mais uniquement lors du calcul de l'histogramme. La solution peut donc échouer par manque de mémoire pour des ensembles de données de taille moyenne.

Enfin, une version de la solution a été développée dans l'environnement Apache Spark afin de comparer les performances de cette implémentation avec un autre environnement traitant la jointure par similarité en mémoire d'une grappe de calcul. Les temps d'exécution de cette version sont similaires à ceux de Hadoop pour le petit ensemble de données de 5 GB. Des expériences plus étendues sur des ensembles de données plus volumineux seraient également intéressante pour les développements futurs.

## 6.3 Conclusion

Dans ce chapitre, nous avons présenté notre solution générique reposant sur deux modèles de programmation parallèles. En résumé,

- Dans les deux cas, l'utilisateur de la solution doit fournir la spécification de l'entrée, une distance et une famille de fonctions LSH, ce qui peut présenter des difficultés à un utilisateur novice,
- Notre solution générique repose sur Hadoop et permet le traitement d'ensembles de données très volumineux. Toutefois, il y a des surcoûts liés à la tolérance aux pannes qui peuvent être évités en se reposant sur d'autres environnements de programmation parallèles pour des ensembles de données de petites tailles,
- L'implémentation reposant sur *FastFlow* évite ces surcoûts et permet des gains de performances sur les petits ensembles de données. Cependant, si un jeu de données ou les résultats de la jointure ne tiennent pas en mémoire, l'exécution échoue. Elle échoue également si une panne survient sur une machine de la grappe.

# Chapitre 7

# La recherche de similarité

Dans ce chapitre, nous introduisons une solution pour traiter la recherche par similarité en utilisant des fonctions de hachage LSH. Nous définissons le problème et nous présentons notre implémentation reposant sur Apache HBase. Nous évaluons finalement les performances de notre solution dans un contexte de mégadonnées.



## 7.1 La recherche de similarité avec LSH

L'opération de recherche par similarité consiste à retrouver, dans un ensemble de données, l'ensemble des enregistrements similaires à un enregistrement fourni par l'utilisateur. L'algorithme prend donc en entrée un enregistrement, une distance et un seuil et produit en sortie l'ensemble des enregistrements similaires.

**Définition 32.** Soit un ensemble de données R. Étant donné un enregistrement $u$, une distance $\mathcal{D}$ et un seuil $\lambda$ fournis par l'utilisateur, la recherche par similarité consiste à retrouver tous les enregistrements qui sont similaires à l'enregistrement $u$. Il s'agit donc de calculer l'ensemble suivant :

$$\{v \in \mathrm{R} \mid \mathcal{D}(u, v) \leq \lambda\}$$

La recherche par similarité est un cas particulier de la jointure par similarité où l'ensemble de données, S, ne contient qu'un seul enregistrement. Naïvement, il est possible de comparer l'enregistrement $u$ à l'ensemble des enregistrements de R. Cependant, cet algorithme aura des performances désastreuses dans un contexte de mégadonnées. Il faut donc drastiquement réduire l'espace de recherche et ne comparer que les enregistrements potentiellement similaires. De la même façon que pour la jointure par similarité, les familles de fonctions LSH peuvent être utilisées pour réduire l'espace de recherche tout en produisant une large majorité des résultats. Cela permet de ne comparer l'enregistrement de l'utilisateur qu'aux enregistrements ayant au moins une valeur commune pour un attribut de jointure, ce qui permet de réduire drastiquement le nombre de distances calculées et d'améliorer les performances de l'opération sur des mégadonnées.

Il est également important que les résultats de la recherche par similarité soient rendus disponibles à l'utilisateur rapidement. Il faut donc pouvoir préalablement prétraiter les données en indexant les enregistrements et les valeurs des attributs de jointure correspondantes, et ce, afin de retrouver rapidement les enregistrements potentiellement similaires. Dans ce but, nous avons utilisé des tables HBase pour stocker et indexer les données.

## 7.2 Apache HBase

Apache HBase [HBa] est un framework de gestion de données non relationnelles orientées colonnes, permettant l'accès aléatoire en lecture/écriture sur des ensembles de données très volumineuses. Ce projet open-source a été développé après la mise en œuvre de BigTable par Google [Cha06]. Une illustration du fonctionnement d'Apache HBase est présentée dans la figure Fig. 7.1. Apache HBase repose sur une architecture maître-esclave pouvant être très facilement étendue à de très larges échelles, et pouvant supporter des accès strictement cohérents et en temps réels à des peta-octets de données sur de larges grappes de serveurs. En cas de défaillance d'un serveur, le système gère automatiquement la répartition des charges entre les nœuds restants de la grappe, ce qui permet de maintenir la disponibilité et la fiabilité des données. Apache HBase s'intègre facilement avec Apache Hadoop et son modèle MapReduce, ce qui permet de préparer les données servies aux clients de manière efficace.

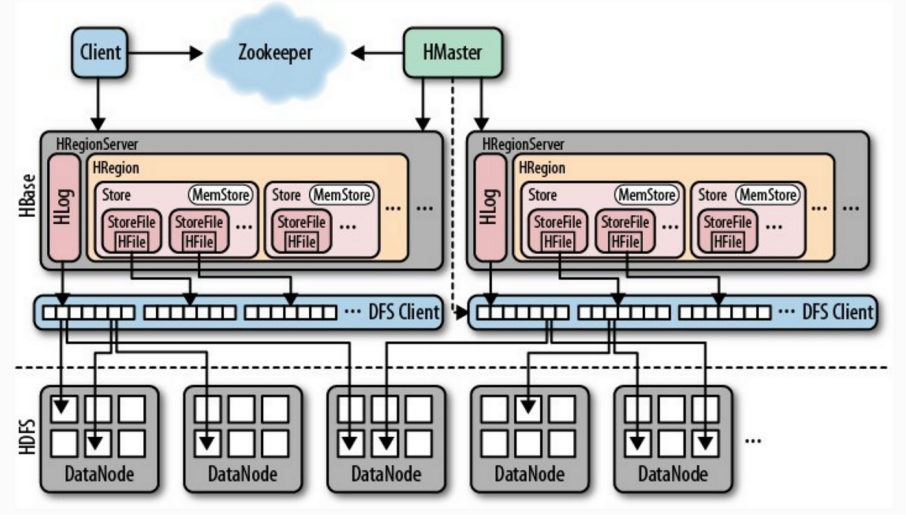



Fig 7.1 : Le fonctionnement d'Apache HBase. Le client peut ajouter, supprimer ou incrémenter une ligne dans l'entrepôt de données. Les données sont réparties sur plusieurs nœuds de la grappe, appelés HRegionServer. Chaque serveur est chargé de gérer une partie des données et de répondre aux requêtes des clients en lecture/écriture sur leurs régions spécifiques. Les données sont également stockées sur le HDFS pour des raisons de tolérances aux pannes. (source : bigdatariding.blogspot.com/2013/12/hbase-architecture.html)

## 7.3 L'algorithme de recherche par similarité

Fondamentalement, l'algorithme de recherche par similarité correspond aux étapes de filtrage et de vérification de l'algorithme *MRSF-join* (cf. Fig. 5.3). En effet, contrairement à l'algorithme de jointure, l'ensemble des valeurs des attributs de jointure des enregistrements doivent être stockés dans des tables HBase, et ne peuvent être pré-filtrés avec un histogramme, puisque l'enregistrement de la requête fourni par l'utilisateur ne fait pas nécessairement partie des données en entrées.

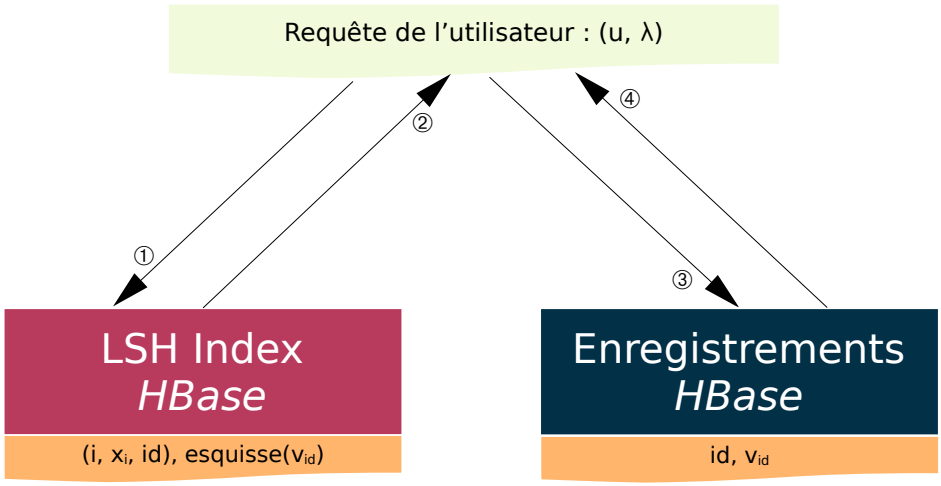



Fig 7.2 : Le fonctionnement de la recherche par similarité repose sur le framework Apache HBase. Les index LSH correspondent à l'association entre le numéro d'itération d'un attribut de jointure, la valeur pour cet attribut, un identifiant de séquence et l'esquisse de l'enregistrement.

L'algorithme est illustré dans la figure Fig. 7.2 et se décompose en quatre étapes, qui sont les suivantes :

①. Dans un premier temps, les fonctions de hachage LSH sont appliquées sur l'enregistrement de l'utilisateur pour récupérer les valeurs des attributs de jointure. Ensuite, un scan est initialisé sur l'ensemble de ces valeurs pour parcourir itérativement les index stockés sur le framework Apache HBase. Plus précisément, le scan utilise les préfixes $(i, x_i)$ pour filtrer, ce qui permet d'éviter de parcourir l'intégralité de la table d'index et assure ainsi de bonnes performances ;

②. Apache HBase renvoie itérativement les lignes correspondantes contenant les identifiants des enregistrements, ainsi que leurs esquisses. Ces identifiants sont filtrés grâce aux esquisses, ce qui permet de réduire drastiquement le nombre d'enregistrements à récupérer ;

③. Un nouveau scan est enfin initialisé pour récupérer les séquences correspondantes sans parcourir l'intégralité de la table des enregistrements. Ce scan permet également de récupérer l'ensemble des séquences similaires, sans devoir les stocker en mémoire du client au même moment ;

④. Les séquences potentiellement similaires sont finalement transmises à l'utilisateur et la distance est calculée et comparée par rapport au seuil $\lambda$.

Il faut remarquer que les index LSH peuvent être déterminés pour une plage de seuil $\lambda$. En effet, en fixant le nombre de concaténations des fonctions de hachage LSH lors

de la construction de l'index, il suffit de modifier le nombre d'itérations lors de l'étape ① en fonction du seuil choisi. Cependant, lorsque la plage des seuils est importante, comme c'est le cas pour les ensembles où le nombre de concaténations varie entre $\mathcal{K} = 3$ et $\mathcal{K} = 26$ pour les seuils respectifs de $\lambda = 0.05$ et $\lambda = 0.4$ (cf. Tab. 4.3), il peut être nécessaire de construire plusieurs tables d'index LSH avec des nombres de concaténations différents pour couvrir toute la plage de seuils.

Plusieurs variantes de cette opération existent dans la littérature pour ne récupérer qu'une partie des résultats. Principalement,

- **La recherche des $k$ plus proches voisins** : Cette opération retourne les $k$ enregistrements les plus proches de l'enregistrement de l'utilisateur.
- **La recherche du plus proche voisin** : Cette opération retourne l'enregistrement le plus proche de l'enregistrement de l'utilisateur.

En utilisant l'algorithme précédent, il est possible de mettre en œuvre ces deux opérations en adaptant l'étape ③. Plus particulièrement, lors du filtrage des identifiants d'enregistrements potentiellement similaires avec les esquisses, il est possible de ne sélectionner que les $k$ plus proches esquisses ou l'esquisse la plus similaire. Cependant, il est important de noter que la première requête effectuée lors de l'étape ② ne garantit pas forcément de retourner $k$ identifiants ou même un seul, mais uniquement une forte proportion des enregistrements similaires pour un seuil donné. Il est donc nécessaire de prévoir un mécanisme de gestion des cas où aucun ou trop peu de résultats ne sont retournés.

## 7.4 L'évaluation de la recherche par similarité

Les expériences en utilisant Hadoop 3.3.4 et HBase 2.5.8 sur une grappe de quatre machines. Chaque machine possède les caractéristiques suivantes : Intel(R) Xeon(R) CPU E5-2650 @2.60 GHz, 64 GB de mémoire dont 10 GB réservés pour les régions de HBase, et de deux HDD d'une capacité de 1 TB chacun.

Au cours de nos expérimentations, nous avons constaté que la table des index occupaient une grande partie de l'espace de stockage nécessaire. Cependant, il est possible de réduire le nombre de cellules de la table index en regroupant plusieurs couples d'identifiants et d'esquisses sous une même valeur, ce qui permet de réduire l'espace nécessaire et d'améliorer les performances lors des parcours. Nous avons utilisé cette optimisation lors des expériences, ce qui a permis de réduire la taille de la table d'environ 1 TB sur 3 TB. De plus, pour garantir une répartition équitable des données

sur l'ensemble des nœuds de la grappe, les clefs sont salées. Cela signifie qu'un préfixe, qui détermine le nœud du cluster, est ajouté à chaque clé.

Nous évaluons la recherche par similarité sur des séquences en utilisant l'ensemble de données METACLUST (cf. section 5.3.6), représentant 166 GB pour 680 millions de séquences d'une longueur moyenne de 220. Pour mesurer les temps moyens de réponse aux requêtes, nous avons utilisé un échantillon de 5 000 séquences aléatoires de l'ensemble de données et nous avons effectué une recherche par similarité pour chacune de ces séquences. Nous faisons également varier le seuil de la distance de Levenshtein normalisée entre 5 % et 15 %, ce qui signifie que le nombre d'itérations varie entre 4 et 64 sans modification de la table des index en reprenant les paramètres de l'algorithme *MRSF-join* (cf. tab. 5.1) pour la taille des $q$-grammes et des esquisses, et le Théorème 2 et Corollaire 2.1 (cf. section 3.2.5.a) pour paramétrer le nombre d'itérations.

| Performances moyennes par requête | $\lambda = 0.05$ | | $\lambda = 0.10$ | | $\lambda = 0.15$ | |
|---|---|---|---|---|---|---|
| | LSH | Naïf | LSH | Naïf | LSH | Naïf |
| Temps moyen d'exécution [s] | 0.296 | 4399 | 0.362 | 4581 | 0.517 | 4930 |
| Nombre moyen de comparaisons d'esquisses | 13 | — | 28.8 | — | 93.79 | — |
| Nombre moyen de distances calculées | 2.32 | $681 \times 10^6$ | 7.79 | $681 \times 10^6$ | 36.87 | $681 \times 10^6$ |
| Nombre moyen de résultats | 1.5 | 2.2 | 3.8 | 4.1 | 5.5 | 5.5 |

Tab. 7.1 : Les performances de la recherche de similarité en utilisant Apache HBase et en variant la distance seuil entre 5 % et 15 % en comparaison à une version naïve nécessitant le calcul de toutes les distances.

Pour la version naïve, nous n'avons pas effectué la recherche par similarité sur l'échantillon complet formé des 5 000 séquences, mais seulement sur les dix premières séquences, puisqu'il faut environ une heure pour comparer une seule séquence à l'ensemble du jeu de données METACLUST. Les chiffres présentés dans le tableau Tab. 7.1 étant les valeurs moyennes résultantes de la recherche par similarité (de 10 séquences sur 5 000) dans l'ensemble de données METACLUST pour présenter les performances des recherches par similarité dans METACLUST.

Nous remarquons que la recherche par similarité ne produit pas l'ensemble des résultats lorsque le seuil de la distance de Levenshtein est fixé à 5 % ou 10 %. Cependant, il est possible d'augmenter l'espérance de collision (cf. section 3.3.6) et donc le nombre

d'itérations pour retrouver les séquences similaires manquantes. Les temps de connexion au serveur Apache HBase n'ont pas été pris en compte dans les mesures de performance, car la connexion est réutilisée pour l'ensemble des requêtes de recherche de similarité.

## 7.5 Conclusion

Dans ce chapitre, nous avons présenté l'opération de recherche par similarité, permettant de retrouver dans des mégadonnées l'ensemble des enregistrements similaires. Nous avons proposé un algorithme reposant sur l'utilisation de tables HBase pour stocker les données et les indexer grâce à une famille LSH. En résumé,

- L'implémentation permet de traiter efficacement l'opération de recherche par similarité sur de longues séquences et même pour de très larges seuils. Cette solution est également générique, dans le sens où l'utilisateur n'a besoin de fournir qu'une famille de fonctions de hachage LSH et une spécification de l'entrée pour indexer et accéder aux enregistrements.
- Elle est également extensible et supporte des ensembles de données très volumineux, et ce, peu importe le nombre de séquences similaires que le serveur doit renvoyer.
- Les index peuvent être réutilisés pour de larges plages de seuils, ce qui permet de limiter l'espace nécessaire pour les stocker au sein de tables HBase pouvant atteindre plusieurs pétaoctets de données.

# Chapitre 8

# Conclusion et futurs travaux

Le dernier chapitre de cette thèse résume nos principales contributions et décrit les perspectives et les travaux futurs. Notre recherche s'est concentrée principalement sur l'opération de jointure par similarité permettant de retrouver tous les couples d'objets similaires dans des mégadonnées. Nous avons utilisé l'environnement Apache Hadoop pour traiter cette opération sur une grappe de machines pour des ensembles de données très volumineuses. En résumé,

- Lors du chapitre 2, nous avons fait l'état de l'art des traitements de l'équi-jointure séquentielle et parallèle, ainsi que les différents déséquilibres pouvant survenir lors des étapes d'évaluation de la jointure. Nous avons également passé en revue et examiné les algorithmes permettant d'éviter les effets de ces déséquilibres et de garantir une répartition équitable de la charge sur l'ensemble des machines de la grappe. Enfin, nous avons introduit l'environnement Apache Hadoop et les implémentations de ces algorithmes de jointure dans cet environnement.

- Dans le chapitre 3, nous avons examiné les différentes techniques de traitement de la jointure par similarité pour différents objets et distances. Nous nous sommes intéressés à la distance de Jaccard pour comparer les ensembles, à la distance de Fréchet pour les trajectoires et à la distance de Levenshtein pour le traitement de séquences. Nous avons classé ces techniques en fonction de la complétude des résultats produits. Les approches exhaustives n'étant pas assez performantes pour le traitement de très grandes masses de données, nous avons donc opté pour des méthodes approximatives pour la suite.

- Lors du chapitre 4, nous avons présenté un algorithme de jointure par similarité, appelé *MRS-join*, pour traiter des trajectoires et des ensembles. Cet algorithme utilise des fonctions de hachage LSH permettant de réduire l'espace de recherche tout en fournissant une preuve probabiliste sur la complétude des résultats. L'algorithme

*MRS-join* permet de réduire les coûts de communication et de lecture/écriture sur disque aux seules données pertinentes tout en évitant les effets des déséquilibres et garantit une répartition équitable de la charge sur les nœuds de la grappe en adaptant les méthodes développées pour l'équi-jointure basées sur l'utilisation de schémas de communication randomisés. Nous avons analysé théoriquement cet algorithme avec un modèle de coût et évalué sa performance sur de très gros volumes de données.

– Le chapitre 5 a présenté une optimisation de l'algorithme précédent en filtrant les données en amont du calcul de la jointure à l'aide d'esquisses. Cette optimisation a permis de réduire considérablement les coûts de communication et le nombre de comparaisons en de la phase de redistribution des données. Nous avons également présenté une famille de fonctions LSH sous "hypothèse" pour le traitement des séquences, permettant de conserver un alignement lors du hachage et de réduire considérablement l'espace de recherche et les coûts de communication. Nous avons évalué cet algorithme en utilisant un modèle de coût et une série d'expériences sur un très grand ensemble de données de protéines, et ce, pour de très grands seuils de la distance de Levenshtein, afin de démontrer son efficacité.

– Dans le chapitre 6, nous avons présenté une solution générique basée sur un squelette algorithmique pour le traitement de la jointure par similarité. Cette solution permet à l'utilisateur d'éviter de rentrer dans les détails de l'implémentation des différents algorithmes de jointure par similarité sur des architectures distribuées et de ne fournir que trois composants : un algorithme de distance, une famille de fonctions LSH et un algorithme de lecture des enregistrements. Cette solution a été implémentée dans un contexte de mégadonnées dans l'environnement Apache Hadoop, ainsi que dans un contexte de calcul haute performance (HPC) en utilisant la bibliothèque *FastFlow*.

– Lors du chapitre 7, nous avons présenté l'opération de recherche par similarité et une implémentation utilisant LSH. Cette solution est basée sur le framework Apache HBase pour stocker, indexer, accéder à de très grandes masses de données. Nous avons évalué ses performances en utilisant un grand ensemble de données de protéines et pour de grands seuils de la distance de Levenshtein. Les expériences ont confirmé l'efficacité et l'extensibilité de l'approche.

## 8.1 Travaux futurs

Plusieurs directions semblent intéressantes pour poursuivre le développement des solutions de jointure par similarité et de recherche par similarité.

### 8.1.1 Une solution générique et graphique pour diverses applications

Nous nous sommes concentrés sur le côté algorithmique de la jointure par similarité et de la recherche par similarité, et sur le développement d'une solution générique de ces opérations basées sur les frameworks Apache Hadoop et Apache HBase.

Cependant, il serait intéressant d'étendre cette approche avec une interface graphique permettant à un utilisateur novice, de procéder à des opérations de recherche ou de jointure par similarité dans des mégadonnées (issues de différentes sources, images, séquences de gènes…). Pour la recherche par similarité, l'utilisateur n'aura qu'à fournir un enregistrement et éventuellement un seuil pour retrouver les enregistrements similaires correspondants dans un grand jeu de données prétraitées pour une distance donnée, ce qui simplifierait l'extraction de connaissances à partir de grands ensembles de données et la rendrait plus facilement accessible.

Nous nous sommes également attachés à être le plus générique possible pour répondre à l'enjeu de variété des données. Toutefois, le cadre théorique LSH peut s'adapter à divers domaines et applications [Jaf21]. Il serait donc intéressant d'adapter cette solution pour des applications plus ciblées. Le développement d'un outil conçu pour détecter des malwares dans de larges collections de logiciels permettraient, par exemple, de renforcer la sécurité des infrastructures [OCN14].

### 8.1.2 Le développement de familles de fonctions LSH

Bien que des familles de fonctions LSH existent pour certaines applications [Jaf21] et distances, celles-ci ne sont pas nécessairement adaptées à un contexte de mégadonnées.

Afin d'obtenir des performances satisfaisantes sur des ensembles de données volumineux, il est nécessaire d'utiliser une famille de fonctions LSH permettant de distinguer facilement les enregistrements similaires des enregistrements dissimilaires. De plus, il est essentiel de définir les paramètres adéquats pour assurer un traitement efficace de ces opérations, ce qui peut s'avérer complexe pour un utilisateur inexpérimenté, car ils sont parfois liés aux spécificités des jeux de données. Par exemple, les paramètres de notre solution de jointure par similarité sur des séquences nécessiteraient une adaptation si les séquences ont de très petites tailles (cf. section 5.3.1). De même, l'opération de

recherche et de jointure par similarité sur de très longues trajectoires nécessiteraient une famille de fonctions LSH ayant des probabilités constantes et les paramètres adaptés pour les traiter efficacement (cf. section 5.1.3).

Enfin, nous nous sommes intéressés à la jointure par similarité et la recherche par similarité en utilisant une masse de données de protéines très volumineuses. Il serait intéressant de poursuivre ces travaux en étudiant les séquences d'ADN et d'ARN. Pour l'étude des génomes, une famille de fonctions LSH a été introduite dans la littérature [Ond16], offrant la possibilité d'explorer les relations évolutives entre les différents groupes d'organismes vivants [Pas91].

Toutefois, la conception d'une famille de fonctions LSH pour les lectures, c'est-à dire les séquences produites lors du séquençage à haut débit ouvrirait des perspectives pour diverses applications dans le domaine de la bio-informatique, notamment l'étude de la diversité microbienne dans des échantillons environnementaux [ODS13 ; FJ16]. La jointure par similarité est utilisée dans ce contexte pour comparer les lectures d'ADN prélevées dans un milieu avec des bases de données de séquences microbiennes connues, ce qui permet d'identifier et de répertorier les espèces présentes.

### 8.1.3 La gestion de requêtes complexes

Nous avons traité l'opération de jointure par similarité pour un ou deux ensembles de données fournis par l'utilisateur. Une prochaine étape pourrait être de développer des outils pour gérer des requêtes plus complexes, pouvant combiner plusieurs sources de données ou intégrer un ou plusieurs flux de données en temps réel [Wan09 ; LC09 ; DG16 ; Yan20]. Ces opérations offrent de nombreux cas d'utilisation et permettent notamment de détecter les duplications en ligne, telles que des articles populaires ou des actualités identiques diffusées sous des titres et des auteurs différents, des fichiers suspects envoyés sous des noms variés, ainsi que des téléchargements très demandés partagés via des adresses web distinctes.

En conclusion, ce travail ouvre des perspectives intéressantes pour cibler diverses applications dans divers domaines et nous espérons qu'il apportera une contribution à la littérature.

# Annexe A

# Appendices

## A.1 Fichier XML de notre solution générique

```
<configuration>
  <property>
    <name>input.format</name>
    <value>reader.SetInput</value>
    <description>Input reader class</description>
  </property>
  <property>
    <name>distance.name</name>
    <value>distance.Jaccard</value>
    <description>Distance class</description>
  </property>
  <property>
    <name>lsh.name</name>
    <value>lsh.MinHash</value>
    <description>LSH class</description>
  </property>
  <property>
    <name>distance.threshold</name>
    <value>0.5</value>
    <description>Distance threshold</description>
  </property>
  <property>
    <name>lsh.p1</name>
    <value>0.5</value>
    <description>LSH probabilities</description>
  </property>
  <property>
    <name>lsh.p2</name>
    <value>0.25</value>
```

```
    <description>LSH probabilities</description>
  </property>
  <property>
    <name>t.max</name>
    <value>100000</value>
    <description>Chunk size</description>
  </property>
  <property>
    <name>f.max</name>
    <value>1000</value>
    <description>Random communication templates parameter</description>
  </property>
</configuration>
```

Exemple A.1: Exemple d'un fichier XML pris en entrée par notre solution.

# Publications associées à la thèse


[Riv22a] Sébastien Rivault, Mostafa Bamha, Sébastien Limet et Sophie Robert. « A Scalable Similarity Join Algorithm Based on MapReduce and LSH ». In : *International Journal of Parallel Programming* 50.3–4 (2022), p. 360-380. doi : 10.1007/s10766-022-00733-6.

[Riv22b] Sébastien Rivault, Mostafa Bamha, Sébastien Limet et Sophie Robert. « Towards a Scalable Set Similarity Join Using MapReduce and LSH ». In : *Computational Science – ICCS 2022*. Cham : Springer International Publishing, 2022, p. 569-583.

[Riv24] Sébastien Rivault, Mostafa Bamha, Sébastien Limet et Sophie Robert. « Towards a Scalable MapReduce Sequence Similarity Join Processing for Very Large Datasets ». In : *Submitted to JPDC int. Journal* (2024).

[Ton24] Nicolò Tonci, Sébastien Rivault, Mostafa Bamha, Sophie Robert, Sébastien Limet et Massimo Torquati. « LSH SimilarityJoin pattern in FastFlow ». In : *International Journal of Parallel Programming. To Appear* (2024).

# Bibliographie


[Afr12] Foto Afrati, Anish Das Sarma, David Menestrina, Aditya Parameswaran et Jeffrey Ullman. « Fuzzy Joins Using MapReduce ». In : *Proceedings - International Conference on Data Engineering* (2012), p. 498-509. doi : 10.1109/ICDE.2012.66.

[Ald17] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick et Massimo Torquati. « Fastflow : high-level and efficient streaming on multi-core ». In : *Programming multi-core and many-core computing systems, parallel and distributed computing* (2017). doi : 10.1002/9781119332015.ch13.

[And09] Alexandr Andoni. « Nearest Neighbor Search : the Old, the New, and the Impossible ». Thèse de doct. Massachusetts Institute of Technology, 2009.

[AGK06] Arvind Arasu, Venkatesh Ganti et Raghav Kaushik. « Efficient Exact Set-Similarity Joins ». In : *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB '06. Seoul, Korea : VLDB Endowment, 2006, p. 918-929.

[Arr] The Square Kilometre Array. *Exploring the Universe with the world's largest radio telescop*. url : https://www.skatelescope.org/.

[AB13] Nikolaus Augsten et Michael H. Böhlen. « Similarity Joins in Relational Database Systems ». In : *Synthesis Lectures on Data Management* 5.5 (19 nov. 2013), p. 1-124. doi : 10.2200/S00544ED1V01Y201310DTM038.

[AC22] M Aumüller et Matteo Ceccarello. « Implementing Distributed Similarity Joins using Locality Sensitive Hashing ». eng. In : OpenProceedings.org, 2022, p. 13.

[BI18] Arturs Backurs et Piotr Indyk. « Edit Distance Cannot Be Computed in Strongly Subquadratic Time (Unless SETH is False) ». In : *SIAM Journal on Computing* 47.3 (2018), p. 1087-1097. doi : 10.1137/15M1053128.

[BE03] M. Bamha et M. Exbrayat. « Pipelining a Skew-Insensitive Parallel Join Algorithm ». In : *Parallel Processing Letters* 13.03 (2003), p. 317-328. DOI : 10.1142/S0129626403001306.

[BH99] M. Bamha et G. Hains. « A Frequency adaptive join algorithm for Shared Nothing machines ». In : *Journal of Parallel and Distributed Computing Practices (PDCP), Volume 3, Number 3, pages 333-345* (1999). Appears also in Progress in Computer Research, F. Columbus Ed. Vol. II, Nova Science Publishers, 2001.

[Bam00] Mostafa Bamha. « Parallélisme et équilibrage de charges dans le traitement de la jointure et de la multi-jointure sur des architectures SN ». Thèse de doctorat dirigée par Hains, Gaétan Informatique Orléans 2000. Thèse de doct. 2000.

[Bam05] Mostafa Bamha. « An Optimal Skew-insensitive Join and Multi-join Algorithm for Distributed Architectures ». In : *Database and Expert Systems Applications*. Sous la dir. de Kim Viborg Andersen, John Debenham et Roland Wagner. Berlin, Heidelberg : Springer Berlin Heidelberg, 2005, p. 616-625.

[Bar04] Ziv Bar-Yossef, T. S. Jayram, Robert Krauthgamer et Ravi Kumar. « Approximating Edit Distance Efficiently ». In : *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*. FOCS '04. USA : IEEE Computer Society, 2004, p. 550-559. DOI : 10.1109/FOCS.2004.14.

[BMS07] Roberto J. Bayardo, Yiming Ma et Ramakrishnan Srikant. « Scaling up All Pairs Similarity Search ». In : *Proceedings of the 16th International Conference on World Wide Web*. Banff, Alberta, Canada, 2007, p. 131-140. DOI : 10.1145/1242572.1242591.

[BR96] Hal Berghel et David Roach. « An Extension of Ukkonen's Enhanced Dynamic Programming ASM Algorithm ». In : *ACM Trans. Inf. Syst.* 14.1 (1996), p. 94-106. DOI : 10.1145/214174.214183.

[Bla10] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita et Yuanyuan Tian. « A comparison of join algorithms for log processing in MapReduce ». In : *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. SIGMOD '10. Indianapolis, Indiana, USA : ACM, 2010, p. 975-986. DOI : 10.1145/1807167.1807273.

[Blo70] Burton H. Bloom. « Space/time trade-offs in hash coding with allowable errors ». In : *Commun. ACM* 13.7 (juill. 1970), p. 422-426. DOI : 10.1145/362686.362692.

[Bri14] Karl Bringmann. « Why Walking the Dog Takes Time : Frechet Distance Has No Strongly Subquadratic Algorithms Unless SETH Fails ». In : *Proceedings of the 2014 IEEE 55th Annual Symposium on Foundations of Computer Science*. FOCS '14. USA : IEEE Computer Society, 2014, p. 661-670. DOI : 10.1109/FOCS.2014.76.

[BK15] Karl Bringmann et Marvin Künnemann. « Quadratic Conditional Lower Bounds for String Problems and Dynamic Time Warping ». In : *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*. 2015, p. 79-97. DOI : 10.1109/FOCS.2015.15.

[Bro98] A.Z. Broder. « On the Resemblance and Containment of Documents ». In : *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*. Compression and Complexity of SEQUENCES 1997. Salerno, Italy : IEEE Comput. Soc, 1998, p. 21-29. DOI : 10/fqk7hr.

[Bro97] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse et Geoffrey Zweig. « Syntactic Clustering of the Web ». In : *Computer Networks and ISDN Systems*. Papers from the Sixth International World Wide Web Conference 29.8 (1997), p. 1157-1166. DOI : 10/br259g.

[Buc17] Kevin Buchin, Maike Buchin, Wouter Meulemans et Wolfgang Mulzer. « Four Soviets Walk the Dog : Improved Bounds for Computing the FréChet Distance ». In : *Discrete and Computation Geometry* 58.1 (2017), p. 180-216. DOI : 10.1007/s00454-017-9878-7.

[CW79] J. Lawrence Carter et Mark N. Wegman. « Universal Classes of Hash Functions ». In : *Journal of Computer and System Sciences* 18.2 (1er avr. 1979), p. 143-154. DOI : 10/dbprb6.

[CDS19] Matteo Ceccarello, Anne Driemel et Francesco Silvestri. « FRESH : Fréchet Similarity with Hashing ». In : *Algorithms and Data Structures*. Sous la dir. de Zachary Friggstad, Jörg-Rüdiger Sack et Mohammad R Salavatipour. Cham : Springer International Publishing, 2019, p. 254-268.

[CGK16] Diptarka Chakraborty, Elazar Goldenberg et Michal Koucký. « Streaming Algorithms for Embedding and Computing Edit Distance in the Low Distance Regime ». In : *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '16. Cambridge, MA, USA : Association for Computing Machinery, 2016, p. 712-725. DOI : 10.1145/2897518.2897577.

[Cha06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes et Robert E. Gruber. « Bigtable : A Distributed Storage System for Structured Data ». In : *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2006, p. 205-218.

[CGK06] S. Chaudhuri, V. Ganti et R. Kaushik. « A Primitive Operator for Similarity Joins in Data Cleaning ». In : *22nd International Conference on Data Engineering (ICDE'06)*. 2006, p. 5-5. doi : 10.1109/ICDE.2006.9.

[Chr19] Tobias Christiani. « Fast Locality-Sensitive Hashing Frameworks for Approximate Near Neighbor Search ». In : *Similarity Search and Applications : 12th International Conference, SISAP 2019, Newark, NJ, USA, October 2–4, 2019, Proceedings*. Newark, NJ, USA : Springer-Verlag, 2019, p. 3-17. doi : 10.1007/978-3-030-32047-8_1.

[DKT17] Søren Dahlgaard, Mathias Bæk Tejs Knudsen et Mikkel Thorup. « Practical Hash Functions for Similarity Estimation and Dimensionality Reduction ». In : *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Red Hook, NY, USA, 2017, p. 6618-6628.

[Das07] Abhinandan S. Das, Mayur Datar, Ashutosh Garg et Shyam Rajaram. « Google news personalization : scalable online collaborative filtering ». In : *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. Banff, Alberta, Canada : Association for Computing Machinery, 2007, p. 271-280. doi : 10.1145/1242572.1242610.

[DG16] Gianmarco De Francisci Morales et Aristides Gionis. « Streaming similarity self-join ». In : *Proc. VLDB Endow.* 9.10 (juin 2016), p. 792-803. doi : 10.14778/2977797.2977805.

[DG08] Jeffrey Dean et Sanjay Ghemawat. « MapReduce : Simplified Data Processing on Large Clusters ». In : *Communications of the ACM* 51.1 (2008), p. 107-113. doi : 10.1145/1327452.1327492.

[Den14] Dong Deng, Guoliang Li, Shuang Hao, Jiannan Wang et Jianhua Feng. « MassJoin : A mapreduce-based method for scalable string similarity joins ». In : *2014 IEEE 30th International Conference on Data Engineering*. 2014, p. 340-351. doi : 10.1109/ICDE.2014.6816663.

[DSD02] D. Dey, S. Sarkar et P. De. « A distance-based approach to entity reconciliation in heterogeneous databases ». In : *IEEE Transactions on Knowledge and Data Engineering* 14.3 (2002), p. 567-582. doi : 10.1109/TKDE.2002.1000343.

[DHW12] Anne Driemel, Sariel Har-Peled et Carola Wenk. « Approximating the Fréchet Distance for Realistic Curves in Near Linear Time ». In : *Discrete and Computation Geometry* 48.1 (2012), p. 94-127. doi : 10.1007/s00454-012-9402-z.

[DS17] Anne Driemel et Francesco Silvestri. « Locality-Sensitive Hashing of Curves ». In : *33rd International Symposium on Computational Geometry (SoCG 2017)*. Sous la dir. de Boris Aronov et Matthew J. Katz. T. 77. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 37 :1-37 :16. doi : 10.4230/LIPIcs.SoCG.2017.37.

[FJ16] Denis Faure et Dominique Joly. *La génomique environnementale : la révolution du séquençage à haut débit*. OCLC : 959711659. ISTE Editions, 2016.

[Fie18] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser et Johann-Christoph Freytag. « Set Similarity Joins on Mapreduce : An Experimental Survey ». In : *Proceedings of the VLDB Endowment* 11.10 (2018), p. 1110-1122. doi : 10/gnqjwg.

[Fli] Apache Flink. url : https://flink.apache.org/.

[GIM99] Aristides Gionis, Piotr Indyk et Rajeev Motwani. « Similarity Search in High Dimensions via Hashing ». In : *Proceedings of the 25th International Conference on Very Large Data Bases*. VLDB '99. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1999, p. 518-529.

[Gra01] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan et Divesh Srivastava. « Approximate string joins in a database (almost) for free ». English (US). In : *VLDB 2001 - Proceedings of 27th International Conference on Very Large Data Bases*. Sous la dir. de Peter M. G. Apers, Paolo Atzeni, Richard T. Snodgrass, Stefano Ceri, Kotagiri Ramamohanarao et Stefano Paraboschi. VLDB 2001 - Proceedings of 27th International Conference on Very Large Data Bases. Morgan Kaufmann, 2001, p. 491-500.

[Had] Apache Hadoop. url : https://hadoop.apache.org/.

[Has09] M. Al Hajj Hassan. « Parallélisme et équilibrage de charges dans le traitement de la jointure sur des architectures distribuées ». Thèse de doctorat dirigée par Loulergue, Frédéric Informatique Orléans 2009. Thèse de doct. 2009.

[HB15] M. Al Hajj Hassan et M. Bamha. « Towards Scalability and Data Skew Handling in GroupBy-Joins using MapReduce Model ». In : *Procedia Computer Science* 51 (2015). International Conference On Computational Science, ICCS 2015, p. 70-79. DOI : 10.1016/j.procs.2015.05.200.

[HBL14] M. Al Hajj Hassan, M. Bamha et F. Loulergue. « Handling Data-skew Effects in Join Operations Using MapReduce ». In : *Procedia Computer Science* 29 (2014). 2014 International Conference on Computational Science, p. 145-158. DOI : 10.1016/j.procs.2014.05.014.

[HBa] Apache HBase. URL : https://hbase.apache.org/.

[HTY17] Xiao Hu, Yufei Tao et Ke Yi. « Output-Optimal Parallel Algorithms for Similarity Joins ». In : *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS'17 : International Conference on Management of Data. Chicago Illinois USA : ACM, 2017, p. 79-90. DOI : 10.1145/3034786.3056110.

[HYT19] Xiao Hu, Ke Yi et Yufei Tao. « Output-Optimal Massively Parallel Algorithms for Similarity Joins ». In : *ACM Transactions on Database Systems* 44.2 (2019), p. 1-36. DOI : 10/gnr4r4.

[Hya10] Doug Hyatt, Gwo-Liang Chen, Philip F. LoCascio, Miriam L. Land, Frank W. Larimer et Loren J. Hauser. « Prodigal : prokaryotic gene recognition and translation initiation site identification ». In : *BMC Bioinformatics* 11.1 (2010), p. 119. DOI : 10.1186/1471-2105-11-119.

[Ind02] Piotr Indyk. « Approximate Nearest Neighbor Algorithms for Frechet Distance via Product Metrics ». In : *Proceedings of the Eighteenth Annual Symposium on Computational Geometry - SCG '02*. The Eighteenth Annual Symposium. Barcelona, Spain : ACM Press, 2002, p. 102-106. DOI : 10.1145/513400.513414.

[Ind04] Piotr Indyk. « Approximate Nearest Neighbor under Edit Distance via Product Metrics ». In : *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '04. New Orleans, Louisiana : Society for Industrial et Applied Mathematics, 2004, p. 646-650.

[IM98] Piotr Indyk et Rajeev Motwani. « Approximate Nearest Neighbors : Towards Removing the Curse of Dimensionality ». In : *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. New York, NY, USA, 1998, p. 604-613. DOI : 10.1145/276698.276876.

[Jaf21] Omid JAFARI, Preeti MAURYA, Parth NAGARKAR, Khandker Mushfiqul ISLAM et Chidambaram CRUSHEV. *A Survey on Locality Sensitive Hashing Algorithms and their Applications*. 2021.

[Jia14] Yu JIANG, Guoliang LI, Jianhua FENG et Wen-Syan LI. « String Similarity Joins : An Experimental Evaluation ». In : *Proc. VLDB Endow.* 7.8 (2014), p. 625-636. DOI : 10.14778/2732296.2732299.

[KSV10] Howard KARLOFF, Siddharth SURI et Sergei VASSILVITSKII. « A Model of Computation for MapReduce ». In : *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '10. Austin, Texas : Society for Industrial et Applied Mathematics, 2010, p. 938-948.

[KST15] Ben KIMMETT, Venkatesh SRINIVASAN et Alex THOMO. « Fuzzy Joins in MapReduce : An Experimental Study ». In : *Proc. VLDB Endow.* 8.12 (2015), p. 1514-1517. DOI : 10.14778/2824032.2824049.

[KTS16] Ben KIMMETT, Alex THOMO et Venkatesh SRINIVASAN. « Fuzzy joins in MapReduce : Edit and Jaccard distance ». In : *2016 7th International Conference on Information, Intelligence, Systems & Applications (IISA)*. 2016, p. 1-6. DOI : 10.1109/IISA.2016.7785408.

[Kon17] Maximilian KONZACK, Tom J. MCKETTERICK, Tim OPHELDERS, Maike BUCHIN, Luca GIUGGIOLI, Jed LONG, Trisalyn NELSON, Michel A. WESTENBERG et Kevin BUCHIN. « Visual Analytics of Delays and Interaction in Movement Data ». In : *International Journal of Geographical Information Science* 31.2 (2017), p. 320-345. DOI : 10.1080/13658816.2016.1199806.

[Leh16] Oliver LEHMBERG, Dominique RITZE, Robert MEUSEL et Christian BIZER. « A Large Public Corpus of Web Tables Containing Time and Context Metadata ». In : *Proceedings of the 25th International Conference Companion on World Wide Web*. The 25th International Conference Companion. Montréal, Québec, Canada, 2016, p. 75-76. DOI : 10.1145/2872518.2889386.

[LL98] A LERNER et S LIFSCHITZ. « A study of workload balancing techniques on parallel join algorithms ». In : *International Conference on Parallel and Distributed Processing Techniques and Applications*. 1998, p. 966-973.

[LK10] Ping LI et Christian KÖNIG. « B-Bit Minwise Hashing ». In : *Proceedings of the 19th International Conference on World Wide Web*. WWW '10. Raleigh, North Carolina, USA : Association for Computing Machinery, 2010, p. 671-680. DOI : 10.1145/1772690.1772759.

[LOZ12] Ping Li, Art B Owen et Cun-Hui Zhang. « One Permutation Hashing ». In : *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'12. Lake Tahoe, Nevada : Curran Associates Inc., 2012, p. 3113-3121.

[LC09] Xiang Lian et Lei Chen. « Efficient Similarity Join over Multiple Stream Time Series ». In : *IEEE Transactions on Knowledge and Data Engineering* 21.11 (2009), p. 1544-1558. DOI : 10.1109/TKDE.2009.27.

[LTO94] Hongjun Lu, K. L. Tan et Beng-Chin Ooi. *Query Processing in Parallel Relational Database Systems*. Washington, DC, USA : IEEE Computer Society Press, 1994.

[MAB16] Willi Mann, Nikolaus Augsten et Panagiotis Bouros. « An Empirical Evaluation of Set Similarity Join Techniques ». In : *Proceedings of the VLDB Endowment* 9.9 (2016), p. 636-647. DOI : 10.14778/2947618.2947620.

[Mar19] Guillaume Marçais, Dan DeBlasio, Prashant Pandey et Carl Kingsford. « Locality-sensitive hashing for the edit distance ». In : *Bioinformatics* 35.14 (juill. 2019), p. i127-i135. DOI : 10.1093/bioinformatics/btz354.

[McC21] Samuel McCauley. « Approximate Similarity Search Under Edit Distance Using Locality-Sensitive Hashing ». In : *24th International Conference on Database Theory (ICDT 2021)*. Sous la dir. de Ke Yi et Zhewei Wei. T. 186. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany : Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 21 :1-21 :22. DOI : 10.4230/LIPIcs.ICDT.2021.21.

[MAE07] Ahmed Metwally, Divyakant Agrawal et Amr El Abbadi. « Detectives : detecting coalition hit inflation attacks in advertising networks streams ». In : *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. Banff, Alberta, Canada : Association for Computing Machinery, 2007, p. 241-250. DOI : 10.1145/1242572.1242606.

[MF12] Ahmed Metwally et Christos Faloutsos. « V-SMART-Join : A Scalable Mapreduce Framework for All-Pair Similarity Joins of Multisets and Vectors ». In : *Proc. VLDB Endow.* 5.8 (2012), p. 704-715. DOI : 10.14778/2212351.2212353.

[ODS13] Aisling O'Driscoll, Jurate Daugelaite et Roy D. Sleator. « 'Big data', Hadoop and cloud computing in genomics ». In : *Journal of Biomedical Informatics* 46.5 (2013), p. 774-781. DOI : https://doi.org/10.1016/j.jbi.2013.07.001.

[Ond16] Brian D. Ondov, Todd J. Treangen, Páll Melsted, Adam B. Mallonee, Nicholas H. Bergman, Sergey Koren et Adam M. Phillippy. « Mash : fast genome and metagenome distance estimation using MinHash ». In : *Genome Biology* 17.1 (juin 2016), p. 132. doi : 10.1186/s13059-016-0997-x.

[OCN14] Ciprian Oprişa, Marius Checicheş et Adrian Năndrean. « Locality-sensitive hashing optimizations for fast malware clustering ». In : *2014 IEEE 10th International Conference on Intelligent Computer Communication and Processing (ICCP)*. 2014, p. 97-104. doi : 10.1109/ICCP.2014.6936960.

[OR07] Rafail Ostrovsky et Yuval Rabani. « Low Distortion Embeddings for Edit Distance ». In : *J. ACM* 54.5 (2007), 23-es. doi : 10.1145/1284320.1284322.

[Pas91] BJ Paster et al. « Phylogenetic analysis of the spirochetes ». In : *Journal of Bacteriology* 173.19 (1991), p. 6101-6109.

[PA16] Ripon Patgiri et Arif Ahmed. « Big Data : The V's of the Game Changer Paradigm ». In : *2016 IEEE 18th International Conference on High Performance Computing and Communications ; IEEE 14th International Conference on Smart City ; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2016, p. 17-24. doi : 10.1109/HPCC-SmartCity-DSS.2016.0014.

[Riv22a] Sébastien Rivault, Mostafa Bamha, Sébastien Limet et Sophie Robert. « A Scalable Similarity Join Algorithm Based on MapReduce and LSH ». In : *International Journal of Parallel Programming* 50.3–4 (2022), p. 360-380. doi : 10.1007/s10766-022-00733-6.

[Riv22b] Sébastien Rivault, Mostafa Bamha, Sébastien Limet et Sophie Robert. « Towards a Scalable Set Similarity Join Using MapReduce and LSH ». In : *Computational Science – ICCS 2022*. Cham : Springer International Publishing, 2022, p. 569-583.

[Riv24] Sébastien Rivault, Mostafa Bamha, Sébastien Limet et Sophie Robert. *Towards a Scalable MapReduce Sequence Similarity Join Processing for Very Large Datasets*. Research Report. Submitted to JPDC int. Journal, 2024.

[Ron17] Chuitian Rong, Chunbin Lin, Yasin N. Silva, Jianguo Wang, Wei Lu et Xiaoyong Du. « Fast and Scalable Distributed Set Similarity Joins for Big Data Analytics ». In : *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 2017, p. 1059-1070. doi : 10.1109/ICDE.2017.151.

[SD89] Donovan A. Schneider et David J. DeWitt. « A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment ». In : *SIGMOD Rec.* 18.2 (juin 1989), p. 110-121. DOI : 10.1145/66926.66937.

[SY90] M. Seetha Lakshmi et P.S. Yu. « Effectiveness of parallel joins ». In : *IEEE Transactions on Knowledge and Data Engineering* 2.4 (1990), p. 410-424. DOI : 10.1109/69.63253.

[Sha48] C. E. Shannon. « A mathematical theory of communication ». In : *The Bell System Technical Journal* 27.3 (1948), p. 379-423. DOI : 10.1002/j.1538-7305.1948.tb01338.x.

[Shr17] Anshumali Shrivastava. « Optimal densification for fast and accurate minwise hashing ». In : *International Conference on Machine Learning*. PMLR. 2017, p. 3154-3163.

[SL14a] Anshumali Shrivastava et Ping Li. « Densifying One Permutation Hashing via Rotation for Fast Near Neighbor Search ». In : *Proceedings of the 31st International Conference on Machine Learning*. International Conference on Machine Learning. 2014, p. 557-565.

[SL14b] Anshumali Shrivastava et Ping Li. « Improved densification of one permutation hashing ». In : *Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence*. UAI'14. Quebec City, Quebec, Canada : AUAI Press, 2014, p. 732-741.

[Sil16] Yasin Silva, Jason Reed, Kyle Brown, Adelbert Wadsworth et Chuitian Rong. « An experimental survey of mapreduce-based similarity joins ». English (US). In : *Similarity Search and Applications - 9th International Conference, SISAP 2016, Proceedings*. T. 9939 LNCS. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 9th International Conference on Similarity Search and Applications, SISAP 2016 ; Conference date : 24-10-2016 Through 26-10-2016. Germany : Springer Verlag, 2016, p. 181-195. DOI : 10.1007/978-3-319-46759-7_14.

[SR12] Yasin N. Silva et Jason M. Reed. « Exploiting MapReduce-based similarity joins ». In : *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA : Association for Computing Machinery, 2012, p. 693-696. DOI : 10.1145/2213836.2213935.

[SRT12] Yasin N. Silva, Jason M. Reed et Lisa M. Tsosie. « MapReduce-based similarity join for metric spaces ». In : *Proceedings of the 1st International Workshop on Cloud Intelligence*. Cloud-I '12. Istanbul, Turkey : Association for Computing Machinery, 2012. DOI : 10.1145/2347673.2347676.

[Spa] Apache Spark. URL : https://spark.apache.org/.

[SKB07] E. Sriraghavendra, Karthik K et C. Bhattacharyya. « Fréchet Distance Based Approach for Searching Online Handwritten Documents ». In : *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*. IEEE Computer Society, 2007, p. 461-465. DOI : 10.1109/ICDAR.2007.121.

[SS18] Martin Steinegger et Johannes Söding. « Clustering huge protein sequence sets in linear time ». In : *Nature Communications* 9.1 (2018), p. 2542. DOI : 10.1038/s41467-018-04964-5.

[TSP08] Martin Theobald, Jonathan Siddharth et Andreas Paepcke. « Spotsigs : Robust and efficient near duplicate detection in large web collections ». In : *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*. 2008. DOI : 10.1145/1390334.1390431.

[Tho17] Mikkel Thorup. « Fast and powerful hashing using tabulation ». In : *Communications of the ACM* 60.7 (2017), p. 94-101. DOI : 10.1145/3068772.

[Ton23] Nicolò Tonci, Sébastien Rivault, Mostafa Bamha, Sophie Robert, Sébastien Limet et Massimo Torquati. *LSH SimilarityJoin pattern in FastFlow*. Research Report. To appear to IJPP int. Journal, 2023.

[Tor19] Massimo Torquati. « Harnessing Parallelism in Multi/Many-Cores with Streams and Parallel Patterns ». Thèse de doct. University of Pisa, 2019.

[Tra20] Thi To Quyen Tran. « Filters based fuzzy big joins ». Thèse de doct. University of Rennes 1, France, 2020.

[TRA20] Thi-To-Quyen TRAN, Thuong-Cang PHAN, Anne LAURENT et Laurent D'ORAZIO. « Optimization for Large-Scale Fuzzy Joins Using Fuzzy Filters in MapReduce ». In : *2020 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. 2020, p. 1-8. DOI : 10.1109/FUZZ48607.2020.9177610.

[TRA18] Thi-To-Quyen TRAN, Thuong-Cang PHAN, Anne LAURENT et Laurent D'Orazio. « Improving Hamming distance-based fuzzy join in MapReduce using Bloom Filters ». In : *2018 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. 2018, p. 1-7. DOI : 10.1109/FUZZ-IEEE.2018.8491658.

[Ukk85] Esko Ukkonen. « Algorithms for approximate string matching ». In : *Information and Control* 64.1 (1985). International Conference on Foundations of Computation Theory, p. 100-118. DOI : https://doi.org/10.1016/S0019-9958(85)80046-2.

[Ukk92] Esko Ukkonen. « Approximate string-matching with q-grams and maximal matches ». In : *Theoretical Computer Science* 92.1 (1992), p. 191-211. DOI : 10.1016/0304-3975(92)90143-4.

[VG84] P. Valduriez et G. Gardarin. « Join and Semi-join Algorithms for a Multiprocessor Database Machine ». In : *ACM Transactions on Database Systems* 9.1 (1984), p. 133-161.

[Val90] Leslie G. Valiant. « A Bridging Model for Parallel Computation ». In : *Commun. ACM* 33.8 (1990), p. 103-111. DOI : 10.1145/79173.79181.

[VCL10] Rares Vernica, Michael J. Carey et Chen Li. « Efficient parallel set-similarity joins using MapReduce ». In : *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010. DOI : 10.1145/1807167.1807222.

[WDJ91] Christopher B. Walton, Alfred G. Dale et Roy M. Jenevein. « A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins ». In : *Proceedings of the 17th International Conference on Very Large Data Bases*. VLDB '91. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1991, p. 537-548.

[Wan14a] Sebastian Wandelt et al. « State-of-the-Art in String Similarity Search and Join ». In : *SIGMOD Rec.* 43.1 (2014), p. 64-76. DOI : 10.1145/2627692.2627706.

[Wan14b] Jingdong Wang, Heng Tao Shen, Jingkuan Song et Jianqiu Ji. *Hashing for Similarity Search : A Survey*. 2014.

[Wan09] Wei Wang, Chuan Xiao, Xuemin Lin et Chengqi Zhang. « Efficient Approximate Entity Extraction with Edit Distance Constraints ». In : *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. Providence, Rhode Island, USA : Association for Computing Machinery, 2009, p. 759-770. DOI : 10.1145/1559845.1559925.

[WO18] Martin Werner et Dev Oliver. « ACM SIGSPATIAL GIS Cup 2017 : range queries under Fréchet distance ». In : *SIGSPATIAL Special* 10.1 (2018), p. 24-27. DOI : 10.1145/3231541.3231549.

[Whi09] Tom White. *Hadoop : The Definitive Guide*. en. Sebastopol, CA : O'Reilly Media, 2009.

[WZ13] Tim Wylie et Binhai Zhu. « Protein Chain Pair Simplification under the Discrete Fréchet Distance ». In : *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 10.6 (2013), p. 1372-1383. doi : 10.1109/TCBB.2013.17.

[XLP17] Dong Xie, Feifei Li et Jeff M. Phillips. « Distributed Trajectory Similarity Search ». In : *Proc. VLDB Endowment* 10.11 (2017), p. 1478-1489. doi : 10.14778/3137628.3137655.

[Yan20] Jianye Yang, Wenjie Zhang, Xiang Wang, Ying Zhang et Xuemin Lin. « Distributed streaming set similarity join ». In : *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE. 2020, p. 565-576.

[Yu16] Minghe Yu, Guoliang Li, Dong Deng et Jianhua Feng. « String similarity search and join : a survey ». In : *Frontiers of Computer Science* 10.3 (2016), p. 399-417. doi : 10.1007/s11704-015-5900-5.

[YL19] Haitao Yuan et Guoliang Li. « Distributed In-Memory Trajectory Similarity Search and Join on Road Network ». In : *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2019 IEEE 35th International Conference on Data Engineering (ICDE). Macao, Macao : IEEE, avr. 2019, p. 1262-1273. doi : 10.1109/ICDE.2019.00115.

[Zah12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker et Ion Stoica. « Resilient distributed datasets : a fault-tolerant abstraction for in-memory cluster computing ». In : *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA : USENIX Association, 2012, p. 2.

[Zah08] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz et Ion Stoica. « Improving MapReduce Performance in Heterogeneous Environments ». In : *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California : USENIX Association, 2008, p. 29-42.

[ZZ17] Haoyu Zhang et Qin Zhang. « EmbedJoin : Efficient Edit Similarity Joins via Embeddings ». In : *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '17. Halifax, NS, Canada : Association for Computing Machinery, 2017, p. 585-594. doi : 10.1145/3097983.3098003.

[Zip49] George Kingsley Zipf. « Human behavior and the principle of least effort : An Introduction to Human Ecology ». In : *Reading, MA, Adisson-Wesley* (1949).

[Zob90] Albert L. Zobrist. « A New Hashing Method with Application for Game Playing ». In : *ICGA Journal* 13 (1990), p. 69-73.

Sébastien RIVAULT

# Parallélisme, équilibrage de charges et extensibilité du traitement des mégadonnées sur des systèmes à grande échelle

**Résumé** : Durant les deux dernières décennies, grâce à la réduction des coûts de stockage, d'échange et de traitement de l'information, le volume de données générées chaque année ne cesse d'exploser. Les enjeux liés au traitement de ces mégadonnées sont souvent décrits par la règle des 3V : le volume, la variété et la vitesse de création, de collecte, d'analyse et de partage des données. Pour stocker et analyser ces ensembles de données volumineux, il est essentiel d'utiliser des grappes de machines et des algorithmes extensibles et garantissant une répartition équitable de la charge sur les machines de traitement.

Des applications telles que le filtrage collaboratif, la déduplication et la résolution d'entités sont nécessaires pour identifier des relations dans des mégadonnées en se reposant sur une notion de similarité entre les enregistrements. Ces applications permettent entre autres de retrouver des utilisateurs ayant des goûts similaires, de nettoyer les données et de détecter des fraudes. Dans ces cas, les opérations de jointure et de recherche par similarité sont utilisées pour retrouver tous les enregistrements similaires dans une ou plusieurs collections de données.

Bien que les équi-jointures aient été largement étudiées et mises en œuvre avec succès sur des systèmes à grandes échelles, ces algorithmes ne sont pas adaptés à l'opération de jointure par similarité. Des techniques existent dans la littérature pour réduire l'espace de recherche tout en garantissant la complétude du résultat, mais leur extensibilité est fortement limitée.

Des méthodes approximatives ont été proposées pour traiter la jointure par similarité en fournissant une garantie probabiliste sur l'exhaustivité des résultats tout en réduisant l'espace de recherche. Ces méthodes reposent sur des fonctions de hachages dont les probabilités de collisions sont sensibles à la similarité des objets. Nous nous reposons sur ces méthodes pour proposer des solutions efficaces, permettant de réduire les coûts de traitement, de communication et de lecture/écriture de données sur disques aux seules données pertinentes pour diverses distances et sur divers objets, et ce, dans le but de proposer un framework générique basé sur le modèle de programmation MapReduce répondant aux enjeux de volume, de variété et de vitesse d'analyse des données.

L'efficacité et l'extensibilité des solutions proposées ont été étudiées en utilisant un modèle de coût et ont été confirmées par une série d'expériences mesurant également l'exhaustivité du résultat et la réduction de l'espace de recherche, garantissant ainsi une solution efficace quels que soient la taille, le déséquilibre des données en entrée, la distance et les seuils fournis par l'utilisateur.

**Mots clés** : Jointure par similarité, Recherche par similarité, Déséquilibre des données, Déséquilibre des résultats de jointure, Locality Sensitive Hashing (LSH), Apache Hadoop, modèle de programmation MapReduce.

## Parallelism, load balancing and scalability in BigData computing

**Abstract** : Over the past two decades, owing to the reduction of storage, exchange and data processing costs, the volume of data generated each year continues to explode. The challenges related to big data processing are often described by the 3Vs : volume, variety and velocity of data creation, analysis, and sharing. To store and analyze these large datasets, it is essential to use clusters of machines and scalable algorithms that guarantee load balance among processing nodes.

Applications such as collaborative filtering, deduplication and entity resolution are necessary to identify relationships in big datasets relying on a notion of similarity between records. These applications enable finding users with similar tastes, cleaning data and detecting frauds. In these cases, similarity join and similarity search operations are often used to retrieve all similar records in one or more datasets.

Although joins have been widely studied and successfully implemented on large-scale systems, the algorithms are not suitable for similarity join operation. Many techniques have been introduced in the literature to reduce the search space while ensuring the completeness of the result, but their scalability is limited.

Approximate methods have been proposed to handle similarity join by ignoring a very small part of the result and providing a probabilistic guarantee on the completeness of the results, while reducing the search space. These methods rely on hash functions whose collision probabilities are sensitive to the objects' similarity. We rely on these techniques to propose efficient solutions for similarity join and similarity search processing, allowing to reduce processing time, communication, and disks I/O costs to only relevant data for various distances and objects. The aim is to propose a generic framework based on the MapReduce programming model that meets the challenges of volume, variety, and velocity of big data analysis.

The efficiency and scalability of the proposed solutions were studied using a cost model and confirmed by a series of experiments measuring the result completeness and the reduction of the search space, while guaranteeing efficient similarity join processing regardless the data size and data skew, the distance and the user-defined thresholds.

**Keywords** : Similarity join, Similarity search, Data Skew, Locality Sensitive Hashing (LSH), Apache Hadoop, MapReduce programming model.

**Université Orléans, INSA Centre Val de Loire, LIFO ER 4022, France**