Concept de dictionnaire multilingue numérique et applications dérivées

Digital multilingual dictionary concept and applicable derivatives

Jean-Louis BARREAU, Université Paul-Valéry (Montpellier) & APLLOD

Jean-Louis.Barreau@univ-montp3.fr

Éric DUREUIL, APLLOD

dureuileric@live.fr

RÉSUMÉ

La technologie numérique apporte une grande souplesse dans la création de ressources lexicographiques mono ou multilingues, ainsi que la possibilité de les utiliser pour générer des outils ou des données plus spécialisées. Cela bouleverse, notamment, l'analyse de corpus et la création de dictionnaires. L'objectif essentiel de ce document est de présenter la conception d'une base de données correspondant à un dictionnaire multilingue et d'évoquer différentes possibilités d'utilisation de cette dernière, sans pour autant être exhaustif dans leur énumération. Pour que le système soit capable de gérer autant de langues que l'on désire, il faut opter pour la modularité des données afin d'optimiser les traitements. Ces approches du partitionnement et de la distribution des données sur plusieurs serveurs sera donc aussi abordée pour permettre de créer une structure souple et efficace pouvant faire face à un nombre de langues seulement limité par les moyens matériels mis en œuvre. Nous évoquerons aussi quelles solutions logicielles seraient les mieux adaptées en fonction de tel ou tel usage.

Mots clés : base de donnée distribuée ; dictionnaire multilingue ; dictionnaire numérique ; Open Data ; Open Source.

ABSTRACT

Digital technology provides great flexibility in creating mono or multilingual lexicographical resources, as well as the ability to use them to generate more specialized tools or data. This could result in the option to analyse corpora and create dictionaries. The main objective of this document is to present the concept of a database corresponding to a multilingual dictionary and also to discuss possible uses of the latter, without being exhaustive in their enumeration. For the system to be able to handle as many languages as we want, we must opt for the modular data to optimize different ways of manipulation. These approaches of partitioning and distribution of data across multiple servers will be also discussed in order to help create a flexible and efficient structure that can cope with a number of languages that are only limited by the hardware resources used. We will also discuss the ways in which software solutions could be best suited in terms of their particular usage.

Keywords: distributed database; multilingual dictionary; digital dictionary; Open Data; Open Source.

1 UN DICTIONNAIRE POUR QUOI ? POUR QUI ?

En réfléchissant un peu à ce que permet l'informatique, on peut simplifier et organiser la conception des dictionnaires de manière efficace, tout en limitant les travaux en doublon. On peut ainsi distinguer deux grandes familles de ressources :

- Les dictionnaires de référence, qui devraient être maintenus par des experts tant que faire se peut (chaque académie nationale des pays de la langue, lexicographes, universitaires, etc). Ils servent pour des requêtes complexes (création d'autres dictionnaires, recherches multicritères complexes, composante de système expert comme des outils d'analyse du langage ou lexicographiques, etc). Ils ont donc vocation à garantir la cohérence des données qu'ils contiennent et à offrir des interfaces de contrôle et d'interrogation plutôt complexes. On en crée un par langue et on peut les développer individuellement tout en les reliant entre eux facilement par le biais des concepts qui serviront de pivot entre les langues. Dans cette organisation des choses, leurs données devraient être Open-Source et en téléchargement libre, car cela permet à n'importe quel individu, institution publique ou agent économique de s'en emparer et de réaliser ses projet à partir d'elles. Cette vision a pour but aussi de faciliter les interactions entre le monde économique et celui de la lexicographie tout en évitant les partenariats de type public / privé souvent complexes et contraignants pour l'accès aux données et à la diffusion des savoirs. Grâce à cette façon de faire, un groupe d'enseignants ou une entreprise pourrait créer des outils ou des dictionnaires adaptés à leurs besoins spécifiques, par exemple. Les données initiales peuplant ces dictionnaires de référence devraient plutôt être créées par un traitement humain ou par l'assemblage de sources de données générées par des humains pour en garantir la fiabilité. Le modèle de base de données que nous décrivons plus loin dans cet article représente un dictionnaire de référence pour une langue et comment le relier à celui d'une autre langue.
- Les dictionnaires pour un public ou un support de diffusion spécialisé (scolaires, universitaires, utilisateurs lambda, etc.) qui dérivent des dictionnaires de référence. Ils peuvent être, en effet, vus comme un sous-ensemble de données de ces derniers et on peut donc automatiser leur création à partir d'un ou de plusieurs dictionnaires de référence (deux si on veut faire un dictionnaire bilingue par exemple). Leur adaptation au format de diffusion n'est qu'un simple formatage. Ces ressources pourraient donc être créées par quiconque ayant les compétences en programmation à partir des données des dictionnaires de référence. Leur création est facile à automatiser entièrement. Dans leurs versions

informatisées, les interfaces de contrôle et d'interrogation seront souvent assez sommaires. Pour les versions accessibles en ligne, ce sont généralement les moteurs de recherche qui feront le travail alors que le site, lui, ne permettra qu'un accès via l'expression dans une langue (comme c'est le cas du Wiktionnaire ou des sites de traduction en ligne comme Reverso, par exemple). L'automatisation complète de l'indexation par les moteurs de recherche sera très aisée. Néanmoins, il ne faudra jamais perdre de vue que ces derniers n'indexent pas forcément tout ce qui leur est présenté, mais qu'il le font selon des critères qui leur sont propres, comme la popularité du site concerné. Un système de recherche interne minimaliste devra donc toujours être proposé pour garantir d'atteindre toutes les entrées mises à disposition.

2 Petits rappels techniques généraux

Pour commencer, rappelons un certain nombre de notions très couramment utilisées dans les projets informatiques. Cela permettra d'avoir en tête les termes différents qui vont être abondamment employés dans les parties suivantes.

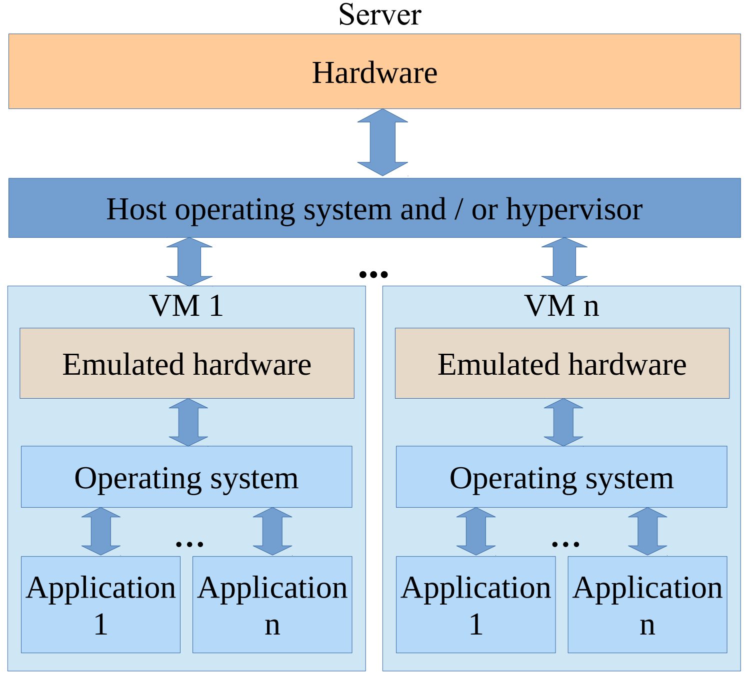
Une architecture logicielle ou matérielle souvent utilisée en informatique est l'architecture *client-serveur*. Le principe est simple : une machine ou application (le *client*) interroge une autre machine ou un autre logiciel (le *serveur* ou *service*) pour obtenir des données. Cette organisation des choses est généralement appliquée à différentes échelles, chaque élément de celle-ci étant ainsi programmé indépendamment. C'est une des techniques qui permet une forte modularité et une évolutivité de tout ou partie d'un système informatique. On retrouve évidemment là une référence à l'organisation du monde du travail où le mot *serveur* serait remplacé par le mot *entreprise*.

La *scalabilité* (de l'anglais « *scalability* ») est la capacité de pouvoir mettre à l'échelle une fonctionnalité, c'est à dire d'adapter le système, matériel ou logiciel, à la charge. On distinguera deux cas de figure :

- La *scalabilité verticale*, qui correspond au fait d'adapter les performances d'une machine en changeant ses composants (mémoire, processeur, etc.) par des versions plus puissantes, en vue d'une montée en charge potentielle. On pourra voir là l'image d'un immeuble dont on augmenterait le nombre d'étages pour avoir plus de surface, pour mieux comprendre la notion de verticalité. On doit bien souvent arrêter la machine le temps d'effectuer les modifications, ce qui rend cette technique quelque peu problématique en terme de visibilité des services et fait qu'elle reste assez peu utilisée sur les gros projets.

- La *scalabilité horizontale*, qui correspond au fait d'adapter le nombre de machines qui se partagent les traitements en parallèle pour absorber la charge de travail. Pour reprendre l'image de l'immeuble, c'est comme si on en construisait d'autres à côté pour loger plus de gens, d'où la notion d'horizontalité. C'est la technique la plus courante car elle est basée sur un principe simple « diviser pour mieux régner » et que l'on peut très aisément appliquer grâce aux machines virtuelles.

La *scalabilité* est donc un des principaux fondements des systèmes informatiques pour leur permettre d'être dimensionnés en fonction des besoins.



Une *machine virtuelle* (une « *VM* » en abrégé, en anglais) est un ordinateur complètement émulé (matériel et logiciel) en utilisant une technique de virtualisation. L'intérêt de ce système est d'avoir un ou plusieurs serveurs puissants avec n'importe quel système d'exploitation sur lesquels on va activer une ou des machines virtuelles, avec le système d'exploitation qui est le plus adapté au besoin des applications que l'on veut exécuter dessus. Le taux d'utilisation des serveurs physiques est mieux rationalisé et on va pouvoir aussi avoir une architecture matérielle et logicielle hétérogène et pourtant transparente pour ceux qui utilisent la *VM*. On augmente la tolérance à la panne, vu que la *VM* n'est pas liée forcément à telle ou telle machine réelle et qu'on peut la transférer et la démarrer sur une autre en cas de problème. De plus, on peut fixer des quotas d'utilisation pour le processeur, la mémoire, etc. Chaque *VM* est donc administrable indépendamment. Elle peut être sauvegardée à chaud (sans l'arrêter), ce qui permet en cas de panne du serveur hôte de la relancer sur un autre avec des pertes minimales. On peut aussi la cloner. Cela facilite l'administration des services, leur création, leur transfert, lancement ou arrêt. Sur une *VM*, il n'est pas possible d'exécuter un système de virtualisation. Le stockage correspond à un fichier spécial qui peut être utilisé simultanément, sous certaines conditions, par différentes *VM*.

Une *API* (acronyme anglais de « *Application Programmable Interface* ») est un ensemble de fonctions et/ou d'objets sur une thématique et dans un langage de programmation donnés qui

permet d'utiliser des fonctionnalités non natives dans celui-ci. Elles se présentent sous forme d'une bibliothèque qui, une fois intégrée au langage, permet aux éléments de l'extension d'être vus par le programmeur comme s'ils étaient des fonctionnalités natives.

3 CHOIX DU SYSTÈME DE GESTION DE BASE DE DONNÉES

Le système de gestion de base de données (acronyme : *SGBD*) est crucial dans le processus de stockage mais aussi dans la conception et la génération de données dérivées ainsi que selon la complexité des recherches que l'on veut effectuer sur elles. Il est constitué d'un logiciel principal avec un certain nombre d'utilitaires annexes et d'extensions optionnelles.

Il existe deux grandes familles de *SGBD* :

- Les *SGBD* relationnels (acronyme courant : *SGBDR*) sont généralistes et les plus couramment utilisés. Ils utilisent un langage normalisé qui s'appelle le *SQL* (acronyme anglais de « *Structured Query Language* ») pour effectuer des requêtes sur les données. Le *SQL* est basé sur l'algèbre relationnel (toutes les opérations sont prouvables et prédictibles en terme de résultats en se basant sur la branche des mathématiques qui traite de la théorie des ensembles). Les données sont structurées en tables où une entrée est une ligne (appelé aussi « *tuple* ») et les éléments de celle-ci sont organisés en colonnes qui sont typées (tous les éléments d'une colonne ont le même type). Des relations (*intégrité référentielle*, c'est-à-dire les *clés étrangères*) lient éventuellement les tables entre elles. Celles-ci peuvent être également contrôlées par des règles variées (unicité d'une valeur sur une colonne, calcul autorisant ou non la valeur à insérer, modifier ou supprimer tels que les *déclencheurs* ou bien les *contraintes d'intégrité*, par exemple). C'est ce qui permet la cohérence des données. Tous les principes *ACID* (acronyme de « *Atomicité*, *Cohérence*, *Isolation*, *Durabilité* ») sont généralement à la base de leur fonctionnement. De plus, les *SGBDR* possèdent tous un langage procédural permettant de réaliser, en plus de ce que permet le *SQL* standard, des opérations très complexes (concept de *base de données épaisses*) sans avoir besoin de faire des allers-retours entre client et serveur. On a ainsi un axe d'optimisation supplémentaire. Enfin, on notera la possibilité d'étendre leurs fonctionnalités grâce à la création de « *fonctions utilisateurs* » dans un langage compilé, principalement le *C/C++*. Les deux principaux *SGBDR* les plus complets sont des solutions payantes (*Oracle* et *Microsoft SQL Server*) mais des solutions Open-Source existent aussi (*PostgreSQL*, *MySQL*, *MariaDB*, etc).

- Les *SGBD NO-SQL* (acronyme anglais de « *Not Only SQL* ») sont des bases de données très

spécialisées. Il en existe environ 150. Elles sont, en fait, une remise au goût du jour de technologies antérieures aux *SGBDR*. Le choix du bon *SGBD NO-SQL* est primordial, sous peine d'avoir des performances plus que dégradées, car elles ne sont pas généralistes. Les données peuvent être de forme variable (chaque entrée ne contient pas forcément des éléments organisés de la même manière) ou structurée. De par leur conception, la plupart des *SGBD NO-SQL* ne sont généralement pas *ACID*. Ils exposent leurs fonctionnalités via une *API* mais cela implique un certain nombre de limites. Beaucoup utilisent le *XML* pour stocker les données. Ils sont très performants pour faire des recherches très simples, surtout sur une base avec des données peu souvent modifiées, mais ne permettent pas de croiser celles-ci directement, c'est au langage appelant de le faire. Les opérations d'insertions ou modifications n'étant pas *ACID* le plus souvent, c'est au langage appelant, là encore, de devoir prendre à sa charge le contrôle de celles-ci pour garantir les conditions ou règles qu'elles doivent respecter. Leur extensions avec des fonctionnalités utilisateur n'est pas toujours possible et peut nécessiter des compétences très avancées. Ils impliquent aussi de choisir le bon produit adapté à ce que l'on veut faire. La plupart sont proposés en Open-Source.

À partir de là, on peut donc avoir une première réflexion sur le choix *SGBDR* ou *SGBD NO-SQL* :

- Les *SGBD NO-SQL* sont très pratiques et efficaces pour faire des dictionnaires permettant des recherches simples dans le style : une expression renvoie toutes les fiches lui correspondant. C'est-à-dire, par exemple, les documents *XML* la contenant ou ayant été mis en vis-à-vis d'elle par une relation « *clé* / *valeur* ». La modification dynamique des données pourra s'avérer problématique et aussi induire une perte plus ou moins temporaire de performances. Si on veut refaire le mécanisme des jointures (croisement de données), natif dans les *SGBDR* cela revient à chercher à réinventer la roue avec un code qui sera, quel que soit le langage utilisé, presque toujours bien moins optimisé que dans ces derniers.

- Les *SGBDR* sont particulièrement adaptés pour concevoir tous les types de dictionnaires. On peut utiliser tout un tas de techniques pour optimiser les données (indexation, tables en formes normales, partitionnement éventuel, mise en cache des requêtes et de leur résultat, etc.) et obtenir des performances identiques à leurs équivalent *NO-SQL*, tout en gardant les avantages des opérations *ACID*, des jointures. Il est facile de produire des données *XML* et de peupler une base *NO-SQL* à partir de la plupart des *SGBDR*, certains ayant des extensions pour communiquer directement avec celle-ci.

Choisir l'un ou l'autre dépend donc surtout du type de projet. Un dictionnaire pour un public ou un support de diffusion spécialisé pourra très bien se faire avec les *SGBDR* ou *NO-SQL*, alors qu'un dictionnaire de référence sera mieux conçu grâce à un *SGBDR* car on aura besoin de requêter de manière complexe et d'avoir l'usage des principes *ACID*.

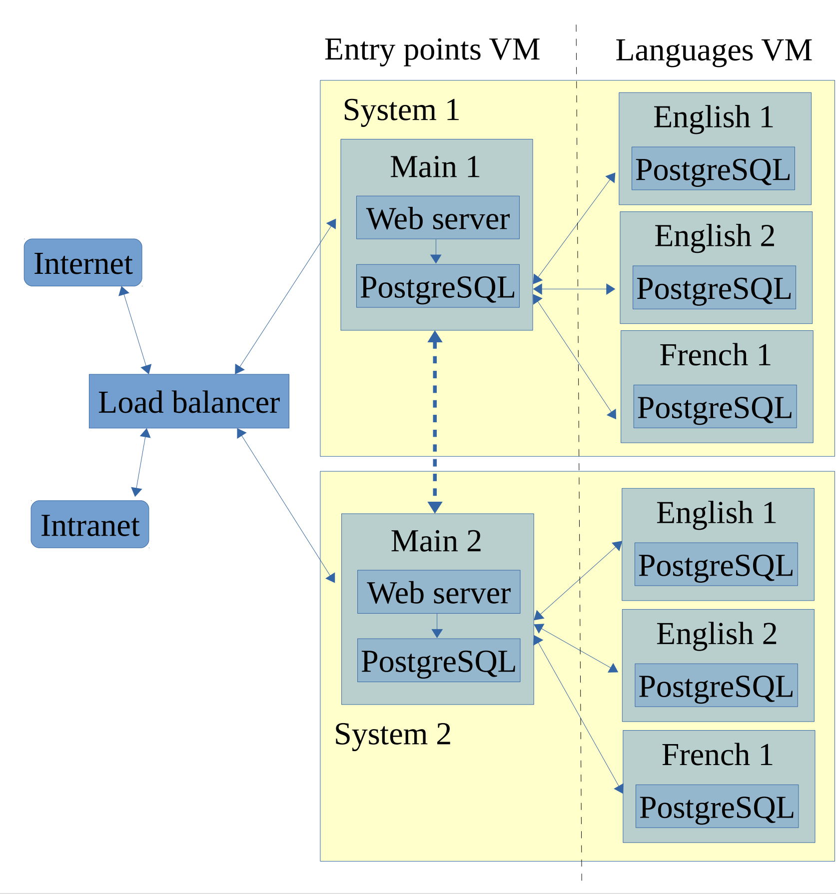
Cet article a surtout pour but de présenter la conception d'un dictionnaire de référence, donc le choix se portera sur un *SGBDR*. Celui qui paraît le plus adapté et ouvert en termes de possibilités tout en restant Open Source est *PostgreSQL*. Mais rien n'empêche d'en utiliser un autre avec une licence propriétaire comme *Oracle* ou *SQL Server*. Ces trois SGBDR sont, en effet, les principaux offrant les fonctionnalités dont nous auront besoin par la suite. Rappels sur l'intégrité référentielle :

- Une *clé primaire* est un ensemble d'une ou plusieurs colonnes d'une table identifiant de manière unique chaque ligne de celle-ci. Ses valeurs sont indexées (dans l'ordre des colonnes s'il y en a plusieurs), c'est-à-dire mémorisées dans une structure spéciale dans le but de les parcourir sans avoir à lire chaque ligne de la table pour les récupérer (ce que l'on nomme « *full scan* » en anglais, soit une lecture complète de la table). Chacune de ses valeurs servira d'identifiant pour les lignes dans tout autre index créé ensuite sur la table. Le comptage du nombre de lignes se fait sur la clé primaire. Il vaut mieux donc n'utiliser que des valeurs numériques pour des questions de performances et de place en mémoire.
- Une *clé étrangère* est un ensemble d'une ou plusieurs colonnes d'une table cible qui référencent le même nombre de colonnes (généralement la clé primaire) dans une autre qui sera la source. Le but est de s'assurer que seules les valeurs possibles dans la table cible soient celles de la table source. C'est l'un des moyens de garantir le bon croisement des données. De plus, on peut répercuter les changements dans la table source sur la table cible.

4 ORGANISATION GÉNÉRALE DU DICTIONNAIRE

On veut créer un dictionnaire multilingue. L'idée de départ est donc simple : on va voir chaque langue comme un nœud de données indépendant. Chaque nœud sera une *VM* qui embarquera PostgreSQL et les données concernant spécifiquement la langue à laquelle il sera dédié. Une *VM* servira de nœud central qui aura pour but de présenter les outils d'interrogation ou d'administration (différentes interfaces web). Cette dernière contiendra aussi PostgreSQL et toutes les données commune à toutes les langues (traduction d'éléments comme les catégories grammaticales, genre, nombre des expressions aussi bien que de ceux des interfaces). On créera donc ce que l'on appelle une bases de données distribuée (ou en grappe).

On obtient ainsi une solution très modulaire (en fonction du nombre de langues désirées) et facile à dupliquer si on a besoin d'absorber une charge plus grande de requêtes. On peut augmenter le nombre de *VM* qui décrivent les langues les plus sollicitées ou aller jusqu'à dédoubler l'intégralité de la structure si nécessaire et cacher l'ensemble ainsi obtenu derrière un répartiteur de charge ou plus simplement à des adresses web différentes pour une répartition des requêtes de manière géographique (on dispatche des systèmes à différents endroits du globe pour raccourcir les temps d'accès à ceux-ci en les rapprochant d'une partie des utilisateurs qui vont les consulter).



Dans l'exemple d'un dictionnaire multilingue *français / anglais* avec un fort taux de consultation, si on se base sur le nombre de locuteurs de ces langues et leur visibilité à l'échelon européen ou mondial, on peut estimer que l'*anglais* sera plus consulté que l'autre langue, que ce soit pour des recherches mono ou multilingues. On peut alors anticiper en répliquant plus de *VM* pour le traitement de l'*anglais*. Si on réplique une structure complète de VM, il peut alors être intéressant ou non d'automatiser la répercussion des changements car cela génère des temps de traitement non négligeables. Il peut être plus rentable de faire cela de manière non systématique. En effet, on n'est pas, dans le cadre d'un dictionnaire, sur ce que l'on peut qualifier de système critique. On voit que le côté modulaire de la conception a tout son intérêt. De la même manière, ajouter ou supprimer une langue revient à ajouter ou supprimer la ou les *VM* afférentes, si l'on néglige le traitement de mise en relation entre langues dont le détail est expliqué plus loin dans ce document.

5 La constitution du jeu de données initial pour une langue

Tout dépend de l'adaptation des données à réaliser pour les insérer dans la base. Plus le modèle de

départ sera éloigné en terme de codage (le plus souvent en termes de complexité, donc de possibilités de valeurs pour coder les choses), plus la quantité de travail manuel sera importante. Prenons un exemple parlant. Si on part d'un dictionnaire contenant une dizaine de catégories grammaticales que l'on veut utiliser pour pré-remplir un dictionnaire plus complexe qui en possède une quarantaine, alors on ne pourra pas automatiser le traitement de la colonne concernée et une intervention humaine sera requise. Il faut garder à l'esprit qu'il est toujours facile d'automatiser une réduction de la qualité de l'information ou sa traduction en un code autre, jamais son augmentation.

Pour certaines langues, ce travail d'adaptation peut concerner tellement de données pour chaque ligne qu'il peut s'avérer ne pas être une mauvaise chose de partir d'une liste de mots qu'on complétera à la main, quasiment ex-nihilo. Cette difficulté d'adaptation doit donc être parfaitement prise en compte.

Le travail préparatoire à la constitution du jeu de données correspond à l'analyse linguistique et lexicographique de la langue. On commence par établir les différents dialectes qui vont représenter les sous-langages qu'elle contiendra. On met ensuite à jour les listes de catégories grammaticales, genres, nombres, modalités qui servent à qualifier les lexies.

Le traitement principal se fera donc sur des listes générées à partir d'export de sources pré-existantes ou d'une liste de lexies que l'on qualifie manuellement. Le format d'échange le plus adapté est le *CSV* (acronyme anglais de « *Comma Separated Values* ») car il est minimaliste et ne nécessite qu'un traitement d'interprétation très léger pour la récupération des données. Utiliser du *XML*, comme cela est souvent fait dans les dictionnaires, ne devrait l'être que pour des extraits et non pas un export total de centaines de milliers d'entrées. On crée des tables spéciales pour l'import, dépendant de l'organisation des données sources. Pour une insertion avec des temps de traitements optimaux, il est préférable de scinder les listes aussi bien pour la phase de qualification des lexies que pour la phase d'insertion.

6 Langues et dialectes, comment jongler entre ces notions

Le concept de langue représente une notion assez vaste et vague qui regroupe des considérations linguistiques, politiques, administratives... Par excès, on assimile parfois la forme d'écriture à une langue comme dans le cas de l'arabe, par exemple. De plus, on peut voir que les essais de nomenclature des langages ne sont pas décisifs non plus, les spécialistes n'étant pas forcément d'accord sur la façon d'organiser les langues entre elles. La structure hiérarchique représentant l'arborescence des langues connues, souvent utilisée, ne semble pas être la plus opportune au vu de la complexité des interactions passées et actuelles entre les langues.

On doit donc essayer de trouver un système efficace tout en intégrant les nomenclatures comme *ISO* qui en représente la principale source internationale. On peut aussi avoir intérêt à connaître la ou les zone(s) géographique(s) où sont parlées une langue et ses différentes variétés. Tout ceci donne le fil directeur de ce qu'on peut générer comme modèle pour décrire cela.

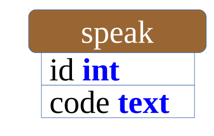
L'organisation *ISO* fournit trois standards liés aux besoins linguistiques :

- *ISO 639* qui décrit les langues, dialectes et métalangues.
- *ISO 15924* qui décrit les formes d'écriture.
- *ISO 3166* qui décrit les pays.

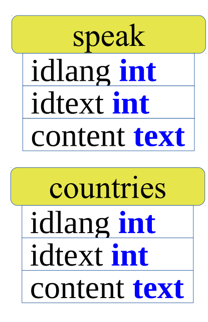
La difficulté est de décider de l'organisation des langues et dialectes. Dans la suite du document, une langue est considérée comme un langage reconnu pour un usage légal et administratif au sein d'entités institutionnelles des nations où il est employé. Un dialecte est une forme d'une langue qui a son système lexical, syntaxique et phonétique propre et qui est utilisé dans un environnement plus restreint que la langue elle-même. Employé couramment pour *dialecte régional* par opposition à *langue*, le dialecte est un système de signes et de règles combinatoires de même origine qu'un autre système considéré comme la langue, mais n'ayant pas acquis le statut culturel et social de celle-ci.

Dans la base, une langue représente un conteneur pour différents dialectes. Chacun d'eux est associé à un ou plusieurs pays. La forme d'écriture n'est pas identifiée à une langue ou un dialecte car plusieurs graphies peuvent servir pour exprimer chaque dialecte (exemple en japonais : on a les systèmes d'écriture *hiragana, katakana, kanji*, sans oublier les translittérations en caractères latins nommées *romaji* par les japonais). La forme d'écriture sera donc associée à chaque lexie.

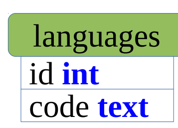
Dans l'application, nous allons distinguer deux types de langues : celle qui sert à faire les traductions pour la restitution (document ou interface) et celle qui sera l'objet d'étude du dictionnaire.



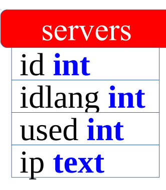
La table « *speak* » sert à lister les possibilités de traduction de l'interface. Elle est constituée d'un identifiant numérique, la colonne « *id* », qui est associé au code sur deux lettres, la colonne « *code* », qui représente une langue dans norme *ISO 639*. Le fait de choisir ce codage alphabétique sur deux caractères réduit le jeu de langues possibles à seulement celles qui sont des langues officielles et administratives, ce qui est suffisant pour une interface. La colonne « *id* » est la clé primaire de la table. On met une contrainte d'unicité sur cette colonne pour éviter qu'une même langue soit enregistrée plusieurs fois.



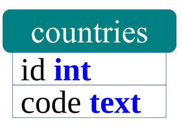
Toutes les tables associées à une ressource qui devra être restituée sous forme d'un texte pour une traduction auront un entête jaune et la même constitution. Elles se situeront dans un schéma dédié de la base nommé « *translations* ». On y retrouvera la colonne « *idlang* » qui est une clé étrangère pointant sur la colonne « *id* » de la table « *speak* », la colonne « *idtext* » qui est une clé étrangère pointant sur la colonne « *id* » de la table qui décrit la ressource à traduire, et enfin la colonne « *content* » qui contient le texte de la traduction. Ici, il est avantageux de déclarer une clé primaire sur les colonnes « *idlang* » et « *idtext* ». On pourra alors s'assurer de ne pas créer des traductions multiples pour un élément de la ressource concernée grâce à la propriété d'unicité de la clé primaire et de rechercher très efficacement si une traduction existe, ou encore savoir le nombre de langues disponibles pour une ressource, par exemple.



La table « *languages* » contient la liste des langues effectivement intégrées dans le dictionnaire. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. Vient ensuite la colonne « *code* » qui décrit sur deux ou trois caractères désignant la langue dans la norme *ISO 639*. On met une contrainte d'unicité sur cette colonne pour éviter la plupart des doublons. Cette approche laisse une grande latitude sur le choix de ce que l'on appellera une « langue » et laisse libre de considérer celles qui n'ont pas de code sur deux caractères individuels.



La table « *servers* » décrit la liste des serveurs (nœuds de données) associés à une langue du dictionnaire. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. Ensuite vient la colonne « *idlang* » qui est une clé étrangère pointant sur la colonne « *id* » de la table « *languages* ». La colonne « *used* » est une valeur initialisée à zéro qui permet de compter le nombre d'utilisations du serveur considéré en vue de la répartition de charge. On se servira de celui qui a la valeur minimale dans « *used* » pour la recherche. La colonne « *ip* », quant à elle, contient l'adresse *IPv4* ou *IPv6* (on évitera de mixer les deux si possible) du serveur. En mettant une contrainte d'unicité sur elle, on évite la plupart des risques d'association d'un même serveur avec plusieurs langues.



La table « *countries* » décrit la liste exhaustive des pays, elle est peuplée intégralement dès le départ. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. Vient ensuite la colonne « *code* » qui décrit le code sur deux ou trois caractères désignant la langue dans la norme *ISO 3166*. On met une contrainte d'unicité sur cette colonne pour éviter les doublons en

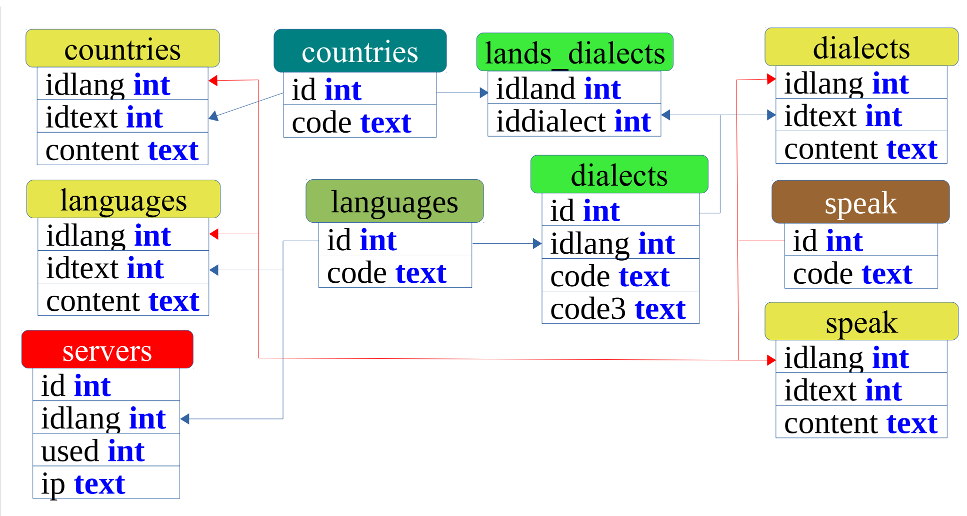
cas de mise à jour future de cette liste.

La table « *dialects* » décrit la liste des dialectes associés à une langue du dictionnaire. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. La colonne « *idlang* » est un identifiant numérique qui une clé étrangère pointant sur la colonne « *id* » de la table « *languages* ». La colonne qui suit est « *code* », un code textuel interne au dictionnaire pour nommer le dialecte. Une contrainte unique sur cette colonne garantira qu'on ne rentre pas plusieurs fois un même code pour nommer différents dialectes. Enfin, la colonne « *code3* » qui représente le code sur trois caractères désignant le dialecte dans la norme *ISO 639*, s'il existe. Cette colonne peut donc contenir des valeurs à « *null* ».

La table « *lands_dialects* » permet de définir le fait qu'un dialecte peut être parlé dans plusieurs pays. La colonne « *idland* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *countries* ». Vient ensuite la colonne « *iddialect* » qui est aussi une valeur numérique. C'est une clé étrangère pointant sur la colonne « *id* » de la table « *dialects* ». Les deux colonnes forment la clé primaire de la table.

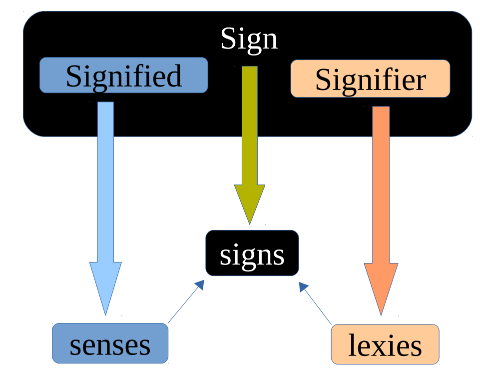
lands_dialects
idland int
iddialect int

Le schéma ci-contre met en perspective les différentes tables citées précédemment pour mieux comprendre les relations qui les lient. Les tables servant pour la traduction des ressources et celle qui liste les serveurs de langues sont aussi représentées. La table « *languages* » est donc centrale dans cette approche. C'est elle qui permettra de sélectionner le nœud de données à interroger. La table « *dialects* » sera dédoublée sur chaque serveur concerné par la langue comme on le verra plus loin dans notre explication. Notre approche est très ouverte car elle laisse libre de définir ce que l'on appelle une langue et les dialectes qu'on lui affilie. Toutes ces tables sont communes à toutes les langues du dictionnaire. Elles sont donc implémentées sur le serveur qui est le point d'entrée du projet (« *main1* » sur le schéma de l'exemple d'implémentation dans la partie 4 de ce document).



7 Décrire une langue efficacement : signe, signifié, signifiant

On reprend les concepts de linguistique formalisés par Saussure que sont le *signe*, le *signifiant* et le *signifié*. Pour mémoire, le signifié représente le concept, l'idée. Le signifiant est le support matériel du signifié (phonèmes, graphèmes, langage corporel). Le signe est, quant à lui, l'association du signifiant et du signifié.



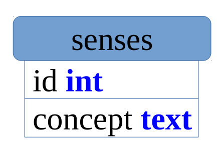
On va adapter cela en créant trois tables qui vont représenter chacun de ces concepts :

- La table « *lexies* » qui va correspondre à la notion de *signifiant*.
- La table « *signs* » qui représentera la notion de *signe*.
- La table « *senses* » qui correspondra à la notion de *signifié*.

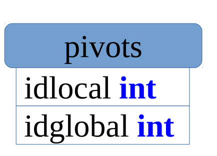
On obtient ainsi la structure centrale du dictionnaire. D'autres tables seront bien entendu nécessaires autour pour compléter les informations complémentaires de chacune de ces tables.

La date d'apparition d'une lexie et ses origines renseignent sur l'influence des autres langues et sur l'époque où elle a eu lieu. Grâce à une base de données lexicographique, on peut donc mettre facilement en évidence via des graphes ou des tableaux de données numériques (montrant l'évolution temporelle et qualitative des langues) la complexité de ces relations.

8 Le signifié, un pivot entre les langues



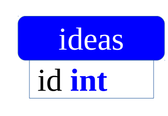
Dans notre conception du dictionnaire le sens est au centre de tout et notamment le moyen fiable de passer d'une langue à une autre. Concrètement, le sens va être représenté par une définition, c'est-à-dire une acception lié à un lemme. On s'assure de l'unicité (au moins en termes de mot à mot) de chaque définition et l'identifiant numérique associé à celle-ci au sein de la langue représentera le code unique de chaque concept pour la langue concernée. En théorie, on devrait réduire leur nombre lors de cette opération puisque les parfaits synonymes devraient avoir la même définition (par exemple *courriel* et *mail* en français). On mettra le résultat de ce traitement dans une table « *senses* » où la colonne « *id* » représentera le code et la colonne « *concept* » le texte de la définition. Cette table sera sur un nœud de données.



Pour lier les concepts, si possible strictement équivalents, entre différentes

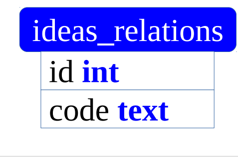
langues, on crée un code numérique unique, lui aussi, qui va représenter ce lien. On pourra parler d'un méta-identifiant de concept. Ce dernier est ensuite mis avec le code de la définition dans la table « *pivots* » de la langue. On commence par peupler cette table avec les valeur de la colonne « *id* » de la table « *senses* » que l'on met dans la colonne « *idlocal* » et on initialise à « *null* » toutes les valeurs de la colonne « *idglobal* ». Les concepts de la langue ne sont, en effet, mis en relation avec ceux d'aucune autre langue à ce stade. Cette table sera sur un nœud de données. Pour permettre son unicité, le méta-identifiant n'est créé que si on veut mettre en relation des concepts de deux langues dont la valeur « *idglobal* » de la table « *pivots* » de chacune d'elles est à « *null* ». Cela signifie qu'aucune relation préexistante ne lie l'un des concepts de chacune des deux langues à une langue tierce. Cette table sera sur un nœud de données.

On se sert d'une table « *ideas* » qui ne contient que la colonne « *id* ». Il suffit d'incrémenter la dernière valeur mémorisée en elle pour obtenir la valeur à mettre dans les colonne « *idglobal* » des tables « *pivots* » des deux langues. Dans le cas où l'une des deux valeurs ne serait pas à « *null* », alors c'est cette valeur qui serait reportée dans l'autre langue. Cette table sera sur le serveur qui est le point d'entrée.



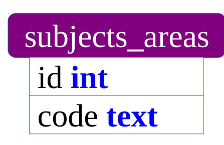
Le *champ lexical* décrit la *polysémie*, c'est-à-dire l'ensemble des relations entre les concepts (*synonymie*, *antonymie*, etc.). Si la synonymie absolue touche principalement les *signes*, le *signifié* peut lui aussi être concerné. Ce sera un moyen de rattraper certaines situations ou plusieurs valeurs de la table « *ideas* » représenteront quand même le même concept. À noter qu'il peut aussi se retrouver une configuration où on pourrait enfreindre l'unicité dans la table « *ideas* » : si on met en relation un groupe de plusieurs langues indépendamment d'un autre et qu'on cherche à les relier ensuite. Il faut donc ne jamais se retrouver dans cette situation, sous peine de devoir refaire le processus d'association sur l'un des groupes.

La table « *ideas_relations* » décrit les différents types de relations possibles dans le champ lexical. Pour limiter les entrées dans cette table, on évitera de citer les deux relations équivalant à une relation bidirectionnelle telle que l'hyperonymie (si le concept *A* inclut le concept *B*, alors le concept *B* est inclus dans le concept *A*) ou l'antonymie. En effet, il est facile de citer l'une des relations (par exemple le concept *A* est le contraire du concept *B*), l'autre, inverse, étant très facile à déduire si besoin. Pour faciliter le traitement, l'antonymie peut être représentée par l'identifiant « *-1* » et la synonymie absolue par la valeur « *0* ». Une valeur strictement positive décrirait alors une synonymie relative. On pourrait tout à fait avoir plusieurs niveaux de synonymie avant de parler d'inclusion d'un concept dans un autre.

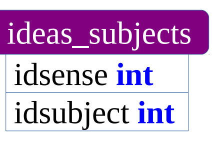


lexical_fields
idrelation **int**
idsense1 **int**
idsense2 **int**

La table « *lexical_fields* » décrit le champ lexical liant les concepts entre eux. La colonne « *idrelation* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *ideas_relations* ». La colonne « *idsense1* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *ideas* ». La colonne « *idsense2* » est aussi une valeur numérique. C'est une clé étrangère pointant sur la colonne « *id* » de la table « *ideas* ». On créera la clé primaire sur les trois colonnes dans l'ordre suivant « *idrelation* », « *idsense1* » et « *idsense2* ».

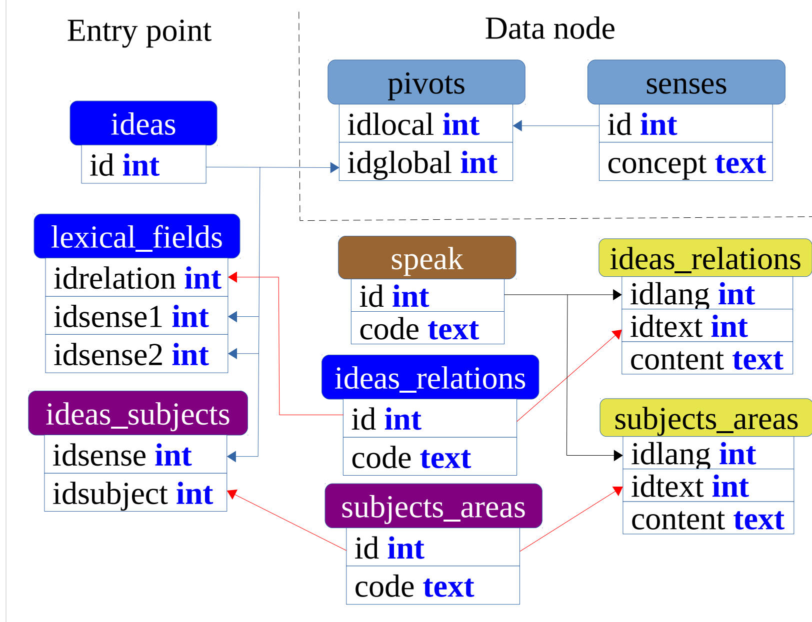


La table « *subjects_areas* » décrit les différents domaines d'activité qui permettent de regrouper les concepts par thématiques. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. Vient ensuite la colonne « *code* » qui décrit le code associé à chaque domaine d'activité. On met une contrainte d'unicité sur cette colonne pour éviter la plupart des doublons.



La table « *ideas_subjects* » permet d'associer autant de domaines d'activité que nécessaire à chaque concept. La colonne « *idsense* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *ideas* ». Vient ensuite la colonne « *idsubject* » qui est aussi une valeur numérique. C'est une clé étrangère pointant sur la colonne « *id* » de la table « *subjects_areas* ». Les deux colonnes forment la clé primaire de la table.

Le schéma ci-contre résume l'ensemble des tables qui touchent au signifié et les relations qui les lient. On notera les deux tables permettant la traduction de « *ideas_relations* » et « *subjects_areas* » qui sont stockées dans le schéma (comprendre ici : la subdivision du domaine de noms de la base de données, un peu comme un répertoire) « *translation* » (les tables avec un titre en jaune iront dedans).



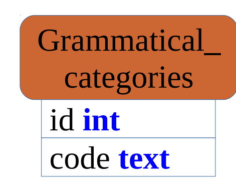
On peut ainsi travailler indépendamment dans chaque langue, sans se soucier de savoir si une idée dans deux langues différentes est représentée par le même code. On ne regarde cela que dans la phase finale du traitement, quand on s'occupe de la mise en vis-à-vis de celles-ci. Ce traitement devrait être manuel pour garantir un résultat de qualité.

9 Le signifiant, la représentation concrète

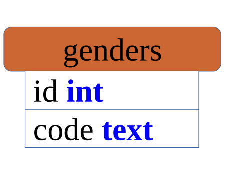
Comme nous l'avons vu précédemment, le signifiant équivaut ici à une *lexie*, c'est-à-dire une unité lexicale. Concrètement, on va créer un certain nombre de tables pour la décrire et faciliter différents contrôles et recherches.

Certaines tables, celles décrivant les caractéristiques grammaticales d'une *lexie*, seront créées sur le serveur qui sert de point d'entrée car elles sont communes à toutes les langues. Une copie de chacune d'elles sera maintenue sur tous les nœuds de données pour accélérer les traitements. En voici la liste :

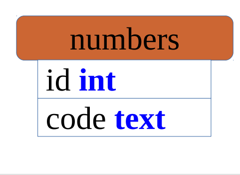
- La table « *grammatical_categories* » décrit la liste des catégories grammaticales. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. Vient ensuite la colonne « *code* » qui décrit le code associé à chaque catégorie grammaticale. On met une contrainte d'unicité sur cette colonne pour éviter la plupart des doublons.



- La table « *genders* » décrit la liste des genres en grammaire. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. Vient ensuite la colonne « *code* » qui décrit le code associé à chaque genre. On met une contrainte d'unicité sur cette colonne pour éviter la plupart des doublons.

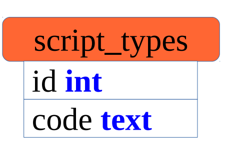


- La table « *numbers* » décrit la liste des nombres en grammaire. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. Vient ensuite la colonne « *code* » qui décrit le code associé à chaque nombre. On met une contrainte d'unicité sur cette colonne pour éviter la plupart des doublons.

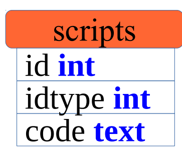


Chaque représentation graphique (une graphie qui représente un ou des mots par exemple) a un seul système d'écriture, mais peut être associée à plusieurs *lexies* et chaque représentation peut avoir plusieurs prononciations possibles. On a donc intérêt à les traiter indépendamment de la *lexie*.

La table « *script_types* » décrit la liste des types de systèmes d'écriture. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. Vient ensuite la colonne « *code* » qui décrit le code associé à chaque domaine d'activité. On met une contrainte d'unicité sur cette colonne pour éviter la plupart des doublons. La table sera sur le serveur qui sert de point d'entrée, car elle est commune à toutes les langues.



La table « *scripts* » décrit la liste des systèmes d'écriture. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. La colonne « *idtype* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *script_types* ». Vient enfin la colonne « *code* » qui décrit le code associé à chaque domaine d'activité. On met une contrainte d'unicité sur cette colonne pour éviter la plupart des doublons. Les valeurs des colonnes « *id* » et « *code* » sont données par la norme *ISO 15924*. Cette table, commune à toute les langues, sera donc implémentée sur le serveur qui sert de point d'entrée et une copie sera faite sur chaque nœud de données pour optimiser les traitements sur ceux-ci.

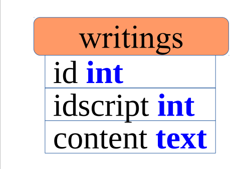


Dans une langue, on remarque qu'une expression peut avoir plusieurs prononciations et qu'il y a aussi à l'inverse l'homophonie de diverses expressions. Les tables suivantes seront sur un serveur qui est un nœud de données :

- La table « *phonetic* » décrit la prononciation des expressions qui représentent les lexies de la langue. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. Vient ensuite la colonne « *content* » qui décrit chaque écriture phonétique utilisée dans la langue. On met une contrainte d'unicité sur cette colonne pour éviter la plupart des doublons.

phonetic
id **int**
content **text**

- La table « *writing* » décrit la liste des expressions qui représentent les lexies de la langue dans un système d'écriture donné. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. La colonne « *idscript* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *scripts* ». Vient enfin la colonne « *content* » qui décrit chaque expression utilisée dans la langue. On met une contrainte d'unicité sur cette colonne pour éviter la plupart des doublons. La table sera sur le nœud de données. On aura intérêt à créer un index calculé sur la longueur en caractères de chaque

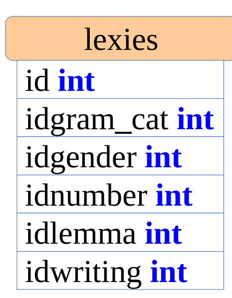


expression dans la colonne « *content* » et aussi un autre sur son premier caractère.

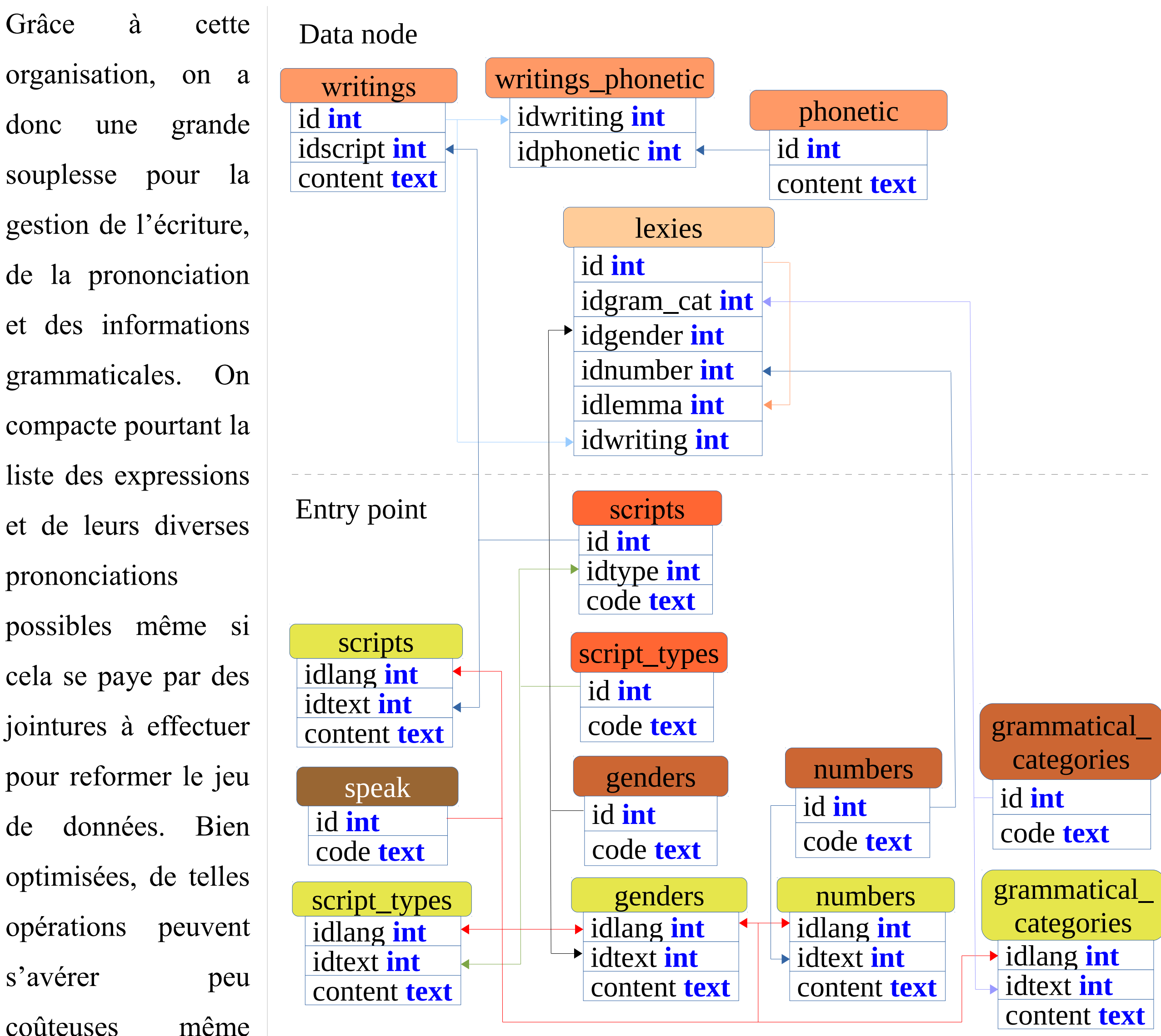
- La table « *writings_phonetic* » met en relation les expressions qui représentent les lexies de la langue dans un système d'écriture donné avec les prononciations qui leur correspondent. La colonne « *idwriting* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *writings* ». Vient ensuite la colonne « *idphonetic* » qui est aussi une valeur numérique. C'est une clé étrangère pointant sur la colonne « *id* » de la table « *phonetic* ». Les deux colonnes forment la clé primaire de la table.

writings_phonetic
idwriting **int**
idphonetic **int**

- La table « *lexies* » décrit la liste des *lexies* de la langue. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. La colonne « *idgram_cat* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *grammatical_categories* ». La colonne « *idgender* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *genders* ». La colonne « *idnumber* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *numbers* ». La colonne « *idlemma* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *lexies* ». Si la *lexie* est un lemme, cette valeur est la même que celle de « *id* » sur la même ligne, sinon elle contiendra l'identifiant de la ligne qui est son lemme. La colonne « *idwriting* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *writings* ». Chaque clé étrangère de cette table sera indexée individuellement pour accélérer les recherches. On pourra éventuellement mettre une contrainte d'unicité multicolonnes sur l'ensemble des clés étrangères, pour empêcher de créer des *lexies* doublons.



Grâce à cette organisation, on a donc une grande souplesse pour la gestion de l'écriture, de la prononciation et des informations grammaticales. On compacte pourtant la liste des expressions et de leurs diverses prononciations possibles même si cela se paye par des jointures à effectuer pour reformer le jeu de données. Bien optimisées, de telles opérations peuvent s'avérer peu coûteuses même entre de grosses tables, car on ne devrait lire que peu de lignes dans chacune d'elles. La réduction du jeu de données à seulement ce dont on a besoin est la clé de l'optimisation pour les jointures sur de grande taille et le fait qu'elles ne s'opèrent que sur des identifiants numériques que l'on lit dans des index au lieu des valeurs et donc des lignes complètes dans les tables elles-mêmes. Les copies des tables « *genders* », « *numbers* » et « *grammatical_categories* » sur chaque nœud de données ne servent que pour que les clés des tables de celui-ci soient internes au serveur. On évite ainsi des connexions réseau pour lire leur équivalent sur le serveur qui sert de point d'entrée, ce qui plomberait les temps de traitement. Les « tables maîtres » sur le serveur qui sont sur ce dernier servent à garantir leur évolution indépendamment des serveurs qui sont des nœuds de données à qui elles transmettent leurs modifications aux « tables esclaves » éponymes qui leur sont associées.



Ce schéma représente les tables sur lesquelles les recherches porteront systématiquement. C'est donc une partie particulièrement importante au niveau de l'impact sur l'efficacité des traitements.

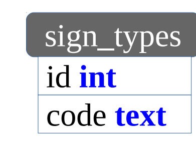
Dans cette optique, les index calculés sur la table « *writings* » trouvent aussi tout leur sens pour réduire les temps de parcours sur celle-ci.

10 LE SIGNE, LE LIANT INTERNE DE LA LANGUE

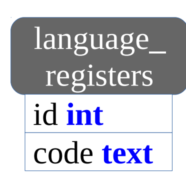
Le signe est l'ensemble formé par un sens et une lexie dans un dialecte d'une langue. C'est donc sur lui que portera aussi un certain nombre de choses comme les périodes d'utilisation connue, permettant de suivre l'apparition, la réapparition d'un signe ou la notion de modalité ainsi que le registre de langage ou les exemples pour la mise en situation, ou enfin le type d'acception (concrète, abstraite, figurée).

On commence par les tables communes à toutes les langues qui seront donc sur le serveur qui sert de point d'entrée et dont une copie sur chaque nœud de données permettra d'optimiser le contrôle des insertions et modifications sur la table « *signs* » :

- La table « *sign_types* » liste les différents types d'acceptions. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. Vient ensuite la colonne « *code* » qui décrit chaque type d'acception. On met une contrainte d'unicité sur cette colonne pour éviter la plupart des doublons.

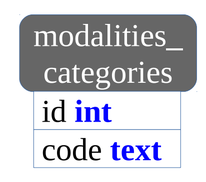


- La table « *language_registers* » liste les registres de langage. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. Vient ensuite la colonne « *code* » qui décrit chaque registre de langage. On met une contrainte d'unicité sur cette colonne pour éviter la plupart des doublons.



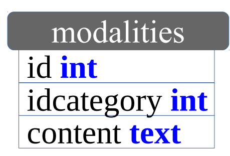
Selon les langues, le signe peut porter plus ou moins d'informations sur la modalité. La notion de modalité représente l'attitude du sujet parlant vis-à-vis de son énoncé. Elle est donc indispensable dans l'analyse du discours. Vu le nombre d'aspects sémantiques qu'elle peut couvrir, on va la classer dans différentes catégories. Ce classement sera surtout intéressant, en pratique, si on veut faire une interface qui permet de l'utiliser pour la saisie des données du dictionnaire ou encore de restreindre une recherche en fonction d'une ou plusieurs modalités. Dans notre système, nous offrons la possibilité d'associer plusieurs modalités à un signe :

- La table « *modalities_categories* » liste les catégories de modalités prises en compte dans le dictionnaire. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. Vient



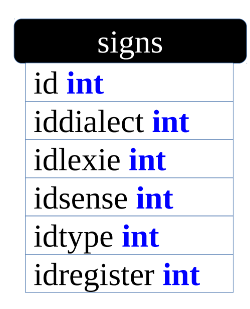
ensuite la colonne « *code* » qui décrit chaque catégorie de modalité. On met une contrainte d'unicité sur cette colonne pour éviter la plupart des doublons.

- La table « *modalities* » liste les modalités possible pour les signes. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. La colonne « *idsign* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *modalities_categories* ». Vient enfin la colonne « *content* » qui décrit chaque modalité. On met une contrainte d'unicité sur cette colonne pour éviter la plupart des doublons.

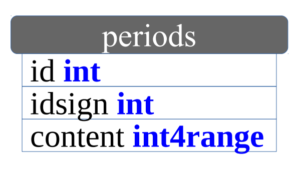


Les tables suivantes seront toutes sur un nœud de données car elles sont spécifiques à chaque langue :

- La table « *signs* » liste toutes les associations entre les sens et les lexies. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. La colonne « *iddialect* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *dialects* ». La colonne « *idlexie* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *lexies* ». La colonne « *idsense* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *senses* ». La colonne « *idtype* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *sign_types* ». La colonne « *idregister* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *language_registers* ». Chaque clé étrangère de cette table sera indexée individuellement pour accélérer les recherches. On pourra éventuellement mettre une contrainte d'unicité multicolonne sur l'ensemble des clés étrangères, pour empêcher de créer des *signes* doublons. La table sera sur un serveur qui est un nœud de données.

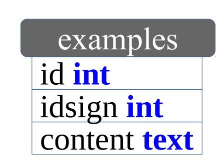


- La table « *periods* » liste les périodes d'utilisation de chaque signe. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. La colonne « *idsign* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *signs* ». Vient enfin la colonne « *content* » qui décrit chaque période sous la forme d'une plage de deux valeurs entières.

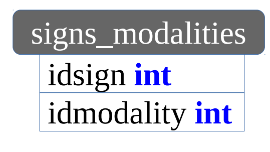


Elles peuvent ou coder une année ou une valeur représentant plus ou moins l'infini. Une plage avec une borne inférieure ou supérieure ouverte est donc possible. On met une contrainte d'unicité sur cette colonne pour éviter la plupart des doublons.

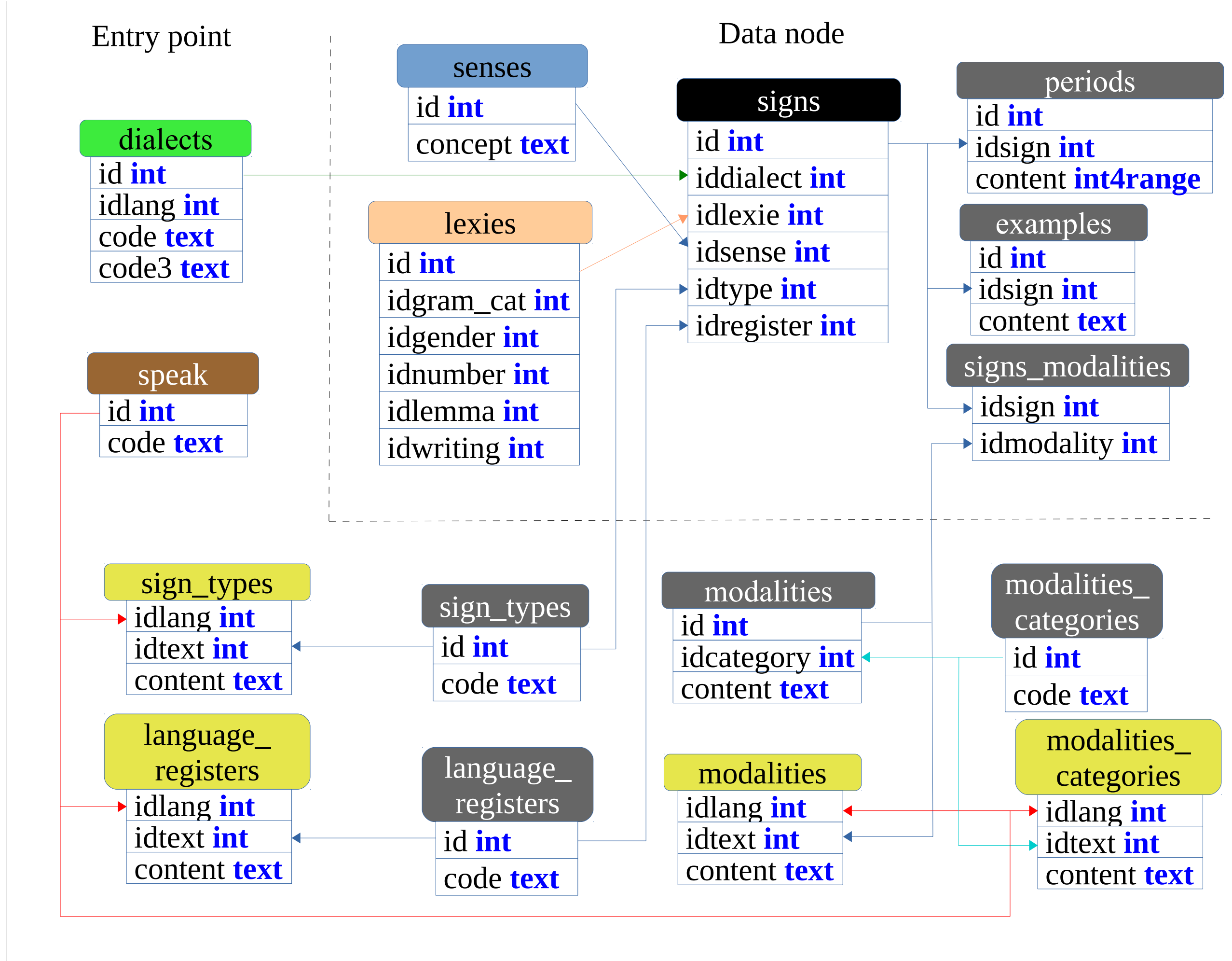
- La table « *examples* » listes des exemples, sous forme de phrases, pour chaque signe. On a une colonne « *id* » qui est un identifiant numérique unique et qui sert de clé primaire à la table. La colonne « *idsign* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *signs* ». Vient enfin la colonne « *content* » qui décrit chaque expression utilisée dans la langue. On met une contrainte d'unicité sur cette colonne pour éviter la plupart des doublons.



- La table « *signs_modalities* » liste les notions de modalité que l'on associe aux *signes*. La colonne « *idsign* » est un identifiant numérique qui est une clé étrangère pointant sur la colonne « *id* » de la table « *signs* ». Vient ensuite la colonne « *idmodality* » qui est aussi une valeur numérique. C'est une clé étrangère pointant sur la colonne « *id* » de la table « *modalities* ». Les deux colonnes forment la clé primaire de la table.



Le schéma ci-après remet en perspective la table « *signs* » et toutes celles qui gravitent autour d'elle. On y retrouve notamment ses liens avec « *senses* » et « *lexies* ». Si on l'assemble avec les deux autres schémas résumé en se servant des tables en commun entre eux, on peut reconstruire le schéma intégral de la base de données.



11 Les jointures, unions, la base du croisement de données

Dans la suite, nous allons utiliser différents types de jointures et les écrire en utilisant les opérateurs ensemblistes adéquats dédiés (« *cross join* », « *inner join* », « *left join* », etc.), et non leur écriture antérieure à la norme *SQL 99*. Cette dernière devrait être proscrite de par le fait qu'elle ne permet pas autant d'optimisation réelle et qu'elle rend la lecture des requêtes complexes plus que pénible.

Le SQL définit divers mots-clés qui définissent les différentes sections d'une requête, les principaux sont :

- « **SELECT** » qui introduit la liste des colonnes des jeux de données initiaux et des opérations qu'on réalise sur elles et donc les colonnes du jeu de données en sortie qu'on peut nommer si besoin est.
- « **FROM** » qui, dans les standards d'écriture du SQL 99 et suivants, introduit soit un jeu de données issu d'une sous-requête, soit une table. Si une jointure est faite dans la requête, ce

sera alors nommé « *le jeu de données de gauche* » pour la jointure.

- « [**CROSS** | **INNER** | **LEFT** [**OUTER**]] **JOIN** [… **ON** …] » qui introduit une jointure entre le jeu de données de gauche et la table ou celui issu d'une sous-requête qui apparaît derrière le mot-clé « JOIN » et qui sera nommé « *le jeu de données de droite* ». Dans la présentation de l'écriture de l'opérateur de jointure « [...] » montre une partie optionnelle et « ...|... » montre une alternative entre deux écritures. La clause « **ON** » est suivie d'une liste de conditions restreignant le nombre de lignes du *jeu de données de gauche* et/ou celui du *jeu de données de droite*. Si plusieurs jointures s'enchaînent le résultat de la jointure précédente devient le *jeu de données de gauche* pour la jointure en cours. Leur enchaînement impacte donc le temps de traitement, la mémoire nécessaire, etc., et il faut parfois forcer leur ordre pour optimiser les choses et ne pas espérer que l'optimiseur du *SGBDR* le fasse pour vous.
- Éventuellement un mot-clé « **WHERE** » qui introduit une liste de conditions restreignant le nombre de lignes du *jeu de données de gauche* et / ou celui du *jeu de données de droite* (si ce dernier existe).
- Éventuellement un mot-clé « **GROUP BY** » qui permet d'introduire une liste de regroupement sur les colonnes du jeu de données en sortie, obligatoire s'il y a des fonctions ensemblistes comme « **COUNT** », « **AVG** », « **SUM** », etc. qui interviennent.
- Éventuellement un mot-clé « **ORDER BY** » qui permet d'introduire une liste de classement sur les colonnes du jeu de données en sortie, ce qui change donc l'ordre final des lignes.
- Éventuellement un mot-clé « **LIMIT** » qui permet de limiter les lignes du jeu de données en sortie entre un rang de début et un rang de fin spécifiques.

Quel que soit le type de jointure, on a toujours les mêmes étapes :

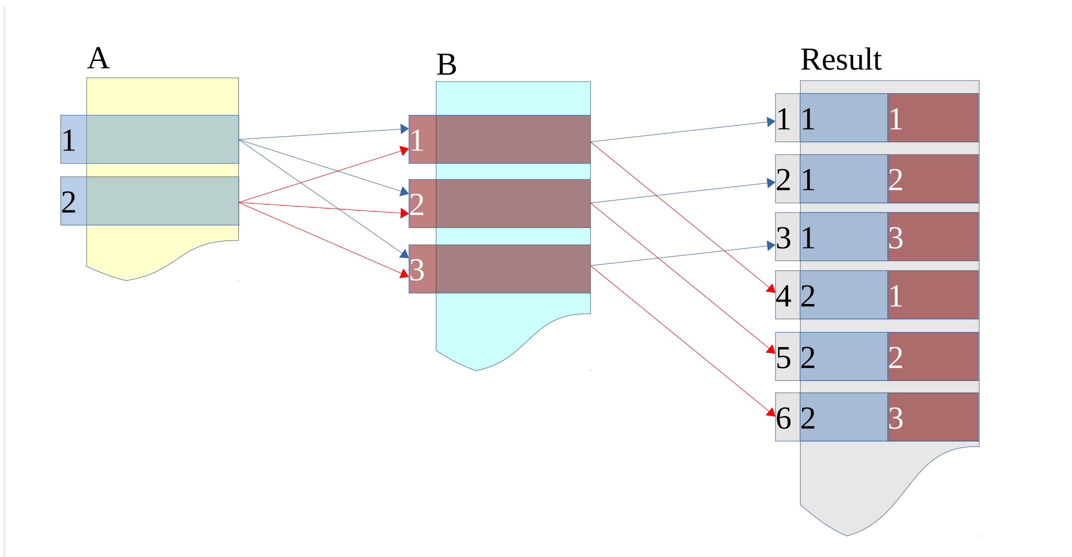
- On réduit éventuellement le nombre de lignes dans le *jeu de données de droite* et *jeu de données de gauche* en fonction des conditions derrière les mots-clés « **WHERE** » et « **ON** » s'ils sont présents.
- On identifie les conditions de jointure parmi les conditions derrière les mots-clés « **WHERE** » et « **ON** » s'ils sont présents, c'est-à-dire les conditions qui déterminent par rapport à quelles colonnes particulières dans les jeux de données obtenus on va faire la jointure.
- On fait le produit cartésien des lignes du *jeu de données de gauche* et du *jeu de données de*

droite obtenus et on ajoute, éventuellement, une partie des deux jeux de données complétées par des colonnes nulles, en fonction du type et des conditions de jointure.

- On ne garde que les colonnes choisies dans le « **SELECT** » sur chaque ligne ainsi créée, on les réorganise en appliquant les éventuelles formules, actions définies par les mots-clés pour constituer le jeu de données en sortie.

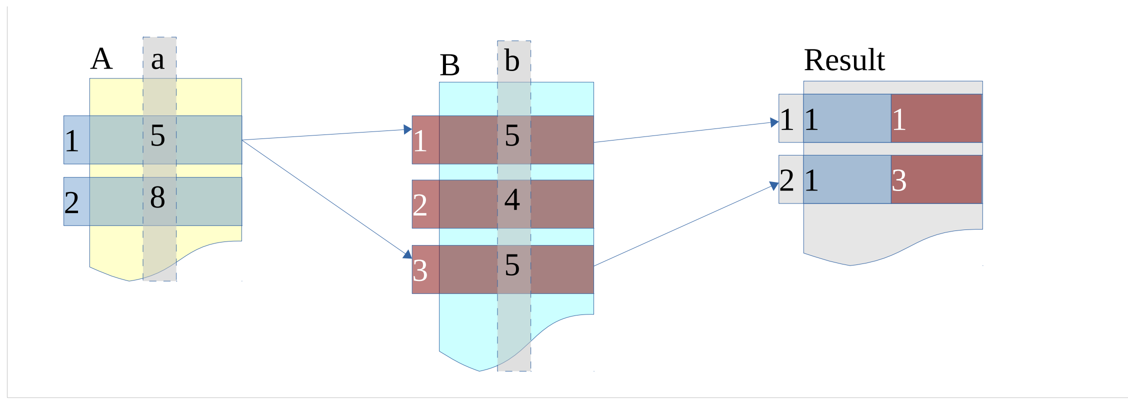
```
SELECT A.*,B.*
FROM A
CROSS JOIN B
```

La jointure la plus simple, qui est définie par l'opérateur « **CROSS JOIN** », est le produit cartésien. La clause « **WHERE** », si elle est présente, permet de limiter les lignes dans *jeu de données de droite* et *jeu de données de gauche*. Il n'y a pas de clause « **ON** » possible avec elle. Dans l'exemple ci-contre, chaque ligne en sortie sera constituée de chaque ligne de *A* à laquelle chaque ligne de *B* va être accolée, l'une après l'autre. Si *A* contient 2 lignes et *B* possède 3 lignes, alors le résultat du produit cartésien de ces deux ensembles contiendra $2 \times 3 = 6$ lignes.




```
SELECT A.*,B.*
FROM A
INNER JOIN B ON A.a=B.b
```

La jointure interne, qui est définie par l'opérateur « **INNER JOIN** », correspond à une opération qui dépend de l'intersection entre les deux jeux de données en entrée. Dans l'exemple ci-contre, les lignes en sortie seront constituées de chaque ligne de *A* accolée à toutes les lignes de *B* telles que la valeur de la colonne *a* de *A* soit identique à celle de la colonne *b* de *B*. On obtient donc bien l'intersection de *A* et *B* sur le critère *a*=*b*. À noter que les valeurs à « *null* » de a ou b même si $a = b$ ne produiront aucune ligne en sortie. Cette jointure sert donc à rapatrier les lignes communes aux deux jeux de données en entrée.

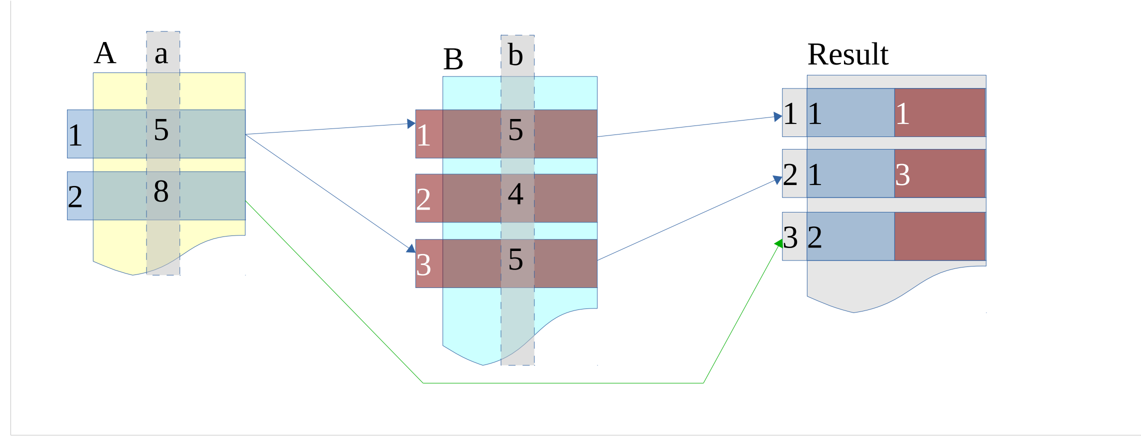


La jointure externe permet la même chose que la jointure interne mais elle rapatrie, en plus, tout ou partie des lignes du *jeu de données de gauche* ou du *jeu de données de droite* qui n'ont pas été prises dans l'équivalent de la jointure interne, complétées par des colonnes à « *null* » pour constituer une ligne en sortie. On distingue trois types de jointures externes en fonction du jeu de données dont on veut garantir la présence intégrale.

La *jointure externe gauche*, qui est définie par l'opérateur « **LEFT OUTER JOIN** » ou plus simplement « **LEFT JOIN** », garantit d'avoir toutes les lignes du *jeu de données de gauche* éventuellement complétées par celles du *jeu de données de droite*.

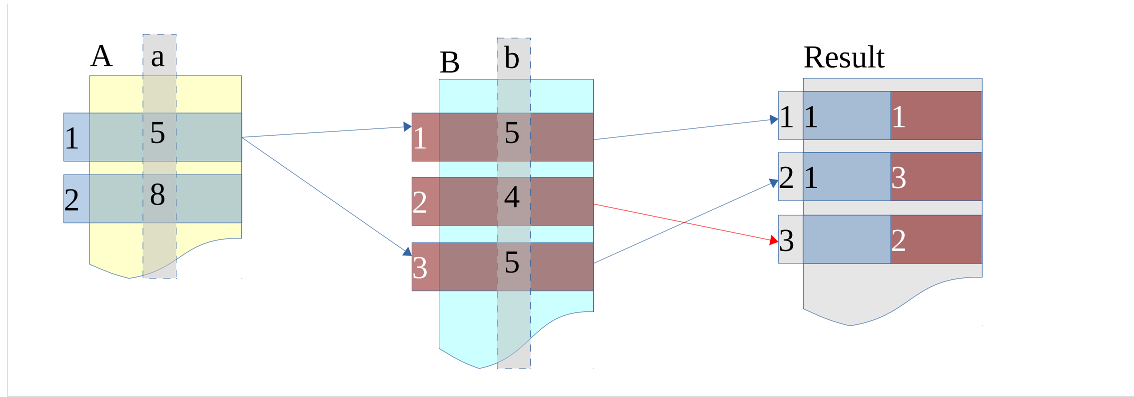
```
SELECT A.*,B.*
FROM A
LEFT JOIN B ON A.a=B.b
```

Le schéma ci-dessous permet de voir la différence qu'il y a avec la jointure interne en comparant avec son schéma. On remarque la ligne supplémentaire issue du *jeu de données de gauche* qui est complétée par des colonnes vides pour la partie qui aurait dû provenir du *jeu de données de droite*.



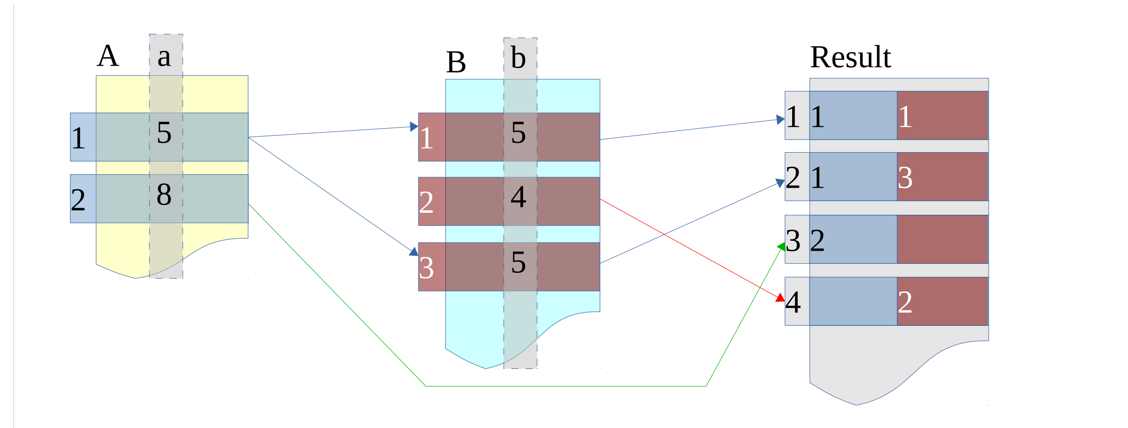
La *jointure externe droite*, qui est définie par l'opérateur « **RIGHT OUTER JOIN** » ou plus simplement« **RIGHT JOIN** », garantit d'avoir toutes les lignes du *jeu de données de droite* éventuellement complétées par celles du *jeu de données de gauche*. La *jointure externe droite* revient à inverser les deux jeux de données en entrée dans une *jointure externe gauche* et n'est donc quasiment jamais implémentée dans les *SGBDR*.

```
SELECT A.*,B.*
FROM A
RIGHT JOIN B ON A.a=B.b
```



La *jointure externe complète*, qui est définie par l'opérateur « **FULL OUTER JOIN** » ou plus simplement« **FULL JOIN** », garantit d'avoir toutes les lignes du *jeu de données de droite* complétées par celles du *jeu de données de gauche*. Mais aussi toutes les lignes des deux jeux qui n'ont pas satisfait à l'équivalent de la jointure interne en les complétant par des colonnes à « *null* ». La jointure externe complète revient à mettre en commun le résultat de la jointure externe gauche et d'une jointure externe droite qui auraient les mêmes conditions en faisant un dédoublonnage du résultat obtenu. Elle n'est donc quasiment jamais implémentée dans les *SGBDR*.

```
SELECT A.*,B.*
FROM A
FULL JOIN B ON A.a=B.b
```



On notera que si un des jeux de données en entrées n'est pas une table mais une requête, alors elle devra apparaître entre parenthèse et un alias lui être donné pour pouvoir utiliser les colonnes de sa sortie.

De même, l'enchaînement des jointures est important, il joue aussi bien :

- Sur la performance, car plus on réduit les quantités de données à traiter tôt dans le processus, plus on accélère les jointures suivantes.
- Sur le résultat final obtenu, si on mixte des jointures externes, internes ou des produits cartésiens dans la même requête, qui peut alors varier.

Diverses écritures permettent, selon les *SGBDR*, de forcer l'ordre d'exécution. Il faut toujours penser une requête comme un arbre d'opérations donc l'exécution ne doit, en général, pas laisser le moindre doute quant à l'enchaînement qui va exister.

L'union est une opération ensembliste qui consiste à rassembler deux jeux de données issus chacun d'une requête différente et qui ont le même ordre et typage pour leurs colonnes. Elle est définie par deux opérateurs :

- « UNION » pour les unions sans doublons en sortie.
- « UNION ALL » pour les unions avec doublons.

```
(SELECT A.*,B.*
FROM A
LEFT JOIN B ON A.a=B.b)
UNION
(SELECT A.*,B.*
FROM A
RIGHT JOIN B ON A.a=B.b)
```

Ci-contre, on trouvera l'exemple de l'union équivalente à une jointure externe complète. On remarquera qu'il est possible d'enchaîner les unions et d'établir des priorités entre elles grâce à des parenthèses pour préciser les divers niveaux d'inclusions. Certains *SGBDR* forcent leur utilisation autour des requêtes, d'autres

sont plus permissifs. Dans l'absolu, c'est une bonne pratique de les utiliser de manière systématique pour ne pas laisser de doute sur ce que l'on voulait énoncer comme code.

12 Exemples de requêtes

À des fins didactiques, nous allons présenter ici différents exemples de requêtes en réponse à des questions.

Pour commencer, une question très simple : **quelles sont toutes les prononciations d'une expression donnée ?** Pour cet exemple on choisit le mot « example ».

```
SELECT p.content
FROM writings w
INNER JOIN writings_phonetic wp ON w.id=wp.idwriting
INNER JOIN phonetic p ON wp.idphonetic=p.id
WHERE w.content="example"
```

Ce que l'on peut réécrire en forçant l'ordre des jointures :

```
SELECT  p.content
FROM
        (SELECT  wp.idphonetic AS result
        FROM writings w
        INNER JOIN writings_phonetic wp ON w.id=wp.idwriting
        WHERE w.content="example") a
INNER JOIN phonetic p ON a.result=p.id
```

Ce qui peut encore s'écrire avec les *CTE* (« *common exchange tables* » en anglais), qui sont des tables temporaires nommées et internes à une requête :

```
WITH
a(result) AS
    (SELECT  wp.idphonetic
    FROM writings w
    INNER JOIN writings_phonetic wp ON w.id=wp.idwriting
    WHERE w.content=”example”)
SELECT  p.content
FROM  phonetic p
INNER JOIN a ON a.result=p.id
```

Une question qui peut être intéressante sur le plan statistique cette fois : **combien y a-t-il de lemmes ?** On se rappellera qu’un lemme est une lexie qui pointe sur elle-même via sa colonne « *idlemma* ».

```
SELECT COUNT(*)
FROM lexies l
WHERE l.id=l.idlemma
```

Faisons un peu de grammaire : **Quelles sont toutes les expressions qui sont des adjectifs qualificatifs ?** Nous employons le code « qual.adj. » pour les désigner en interne dans la base.

```
SELECT w.content
FROM grammatical_categories c
INNER JOIN lexies l ON c.id=l.idgram_cat
INNER JOIN writings w ON l.idwriting=w.id
WHERE c.code=”qual.adj.”
```

Attaquons-nous à la base de la constitution d’un dictionnaire : **Quelles sont les acceptions pour une lexie dont on connaît l’identifiant ?** Ce dernier ayant été récupéré par une autre requête. On

prendra par exemple la valeur numérique « *5* ».

```
SELECT se.content
FROM signs si
INNER JOIN senses se ON se.id=si.idsense
WHERE si.idlexie=5
```

Tous les exemples précédents utilisent du *SQL* standard, c'est-à-dire commun à tous les *SGBDR*. Néanmoins, certaines fonctionnalités, ou alors quand on va devoir encapsuler les requêtes dans du code procédural, ou au moins dans des fonctions *SQL*, impliquent un code final qui est spécifique à l'un d'eux.

L'encapsulation des requêtes dans des fonctions va permettre de les rendre paramétriques. De plus, on peut ainsi découper les actions à réaliser en modules cohérents. On peut en faire des morceaux de code génériques et faciliter la compréhension et la maintenance de l'application en augmentant la sécurité grâce au typage des paramètres et au contrôle des droits des utilisateurs. Le choix entre fonction procédurale ou fonction *SQL* pure dépend de la complexité de la logique algorithmique nécessaire à l'encadrement de la ou des requêtes dans la fonction. Pour autant, tant que faire ce peut, on essaiera de faire des fonctions en pur *SQL*. Vu que l'on vise à utiliser *PostgreSQL*, les exemples de codes suivants lui seront donc dédiés, mais on peut toujours trouver des solutions équivalentes pour les autres *SGBDR*.

PostgreSQL utilise le système des schémas pour organiser les objets (tables, fonctions, types, etc.) dans une base de données, on peut les voir comme des répertoires. Cela a aussi pour but d'éviter le télescopage de noms entres objets du même type et de permettre d'organiser la sécurité par schémas entiers.

Une fonction va comporter différentes informations autour du code :

- La liste des paramètres passés et leur typage. Dans le code, « *$1* » représente le premier, « *$2* » le deuxième, etc. On peut aussi utiliser le nom du paramètre mais il est possible de ne passer que le typage, ce qui est généralement fait, et pas de nom donc seul l'écriture « *$n* », avec *n* le numéro d'ordre du paramètre, est garantie. Attention, *PostgreSQL* supporte la *surcharge* donc deux fonctions avec un même nom et une liste de paramètres différents représentent deux signatures différentes donc deux fonctions différentes.

- La sécurité, qui dit avec les droits de quel utilisateur de la base de données le code dans la fonction va s'exécuter. On choisira l'utilisateur qui a créé la fonction, pour nous un « *super utilisateur* ». Il ne faut pas confondre le droit d'exécuter une fonction dans un schéma d'une base (l'appel de la fonction) avec le droit d'exécution utilisé au sein de la fonction pour son code. PostgreSQL fait en effet cette distinction qui est très pratique pour avoir une politique de sécurité plus globale.
- Le typage de ce qui est éventuellement retourné.
- Le nom du langage de programmation utilisé.
- Divers paramétrages optionnels pour l'optimisation.

Si on reprend le dernier exemple pour créer une fonction « acceptions » sur un schéma « test », on obtient le script ci-dessous :

```
CREATE FUNCTION test.acceptions(int)
RETURNS setof text stable
AS $$
        SELECT se.content
        FROM signs si
        INNER JOIN senses se ON se.id=si.idsense
        WHERE si.idlexie=$1
$$ LANGUAGE sql  SECURITY definer ;
```

À noter que si l'on préfixe un nom de fonction avec un nom de schéma et si l'on ne préfixe pas les tables appelées dedans, alors elles sont supposées appartenir au même que celui de la fonction. Dans la pratique, il vaut mieux toujours préfixer les objets par le nom du schéma qui les contient.

13 L'optimisation au niveau du SQL

Une grande partie de l'optimisation se fait grâce à l'utilisation que peut faire le *SGBDR* des index sur les tables pour éviter de lire leurs lignes en intégralité (« *full scan* » en anglais). Un réglage des paramétrages de la mémoire partagée dédiée au stockage des index devra peut-être être réalisé pour permettre leur chargement intégral en mémoire.

Quand une fonction ne risque pas de modifier des tables (on ne parle pas ici de tables temporaires) on doit le dire à l'optimiseur. Cela lui permet de mieux planifier le plan d'exécution et peut drastiquement réduire le temps d'exécution final du traitement. On lui dit que la fonction est « *stable* » quand le résultat ne varie pas avec les mêmes paramètres et données dans les tables éventuellement lues, « *volatile* » quand le résultat peut changer et enfin « *immutable* » si le résultat ne dépend que des paramètres. Par exemple :

- Une fonction qui ne contient que des « SELECT » ou des curseurs sera donc à déclarer en « *stable* ».
- Une fonction qui contient des « INSERT », « UPDATE » ou « DELETE » sera donc en « *volatile* ».
- Une fonction qui fait un traitement ne dépendant que des paramètres en entrée (un calcul, un encodage, etc.) sera en « *immutable* ».

Il faut réduire la taille des jeux de données en entrée de jointures le plus tôt possible dans le plan d'exécution et jouer à bon escient sur l'utilisation de fonctions qu'on a créés et qui évitent certaines jointures.

14 La sécurité de la partie SQL de l'application

Elle se base sur l'utilisation d'un ou plusieurs schémas où l'on met seulement les différents *API* de fonctions que l'on veut exposer (maintenance ou administration, application) et dans d'autres les tables et fonctions utilisées par les *API*.

Sur les schémas des *API* on ne permet la connexion depuis l'extérieur que par un utilisateur qui n'a que deux droits :

- « *SELECT* » car c'est l'action minimum qu'il peut posséder pour accéder à une fonction
- « *EXECUTE* » pour qu'il puisse appeler des fonctions

Sur tous les schémas on ne permet au super-administrateur de se connecter qu'en local. Si on veut l'utiliser depuis l'extérieur, il faudra le faire via un tunnel *SSH* en utilisant une clé *RSA*, ce qui implique une liaison vraiment très sécurisée.

Toutes les fonctions seront créées avec « **SECURITY** *definer* » garantissant que le contenu de la fonction s'exécutera indépendamment de l'utilisateur qui l'appelle.

Si une identification doit être réalisée pour garantir des opérations d'administration du dictionnaire

(via une interface web et non via un « *super administrateur* » de *PostgreSQL*), un jeton devra être passé aux fonctions de l'*API* concernées une fois que l'utilisateur de l'application (pas de *PostgreSQL*) se sera authentifié. Le jeton pourra même être changé à chaque action si on est en mode paranoïaque.

15 Conclusions

On voit que le nombre d'opportunités qu'offre ce genre de structuration des données est immense. De plus, l'export des données peut facilement s'organiser en tout ou partie et si l'on doit l'adapter à un format spécifique, il suffit de le faire grâce à des requêtes que l'on créera en conséquence. Se constituer des jeux de données particuliers à partir de ce genre de base est particulièrement pratique.

Cette organisation des tables peut être refondue grâce au partitionnement de celles qui sont spécifiques aux nœuds de données pour les mettre sur un serveur unique mais on devra alors limiter le nombre de langues pouvant être prises en charge, une ou deux dizaines maximum, serait certainement une limite raisonnable, ce qui permettra alors une intégration comme cœur d'un logiciel de *TAO*, de dictionnaire ou toute autre application en stand-alone.

L'approche actuelle n'est plus forcément d'avoir un outil sur un poste de travail mais il peut être proposé comme un service sur un réseau local ou le Net (modèle « *SaaS* » qui est l'acronyme de « *Software as a Service* » en anglais). L'approche que nous proposons va tout à fait dans ce sens voie mais peut donc aussi être transposée dans des solutions logicielles plus classiques. Le modèle de données est très ouvert et peut traiter toute langue réelle ou même inventée soit dans sa forme d'écriture native, soit dans n'importe quelle translittération si aucune n'est représentée grâce à l'*UTF-8* (donc d'après le standard de l'Unicode).

Le développement des principes linguistiques et technologiques présentés dans cet article est actuellement en cours de mise en œuvre dans le cadre d'un projet associatif mené par *APLLOD* (acronyme de « Association pour la Promotion des Langues via la Lexicographie et l'Open Data »), qui a pour but principal de concevoir un dictionnaire multilingue numérique, tout en proposant l'accès à ses données en open data.