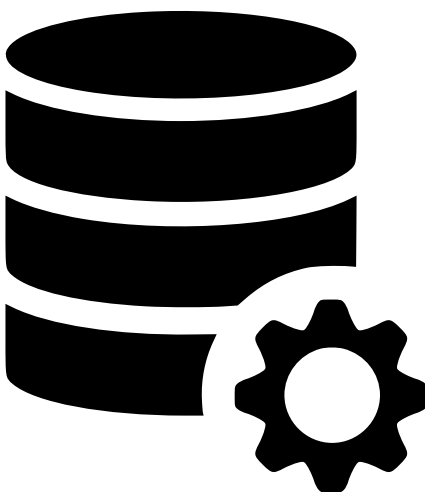


— not Only SQL —

0.99999999



Tout ce que vous avez toujours voulu savoir
sur les SGBD
sans jamais avoir osé le demander

Vincent LOZANO & Étienne GEORGES





Ce livre est publié sous licence **Art libre**
<http://artlibre.org>

À chaque réutilisation ou distribution, vous devez faire apparaître clairement les conditions contractuelles de mise à disposition de cette création. Chacune ces conditions de cette licence peut être levée si vous obtenez l'autorisation du titulaire des droits.

L'utilisateur quelque peu avachi devant son ordinateur à la page 2 provient de OpenClipart, licence Creative Commons Zero 1.0 (<https://creativecommons.org/publicdomain/zero/1.0/>).

Le symbole de la base de données de la couverture est régi par une licence de type CC BY 3.0 (<https://creativecommons.org/licenses/by/3.0/fr/>) et est téléchargeable ici :

<https://www.onlinewebfonts.com/icon/510380>

Version : Mars 2019



Sommaire

1	Ligne directrice	1
1.1	Vous avez dit « base de données »?	2
1.2	Informatiser un système d'information	4
1.3	Par quel bout on le prend, ce SGBD?	6
1.4	SQL, une séquelle de SEQUEL	7
1.5	Et un tableur, plutôt? non?	12
1.6	Les SGBD : un milieu plutôt « acid »	13
1.7	Le titre du livre et « NoSQL »?	14
1.8	Comment lire ce livre?	15
1.9	Kit de démarrage	17
2	Principes fondamentaux	21
2.1	Avant-propos	22
2.2	Codage des entiers	25
2.3	Nombres à virgule flottante	28
2.4	Codage des caractères	34
2.5	Performances d'un algorithme	40
2.6	Trier	44
2.7	Chercher	51
2.8	En résumé, à quoi ça sert, tout ça?	65
3	Modèle conceptuel des données	67
3.1	Qu'est-ce qu'une donnée?	68

3.2	Modèle conceptuel des données	68
3.3	Difficultés rencontrées	75
3.4	Micro-études de cas	80
3.5	Étude de cas : la discographie de FZ	84
4	Bâtir les données	95
4.1	Introduction	96
4.2	Ceci n'est pas une table	97
4.3	MCD → MR : « 1 à plusieurs »	104
4.4	« C'est pas faux »	109
4.5	Comment identifier un tuple	112
4.6	Un peu plus loin avec les contraintes	117
4.7	MCD → MR : « plusieurs à plusieurs »	124
4.8	Gestion des dates	135
4.9	Tas d'octets	141
4.10	Normalisation du modèle relationnel	142
5	Manipuler les données	149
5.1	Demander simplement	150
5.2	Modifier les données	171
5.3	Assembler : les jointures	177
5.4	Agréger	186
5.5	Combiner des requêtes	194
5.6	Utiliser des données externes	201
6	Stocker des traitements	207
6.1	Vues	208
6.2	Avant-propos sur les procédures stockées	214
6.3	Un nouveau langage	215
6.4	Procédure	218
6.5	Trigger	241
6.6	Utiliser un autre langage	253
6.7	Accès concurrents	258
6.8	Interaction avec un programme : PHP	273
7	Du côté de chez le DBA	281
7.1	Psql : la console à tout faire	282
7.2	Gestion des droits	288
7.3	Index	298
7.4	I18n : internationalisation	306
7.5	Tables « système »	308
7.6	Sauvegarde et restauration	314

7.7	Épilogue : la petite sirène	317
A	Notes de production	333
A.1	Prérequis	334
A.2	Les sources du manuel	334
A.3	Compilation	335
A.4	Imprimer	336
A.5	Nettoyage	337
A.6	Chapitrage	337
	Bibliographie	339
	Glossaire	341
	Index	349



- 1.1 Vous avez dit « base de données » ?
- 1.2 Informatiser un système d'information
- 1.3 Par quel bout on le prend, ce SGBD ?
- 1.4 SQL, une séquelle de SEQUEL
- 1.5 Et un tableur, plutôt ? non ?
- 1.6 Les SGBD : un milieu plutôt « acid »
- 1.7 Le titre du livre et « NoSQL » ?
- 1.8 Comment lire ce livre ?
- 1.9 Kit de démarrage

Ligne directrice

*It may well be that some composers do not believe in God.
All of them, however, believe in Bach.*

Bela BARTOK.

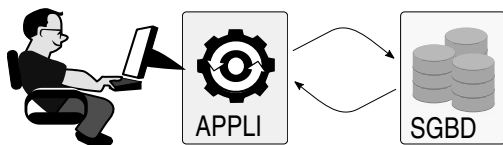
LA MOTIVATION première de ce livre est de restituer quelques 25 ans d'expérience dans le domaine des bases de données. Expérience que nous qualifierons d'atypique puisque vos serveurs ont tous deux un parcours quelque peu particulier vis-à-vis de ce qu'on appelle « l'informatique ». Tous deux diplômés de l'École nationale d'ingénieurs de Saint-Étienne (Énise) en 1993, nous recevons le grade de docteur (en 1998 thèse en image numérique pour Vincent LOZANO, en 1996, thèse en tribologie pour Étienne GEORGES).

Le premier suit la voie universitaire et devient maître de conférences en 1999 à l'Énise où il enseigne l'informatique et participe activement à la gestion du système d'information, pour s'y consacrer complètement en tant qu'ingénieur de recherche en 2017. Le deuxième embrasse une carrière d'ingénieur puis de consultant dans le domaine du Supply Chain (ce qu'on appelle logistique en France) et fonde sa propre entreprise ALOER en 2012, après de nombreuses pérégrinations dans des entreprises et groupes, d'activités et de tailles diverses.

Notre volonté est donc de vous faire part — de manière didactique — d'un savoir acquis tout au long d'expériences à la fois dans le secteur privé (consulting dans le domaine de Supply Chain) et dans le secteur public (gestion d'un établissement de l'enseignement supérieur). Ce livre n'est donc pas un cours magistral mais une présentation — que d'aucuns pourraient juger peu orthodoxe — de l'art des bases de données s'appuyant sur nos activités concrètes et variées autour de l'intégration, du développement, de la modélisation et de la gestion de systèmes d'information.

1.1 Vous avez dit « base de données » ?

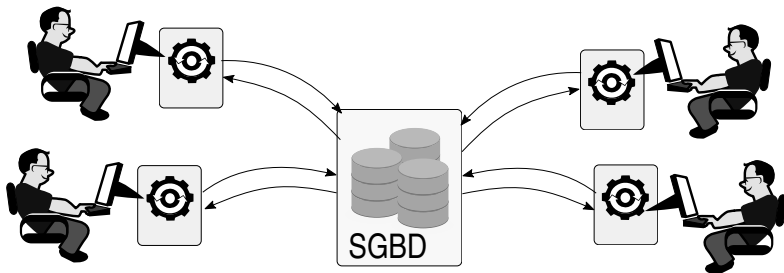
Ce manuel a pour objectif de guider un utilisateur novice et curieux dans le monde des bases de données. L'expression même « base de données » est souvent utilisée mais l'expérience montre que finalement peu d'utilisateurs en ont vues de près. La raison en est très simple : les systèmes de gestion de base de données fonctionnent sur le principe du client/serveur, par conséquent l'utilisateur final ne voit que le client, c'est-à-dire l'application dialoguant avec le SGBD :



En d'autres termes :

- l'utilisateur interagit toujours avec l'application (le client) ;
- l'application interagit avec le SGBD (le serveur).

Ajoutons qu'il y a toujours plusieurs clients pour un serveur, c'est-à-dire que *plusieurs applications* accèdent la plupart du temps *en même temps* au SGBD :



Les SGBD existant depuis presque 40 ans, nous avons tous été utilisateurs de ces systèmes, parfois quotidiennement et souvent sans même le savoir. Par exemple, aujourd'hui tous les sites marchands du Web ou les moteurs de recherche (au sens large du terme), utilisent une base de données. De la même manière, certains logiciels cachent souvent des bases de données derrière le nom de l'application : « système de gestion de données techniques », « ERP », « système d'information géographique (SIG) », « wiki », « blog »...

Cependant, même si les SGBD sont des outils puissants et adaptés à la gestion de données, notre monde d'il y a une vingtaine d'années n'était pas aussi « numérisé » qu'il l'est actuellement. Ceci

pourrait être l'explication qui pousse encore aujourd'hui certains utilisateurs à préférer un tableur pour gérer les données de leur structure publique ou privée ¹.

Nous vivons dans un monde où les informations que nous manipulons sont numériques, pour leur plus grand nombre. La tendance générale est à la dématérialisation : éviter le papier pour les documents, stocker la musique sur des supports numériques, partager des agendas, etc. C'est pourquoi, si dans les années 80, les SGBD étaient cantonnés à des domaines restreints (banques, ...), ils sont aujourd'hui incontournables même dans les petites entreprises ou les administrations de taille réduite.

Les systèmes de gestion de base des données que vous serez amenés à rencontrer sont :

Oracle : la « Rolls » des SGBD. C'est le logiciel dont la réputation est de supporter un grand nombre d'utilisateurs et d'être capable de manipuler de gros volumes de données. C'est aussi un des premiers SGBD à avoir été commercialisé. À noter : la société Oracle propose à chaque utilisateur le libre téléchargement de ses produits (qui sont multi-plateforme). À moins d'utiliser le logiciel à des fins personnelles ou pour l'enseignement, l'utilisateur devra se déclarer et se verra facturé en fonction de son usage (nombre d'utilisateurs, nombre de processeurs, nombre de serveurs, etc.);

Sql Server : le SGBD de la société Microsoft qui joue dans la même cour qu'Oracle en termes de fonctionnalités. Ne peut être utilisé que sur une machine tournant sur un système d'exploitation Windows;

Access : le petit frère Microsoft de Sql Server, est connu pour supporter de petites bases mais a la particularité de comprendre également un environnement graphique pour la modélisation et pour le développement d'applications. Ceci le rend accessible au néophyte même si la facilité apportée par l'interface ne remplace pas les connaissances en modélisation. Ces deux logiciels sont régis par des licences « propriétaires »;

MySql : un des premiers SGBD libres, racheté depuis par la société Sun, elle-même rachetée par Oracle; les premières versions avaient pour vocation d'être adossées à de petits sites

1. Un des objectifs de ce manuel est de convaincre que les tableurs ne sont pas une bonne solution pour gérer de manière pérenne les données d'une entreprise ou d'une administration.

web et ne disposaient pas de toutes les fonctionnalités attendues d'un SGBD ;

PostgreSQL : SGBD libre qui se targuait à une époque d'être un *Oracle Killer*. Il a effectivement beaucoup de fonctionnalités communes avec Oracle et est sans doute un des SGBD libres les plus aboutis. Ces deux logiciels sont régis par une licence libre.



Les auteurs du présent manuel ont tous deux une expérience de plusieurs années dans le premier (Oracle) et le dernier (PostgreSQL). Dans le contexte des logiciels libres du projet Framabook, nous ferons l'hypothèse que vous aurez installé sur votre ordinateur une version du dernier ².

1

1.2 Informatiser un système d'information

1.2.1 C'est quoi d'abord un SI ?

Le terme SI recouvre plusieurs notions mais qui tournent toutes autour du stockage, de la transformation et de la distribution d'information. En entreprise, le SI désigne l'ensemble des informations et leurs traitements permettant de mettre en œuvre les processus de la société. Le traitement peut consister en une circulation d'un papier dans un atelier : comme une étiquette à code-barres ou fiche suiveuse qui suit et contient les informations lors de la fabrication d'un lot de production. Ou alors le transfert par voie électronique d'un bon de commande ou la réponse par transmission électronique d'une facture dans un format prédéterminé. On parle alors d'échanges de données informatisées (ou EDI, même acronyme pour la version anglaise, Electronic Data Interchange).

Plus restrictivement en informatique, on emploie aussi SI pour désigner l'ensemble des logiciels et bases de données constituant le support de ces traitements. Mais les informaticiens d'entreprises, en manque de reconnaissance, ont très vite compris l'intérêt de passer de DSI (Directeur des Systèmes Informatiques) à SI (Directeur du Système d'Information).

Pour autant l'informatisation d'un système d'information n'est pas une tâche aisée car le SI couvre plusieurs activités :

1. l'analyse et la modélisation du système d'information ;

2. Une petite aide vous est fournie à la fin de ce chapitre introductif.

2. la conception d'une ou plusieurs bases de données ;
3. la création d'applications pour les besoins des utilisateurs.

1.2.2 Modéliser

Le premier point consiste à analyser le système que l'on veut automatiser partiellement ou totalement pour en tirer une modélisation c'est-à-dire aussi une simplification. Cette analyse est à mener avec les spécialistes du système étudié : ce sont eux qui fourniront les réponses aux diverses interrogations qui surgiront lors de l'étude. En tout état de cause le résultat de l'analyse devra être exprimé à l'aide d'un formalisme — au même titre que la formulation d'une solution à un problème peut être exprimée sous la forme d'un algorithme.

Nous présentons dans cet ouvrage, en particulier au chapitre 3, les grands principes de la modélisation via le *formalisme entité/association*.

1.2.3 Concevoir la base

La deuxième partie — conception d'une ou plusieurs bases — se rapproche de l'implémentation dans le cadre de la programmation : on choisit dans ce cas un langage pour implanter l'algorithme sur une machine. Ce qui est présenté dans ce manuel est SQL, le langage permettant de manipuler les *relations* du modèle relationnel. Il faut noter que tous les objets — en particulier les tables ou relations, mais nous verrons qu'il y en a d'autres — peuvent être manipulés via SQL.

Au stade de la conception de la base on pourra utiliser tous les outils (si possible standard) du logiciel utilisé. En particulier les SGBD dignes de ce nom proposent des « garde-fous » (appelés contraintes d'intégrité) participant à la sécurité et au maintien de la cohérence des données. Nous verrons également que ces logiciels proposent généralement la possibilité de stocker des traitements pouvant être déclenchés implicitement (par ex. lors d'un effacement ou d'une mise-à-jour) ou explicitement (par ex. via une application à la suite d'une action de l'utilisateur).

On peut alors voir le SGBD comme une coquille protégeant l'accès aux données — à l'instar du système d'exploitation garantissant la sécurité des ressources en limitant l'accès à celles-ci via des services limités — mais permettant aussi d'en faciliter l'accès sous certaines conditions.

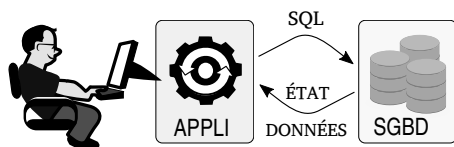
1.2.4 Développer une application

La troisième partie – le développement d’application — consiste à créer, à partir d’un langage choisi, un programme « attaquant ³ » — dialoguant avec — le SGBD. Plusieurs technologies peuvent être envisagées pour arriver à cette fin. Cependant quel que soit le choix, le principe est toujours sensiblement le même : on fait appel à une ou plusieurs routines d’une bibliothèque permettant de se connecter, lancer une requête, récupérer les résultats ou l’état, fermer la connexion. Le présent manuel **ne traite pas du développement d’application**, même si quelques aspects techniques sont exposés à la toute fin du chapitre 6, qui présente la manière de stocker des traitements dans un SGBD.

1

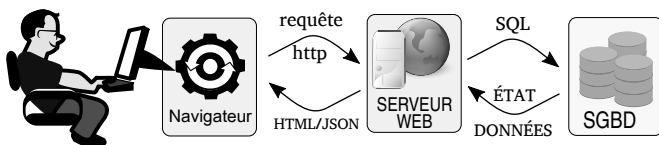
1.3 Par quel bout on le prend, ce SGBD ?

Pour être un peu plus précis sur la position du SGBD, agrémentons notre petite figure précédente :



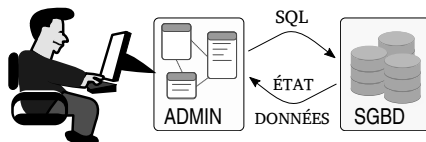
Les interactions de l'utilisateur avec l'application finissent par déclencher des *requêtes* à la base de données, qui doivent être exprimées dans le langage SQL. Le SGBD répondra toujours par un *état* (« tout va bien » ou « désolé mais je n'ai rien fait car quelque chose ne va pas dans ce que vous me demandez ») et des *données*, le cas échéant. Ces données seront alors traitées en retour par l'application, pour les présenter à l'utilisateur. Cette présentation consiste à transformer les données brutes en informations adaptées à l'application (texte, tableaux, graphiques, images, etc.).

Il n'est pas rare, à l'ère des applications web, d'avoir un étage de plus entre l'application et le SGBD :



3. Le jargon informatique est parfois un peu violent...

Comme indiqué précédemment, nous ne parlons pas dans cet ouvrage des joies du développement d'applications interagissant avec un SGBD. Vous serez par contre amenés à dialoguer avec lui via des applications spécifiques nommées *consoles* ou *interfaces d'administration* :



Ces interfaces vous permettent, dans leur mouture la plus rudimentaire (les consoles), d'envoyer interactivement des requêtes SQL au SGBD. Dans leur version plus évoluée, vous pourrez disposer d'interfaces graphiques qui, à partir d'actions sur des objets graphiques (cases à cocher, listes déroulantes, etc.), *fabriqueront des requêtes SQL pour vous* et les enverront au SGBD.



« Les applications passent, la base de données reste ». C'est elle qui est intrinsèquement responsable de ce qui se passe avec ses données. Et comme sa modélisation représente un métier avec ses contraintes et ses règles, c'est bien la base qui doit garantir que l'on ne peut les enfreindre. Les règles métier sont donc modélisées quelle que soit l'application utilisée pour traiter les données.

1.4 SQL, une séquelle de SEQUEL

Quand tout a commencé...

Il serait facile de penser qu'à une date précise une personne a décidé de travailler sur le sujet SQL. Mais comme souvent, la genèse est difficile à dater, et beaucoup de gens y ont travaillé de manière simultanée avec des intérêts divers.

Cependant, quelques-unes des personnes qui ont travaillé à l'origine sur ce thème pour IBM se sont réunies en 1995 afin de célébrer la naissance du projet *System R* et ses dérivés *R**, *SQL/DS* et *DB2*, 20 ans après leur création. Cette réunion d'anciens a été transcrite sur le web et est disponible à l'adresse http://www.mcjones.org/System_R/SQL_Reunion_95/sqlr95.html et porte le titre de « *The 1995 SQL Reunion: People, Projects, and Politics* ». Nous en extrayons quelques moments clefs qui montrent les tergiversations et autres égarements qui ont amené à un produit relativement fini

approximativement 50 ans après. Cette histoire n'est pas l'histoire officielle de la création de SQL mais plutôt quelques notes sur son historique et sur les personnes qui y ont participé.

Lors de la préhistoire ⁴, donc vers 1968, il n'était pas encore question de SQL ni même de « *base de données* » mais plus de « *gestion des données* » et de « *systèmes de fichiers* ». Ce qui était d'importance car à l'époque les recherches de certains laboratoire IBM (notamment celui de San Jose, USA, CA), financées par les fonds du projet APPOLO de la NASA, tournaient autour de logiciels permettant la gestion de fichiers et de données après le développement du disque dur dans les années 1950 dans le même laboratoire.

En 1970, Ted Codd publie un article d'importance (Codd, 1970) ⁵ et commence à travailler sur son projet avec une équipe de trois personnes. En parallèle et depuis 1959 ⁶, d'autres projets beaucoup plus importants, auxquels participait IBM, étaient menés sur des thèmes proches : SIGFIDET, CODASYL (*Conference/Committee on Data Systems Languages*, qui amènera à la création du COBOL).

Toutes ces équipes, en plein bouillonnement d'idées, passaient beaucoup de temps à débattre du « meilleur » système. Pour se départager, certains chercheurs jouaient au « jeu de la question » qui consistait à effectuer la même recherche avec chacun de leur langage afin de voir lequel permettait d'obtenir le résultat avec le moins de complexité dans l'écriture de la question. Ce qui a amené l'équipe dans laquelle travaillait Don CHAMBERLAIN, vers 1972, à la création de SQUARE, un langage tentant d'unifier ceux créés par Codd sur l'algèbre relationnelle et le calcul relationnel.

À la même époque, à l'Université de Californie à Berkeley ⁷, une équipe travaillait sur un système d'information géographique permettant l'analyse de la planification urbaine dont le résultat s'appelait INGRES et dont sont issus Ingres et Sybase.

Pour participer à un congrès avec comité de lecture à Stockholm,

4. En informatique, les notions de temps géologiques ont peu d'importance mais on aime bien reprendre les noms. Cela permet de rapprocher le big-bang d'il y a quelques 13,8 milliards d'années avec l'origine des temps informatique au milieu du vingtième siècle de notre ère

5. L'année précédente il avait déjà écrit dans un rapport technique interne IBM (Codd, 1969) les concepts développés dans cet article.

6. <https://en.wikipedia.org/wiki/CODASYL>

7. Il est à noter que les relations entre les laboratoires d'IBM et universitaires de Berkeley, Cambridge, MIT étaient très consanguines mais conflictuelles : nombres de chercheurs y ont fait leurs études avant de rejoindre les différents sites d'IBM. Mais, malgré de nombreuses conférences, ils évitaient de se parler de peur de se faire piquer leurs idées.

Don CHAMBERLAIN et Ray BOYCE ont écrit en 1974 deux articles : l'un sur la modification de données avec leur langage SEQUEL/DML (Data Modification Language) et l'autre sur la définition de données SEQUEL/DDDL (Data Defintion Language). Cela permet de dater les débuts de SEQUEL (Structured English Query Language) issu du projet IBM *Phase Zéro*. Le premier article n'a pas été retenu par le comité de lecture mais les 24 pages du second se sont transformées avec le temps en plus de 1600 pages dans la norme ISO définissant le langage SQL actuel.

En 1975 lors de la crise pétrolière, un premier « client », le MIT, obtient des fonds et exprime le besoin d'avoir une base de données pour suivre les stocks disponibles de pétrole. Cette base de données fournie par IBM marque l'origine de *System R*. Elle possédait déjà index, verrous, journalisation et transactions concurrentes. Le projet *System R* était lui même séparé en deux sous groupes : RDS (Relational Data System) qui définissait le langage et RSS (Research Storage System) qui s'occupait de la gestion physique des informations. Les logiciels développés étaient nativement client/serveur car ils tournaient déjà sur des mainframes⁸ avec des consoles déportées ou des « mini-systèmes ».

À la même période, Larry ELLISON travaillait alors pour Ampex, une société sous contrat avec la CIA, afin de développer une base de données de nom de code *Oracle*. À partir de ce projet, il créa avec Bob MINER une société connue d'abord sous le nom de Software Development Laboratories (1977), avant de changer son nom en Relational Software Inc en 1978, puis en Oracle Systems Corp en 1982 et finalement en Oracle Corp en 1995.

Le succès de sa compagnie, qui vendait donc un système de base de données relationnelles, tient en la personnalité de Larry ELLISON et de ses idées sur le sujet, basées sur la lecture de l'article de Ted Codd de 1970. En 1975, il contacte Don CHAMBERLAIN d'IBM afin d'être sûr que son système est totalement compatible avec celui d'IBM jusque dans les messages d'erreurs. Il ne les obtiendra pas car on lui fait répondre que ce sont des informations propriétaires confidentielles.

Le fort développement d'Oracle tient aussi au fait que le logiciel d'origine était écrit en C, langage déjà facilement portable sur de nombreux systèmes d'exploitation et plateformes (mainframes, mini-ordinateur, ordinateur personnel).

8. Machines de l'ère jurassique supérieur de l'informatique auxquelles on accédait de manière déportée via des terminaux ou consoles.

En 1976, IBM a été obligé de changer le nom *SEQUEL* pour une raison juridique : le fabricant d'avions anglais Hawker Siddeley avait déposé le nom pour un prototype qui n'a jamais volé. D'où la réduction de l'acronyme uniquement à SQL. Mais déjà à cette époque de nombreuses utilisations de *System R* existaient dans des sociétés aussi diverses que les usines pharmaceutiques Upjohn ou chez des constructeurs aéronautiques tels que Pratt & Whitney, Boeing, ou en interne dans d'autres divisions IBM. Ces installations étaient appelées « études communes » (*joint studies*) car la distinction entre logiciel commercial et développement spécifique n'était pas aussi claire que la documentation de l'époque voulait le faire croire. Pour information, le code contenait alors quelques 80000 lignes (à comparer avec les millions de lignes de codes des logiciels actuels) et tenait dans une RAM de 1 mégaoctet. Durant la vie « commerciale » de *System R*, il y a eu 251 bugs relevés et 161 demandes d'améliorations, dont aucune n'a été développée : pas facile de faire accepter des demandes dans le déjà mastodonte IBM de l'époque.

Après de nombreux changements d'organisation chez IBM, l'évolution de *System R* est portée dans ce qui sera la première version commerciale de DB2 proposée sur le marché en 1978. DB2 était alors vendu en tant que système d'aide à la décision et pas du tout en tant que système transactionnel. Les premiers utilisateurs ne l'auront que vers 1981.

En 1981 et en dehors d'IBM et Oracle, d'autres sociétés travaillaient aussi sur des bases de données relationnelles : on peut citer Esvel qui a ensuite été rachetée par Hewlett-Packard en 1984 pour sortir ALLBASE. En 1984, la société Tandem produisait déjà un outil de requête ENFORM, et une gestion de fichiers appelé ENSCRIBE mais leur produit du futur nommé Rainbow, planifié pour sortir en 1987 et qui devait tout unifier et ajouter les fonctionnalités de traitements transactionnels en ligne OLTP (On Line Transaction Processing), n'est jamais sorti.

Pour en revenir à Oracle, l'évolution était très différente de celle d'IBM mais en tout point parallèle : si la structure du langage venait à l'origine d'IBM, le développement en était complètement différent. Même si le nombre de personnes à la conception était relativement comparable, tout Oracle était dédié à cette unique activité alors que *System R* et autres DB2 ne représentaient qu'une toute petite partie des activités d'IBM. Le logiciel commercial Oracle Release 2⁹ sorti bien avant celui d'IBM, était par essence moins lié à une architecture

9. Il n'y jamais eu de version commerciale numéro 1.

tout en étant plus proche car développé dans un langage proche de l'assembleur sur un ordinateur personnel ; et contrairement à IBM, ne venait pas de chez un constructeur de machines dotées de systèmes d'exploitations existants¹⁰.

Afin d'accélérer un peu le temps, on peut établir la frise suivante reportant quelques versions successives d'Oracle et les fonctionnalités créées ou incorporées :

- 1979** premier ORACLE parent / cousin illégitime de *System R*
- 1986** SQL défini par les normes ANSI et ISO
- 1989-92** ajout de contraintes, procédures stockées (Oracle 7)
- 1997** fonctions orientées Objet - multimédia (Oracle 8)
- 1999** Orientation internet - JVM (Oracle 8i)¹¹
- 2002** Datawarehouse global (Oracle 9i)
- 2003** Grid ready (Oracle 10g)
- 2005** Express Edition 11+1 Gb (Oracle XE)
- 2007** support natif Linux et Microsoft (Oracle 11g)
- 2010** Acquisition de Sun par Oracle
- 2012** Cloud OS (Oracle Solaris 11)

Bien d'autres systèmes de base de données sont nés durant cette période (cf. la liste ci-dessous¹²) mais raconter chacune de leur histoire sortirait bien évidemment du cadre de ce livre :

4e Dimension (4D) Microsoft Access OpenOffice Base DB2, Firebird, Visual FoxPro, HyperFileSQL, Informix, Ingres, MariaDB, MaxDB (anciennement SAP db), Microsoft SQL Server, Mimer, MongoDB, MySQL, NoSQL, Ocelot, Oracle, Paradox, PostgreSQL, SQLite, SQL/MM, Sybase, Teradata.

Juste quelques mots sur PostgreSQL : créé en 1986 par Michael STONEBRAKER¹³, alors professeur à l'Université de Berkeley (et oui, encore UCB), c'est un digne successeur Open Source de Ingres. Son nom « *Postgres* » vient d'ailleurs d'un jeu de mots sur son prédécesseur traduisible en Français par « Après Ingres ». Ingres puis

10. IBM veut dire International Business Machines (créé en 1924) et fournissait à l'origine une machine (la *tabulating machine*) destinée à assister le recensement de 1890.

11. Pour mémoire, vers la fin des années 1990, Oracle était qualifié de « dinosaure » par Informix (racheté par IBM en 2001) car en retard sur les fonctionnalités issues de d'Internet.

12. Dans laquelle se sont glissés quelques intrus « non relationnels ».

13. Celui-ci est devenu directeur technique (CTO) d'Informix par la suite.

Postgres, développés de 1977 à 1995 comme exercices d'application des technologies des relations entre objets, ont quitté le monde universitaire lorsque Ingres a été racheté par Computer Associates en 1986 et, une dizaine d'années plus tard, en 1995, lorsque Postgres a été ouvert au monde Open Source et n'a plus été maintenu à UCB mais uniquement par un ensemble de développeurs indépendants. Depuis cette date et dès lors nommé PostgreSQL, il s'est taillé une réputation de système à toute épreuve, ayant un nombre de fonctionnalités équivalent aux systèmes commerciaux.

1

1.5 Et un tableur, plutôt ? non ?

Nous, auteurs de ce manuel, bien qu'ayant des activités professionnelles dans des secteurs et contextes différents, avons tous deux rencontré un nombre incalculable de cas où le tableur¹⁴ est érigé en panacée numérique pour la gestion des informations.

Un jour, lorsque je¹⁵ tentais de présenter l'activité de *modélisation* des données dans le but de structurer une base et que pour illustrer mon propos je montrais un « fichier Excel » — provenant d'une filiale d'un grand groupe du génie civil gérant la rénovation d'appartements — qui contenait tout un tas de cases coloriées, avec des jolies bordures, un étudiant eut une illumination :

« En fait monsieur, ça (en désignant le fichier Excel),
ce sont des informations *numérisées*,
et nous on veut *modéliser*. »

Remarque particulièrement judicieuse, que nous résumons en :

Numériser n'est pas modéliser.

En d'autres termes, remplir des cases, les colorier, mettre des bordures, du gras, mais n'avoir aucun *contrôle de validité*, ni de sécurité vis à vis de l'*effacement*, n'avoir aucun *lien* entre les données, ne pas pouvoir les *exploiter automatiquement* pour un traitement, *n'est pas une modélisation*, c'est juste un *stockage dans un fichier*. De plus :

- un tableur ne supporte pas de traiter un grand volume de données, il n'est pas conçu pour optimiser les différents calculs ;
- un tableur n'est pas conçu pour gérer les accès simultanés aux données, car il délègue cela au système d'exploitation.

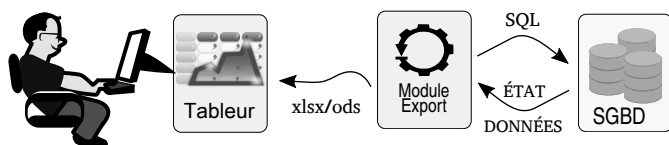
14. Excel pour ne pas le nommer.

15. Vincent LOZANO

Mais ne nous faites pas dire ce que nous n'avons pas dit : un tableur est utile dans bien des cas — et reste l'outil le plus accessible et efficace pour un utilisateur non informaticien — pour des activités spécifiques, dans lesquelles un SGBD n'excelle pas¹⁶ :

- faire des statistiques sur un nombre réduit de données ;
- présenter des données sous forme de graphiques ;
- calculer toutes sortes d'indicateurs ;
- ...

C'est d'ailleurs une pratique courante d'exporter des données du SGBD dans un format tableur. Les utilisateurs pourront alors se livrer à la construction de graphes et autres tableaux croisés dynamiques :



Dans le schéma ci-dessus, on imagine qu'à la demande ou périodiquement, on produit un fichier à partir des données du Sgbd. Il faut savoir qu'il est également possible de connecter directement une des feuilles de calcul du tableur au Sgbd et ainsi de l'alimenter en données sans passer par un fichier.

1.6 Les SGBD : un milieu plutôt « acid »

Les systèmes de gestion de bases de données relationnelles ont été conçus pour gérer des *transactions*, chacune de ces transactions étant composée d'opérations élémentaires. L'exemple classique est celui de la transaction bancaire nécessitant de débiter un compte pour en créditer un autre. Tous les systèmes de réservations de ressources (place de concert, achat dans un magasin en ligne, etc.) font également appel à des transactions. Ainsi, acheter une place de concert constituera une transaction pour le SGBD dans la mesure où cela demandera l'exécution de plusieurs opérations élémentaires.

L'histoire dit que c'est James GRAY¹⁷ qui a explicité **les mécanismes permettant de garantir qu'une transaction puisse s'exécuter de manière fiable sur un système informatique**. L'énoncé de ces propriétés constitue un acronyme¹⁸ (ACID):

16. Vous noterez la finesse de l'allusion.

17. https://fr.wikipedia.org/wiki/Propri%C3%A9t%C3%A9s_ACID

18. Dont les informaticiens sont friands.

Atomicté : une transaction est réalisée complètement ou pas du tout ;

Cohérence : une transaction ne peut basculer la base dans un état incohérent ;

Isolation : une transaction s'exécute comme si elle était seule sur le système ;

Durabilité : une fois une transaction validée, son effet perdure même si on coupe l'électricité.

Nous reviendrons en détail sur le concept de transaction, partiellement au paragraphe 5.2.4 page 176 puis en détail au chapitre 6 dans la section traitant des accès concurrents◀.

1

► § 6.7
p. 258

1.7 Le titre du livre et « NoSQL » ?

Les systèmes que nous allons étudier dans cet ouvrage sont de la famille des SGBD relationnels, ils ont été conçu pour la plupart pour gérer les données d'un seul établissement (entreprise ou structure publique). L'usage classique consiste alors à héberger les données — que l'on aura structurées très finement pour coller parfaitement au fonctionnement de l'établissement — sur un seul ordinateur.

Au moment de l'avènement des plates-formes « Web 2.0 », les sociétés qui ont émergé ont toutes préféré créer leurs propres systèmes de gestions de données, et ce pour différentes raisons :

1. les *volumes de données* traités dans le cadre d'application Internet dépassent très largement ceux traités pour gérer un seul établissement ;
2. la *structuration des données* est très faible (souvent limitée à la notion d'association clé/valeur) ou mouvante, évoluant avec le temps ;
3. les SGBD de l'époque ne supportaient pas la *scalabilité horizontale*, c'est-à-dire l'idée d'augmenter les performances du système en ajoutant une ou plusieurs machines ;
4. et enfin, la simplicité des traitements effectués sur ces nouvelles plates-formes Internet, n'exigeait pas nécessairement l'artillerie imposée par les propriétés ACID des SGBD relationnels¹⁹.

19. Certains ont même conjecturé et prouvé (cf. le théorème de BREWER) qu'il n'est pas possible de respecter les propriétés ACID sur un système distribué sur plusieurs machines.

Finalement, Google, Facebook, Twitter, LinkedIn, Baidu, Amazon, Microsoft et d'autres, ont créé des systèmes :

- distribués sur plusieurs machines situées dans des datacenters ;
- pouvant manipuler des volumes astronomiques de données ;
- dont la structuration et la variété des traitements restent limitées.



Ces systèmes, s'appuyant sur des paradigmes variés ont été nommés « noSQL » pour « not only SQL. » Il ne s'agit pas de l'acronyme d'un langage, ni d'un logiciel mais d'une nouvelle famille de logiciels (MongoDB, Cassandra, CouchDB, ...) ²⁰ mettant chacun en œuvre leur propre paradigme non relationnel (orienté graphe, orienté agrégat, sans schéma, etc.) adaptés à des applications ciblées.

Vous aurez sans doute compris, vu le titre du livre que vous lisez, que **nous ne vous parlerons pas plus avant de NoSQL...**

1.8 Comment lire ce livre ?

Les auteurs ont décidé de diviser en chapitres dont les grands thèmes sont :

Concepts fondamentaux : comment mémoriser les données sur une machine : nombres, caractères, etc. et comment réaliser le plus efficacement possible les opérations fondamentales (tri et recherche d'informations) ;

Modélisation des données : comment analyser un ensemble de données, le structurer et en faire un plan dans un formalisme compréhensible par tous ;

Bâtir les données : comment construire les données dans une base de données relationnelle ; dit autrement, à partir du plan dont il est question ci-dessus, passer à la « fabrication », c'est-à-dire la mise en œuvre dans une base de données ;

Manipuler les données : avec un langage de requête « vieux comme le monde » pour retrouver des informations dans la base, selon des critères fournis par l'utilisateur ;

Stocker des traitements : comment exploiter la partie « langage de programmation procédural » de SQL pour stocker dans la

20. <https://fr.wikipedia.org/wiki/NoSQL>

base des « petits programmes » capables d'exécuter des tâches imaginées par le concepteur ;

Du côté de chez le DBA : quelques petites choses à savoir pour installer, configurer, explorer le système de gestion de bases de données PostgreSQL. Et en bonus, une présentation des index et des mécanismes basiques de sauvegarde. En d'autres termes, la présentation de quelques tâches qui incombent au *database administrator* (DBA).

1



Les notes de ce style ont pour but d'attirer votre attention sur des concepts ou mécanismes importants.



Les notes de ce style ont pour objet d'entrer un peu plus avant dans le détail des points présentés.



Les notes de ce style vous indiqueront que ce qui suit n'est pas forcément facilement digérable en première lecture et qu'il peut être raisonnable de passer outre si vous n'avez pas encore maîtrisé les concepts présentés jusque-là dans le chapitre.



Vous devriez pouvoir exploiter dans n'importe quel Sgbd les requêtes SQL présentées dans ce livre. Malgré tout, les notes de ce style inspiré du logo du Sgbd PostgreSQL, auront pour objet de signaler que les informations exposées ne respectent pas le standard SQL, mais seront :

- soit uniquement exploitables avec PostgreSQL, le Sgbd choisi pour ce manuel ;
- soit des particularités du Sgbd Oracle utilisé par les auteurs pour leur activité professionnelle.

Une commande à lancer dans un terminal Unix, sera produit dans le document, comme ceci ²¹ :

```
ls -l *tex
```

Une commande SQL et son effet dans la console de PostgreSQL seront produit comme suit :

```
yahozna=> drop table de_multiplication;  
DROP TABLE  
yahozna=>
```

21. Sauf à quelques endroits dans cet introduction, par souci de clarté.



Si vous n'avez pas de version imprimée sous la main, sachez qu'il vous est possible, soit de récupérer les fichiers PDF qui vous conviennent (noir&blanc ou couleur) sur le site Framabook :

<http://framabook.org>

ou même les sources sur le GitLab de l'association :

```
git clone git@framagit.org:framabook/onlysql.git
```

qui vous rendront autonome pour produire votre propre version, couleur ou noir & blanc, à lire via un lecteur de pdf, ou à imprimer.

1

1.9 Kit de démarrage

1.9.1 Installation



Vous aurez besoin des droits d'administrateur (**root**) sur votre système pour effectuer les opérations décrites ci-dessous.

Sur un système Linux utilisant les paquets Debian (tel que Ubuntu ou Mint pour ne citer qu'eux), vous pouvez taper dans un terminal de commande :

```
sudo apt install postgresql
```

pour installer la partie serveur. Après l'installation de ce paquet un utilisateur système nommé **postgres** est créé. C'est sous son nom que PostgreSQL fait tourner le ou les processus qui attendent les connexions des clients. **postgres** est également le nom de l'utilisateur logique reconnu cette fois par le SGBD en tant qu'administrateur, c'est-à-dire ayant tous les droits sur toutes les bases. Puis :

```
sudo apt install postgresql-client
```

pour installer la partie client et en particulier la console SQL qui va être utilisée pour tester les différentes subtilités du langage présentées dans ce manuel.

1.9.2 Configuration minimale



Les opérations ci-dessous doivent être effectuées soit par le super utilisateur **root** soit par l'utilisateur **postgres**. Ce dernier n'a pas de mot de passe au moment de sa création, vous pouvez malgré tout vous faire passer pour lui à partir de **root** en tapant :

```
moi@txttxtx:~$ sudo bash
root@txttxtx:~# su - postgres
postgres@txttxtx:~$
```

Pour commencer à travailler on va créer :

- un *utilisateur logique* reconnu par PostgreSQL :
 postgres@txttxtx:~\$ createuser moi
- une *base de données* :
 postgres@txttxtx:~\$ createdb bacasable

1



Dans ce qui suit, nous supposons que votre login sur le système d'exploitation de la machine sur laquelle vous travaillez est constitué des trois lettres : moi.

Il faut ensuite autoriser l'utilisateur moi à se connecter sur la base bacasable. Ceci peut-être fait en éditant le fichier (x.y sera la version installée sur votre système, par exemple 9.1) :

```
/etc/postgresql/x.y/main/pg_hba.conf
```

en y ajoutant la ligne :

```
# Put your actual configuration here
# -----
# [...]
local moi bacasable trust
```

qui indique qu'on fait confiance à l'utilisateur moi pour se connecter sur la base bacasable en local (c'est-à-dire pas via le réseau) sans qu'il ait à saisir de mot de passe (c'est le sens de trust).

1.9.3 Psql : la console SQL ²²

Si tout s'est bien passé jusqu'ici, vous devriez être en mesure de lancer l'interface de base avec PostgreSQL grâce à la commande :

```
psql bacasable
```

qui vous accueillera avec un gentil :

```
bacasable=>
```

²². La console, c'est cette fenêtre noire d'aspect un peu rébarbatif mais finalement très ouverte sur les données et leur utilisation.

Cette console vous permet essentiellement de soumettre des requêtes au serveur qui les exécutera si ce que vous lui demandez n'enfreint pas les règles en vigueur (syntaxe, permission, intégrité, etc.). La soumission d'une requête SQL via la console se fait **en saisissant un point virgule après la requête**. Par exemple :

```
bacasable=> create table bidule(i int, t text);
```

Dans la mesure où il ne fait pas à proprement parler partie de la syntaxe de SQL, ce point virgule *n'apparaîtra dans aucun exemple de requête* dans ce manuel. Vous avez donc l'entière responsabilité de le saisir. En cas d'oubli, votre requête ne sera pas envoyée au serveur et restera dans un *buffer* (zone de mémoire tampon). Nous vous invitons à lire le paragraphe 7.1 page 282 pour prendre connaissance de toutes les subtilités de `psql`.

Enfin la console `psql` est également l'outil qui vous permettra d'inspecter les différents objets de la base que vous serez amenés à créer. Pour démarrer, deux commandes sont indispensables :

```
bacasable=> \d
```

qui liste (décrit) tous les objets de votre base. Et :

```
bacasable=> \d musicien
```

pour inspecter la structure de l'objet `musicien` (dont on suppose très vaguement ici qu'il s'agit d'une relation).

Remerciements

Un grand merci aux relecteurs de l'association Framabook (Mireille BERNEX, Barbara BOURDELLES et Jean-Bernard MARCON) pour leurs yeux de lynx capables de dépister des coquilles à plusieurs kilomètres à la ronde.

Remerciement spécial à Stéphane CROZAT pour son œil d'expert en système de gestion de base de données.

Un merci tout particulier à mes collègues de l'Énise pour leurs encouragements, en particulier Françoise FAUVIN pour sa relecture attentive et à Édouard VIDAL pour nous avoir fait le plaisir d'accepter de rédiger la 4^e de couverture.

Un grand merci à Christophe MASUTTI pour porter le projet Framabook et pour la confiance qu'il nous a accordée.

PS : le présent document a été généré à l'aide du magnifique et monstrueux ~~LaTeX~~ et avec les élégantes fontes KP dessinées par Christophe CAIGNAERT (<https://ctan.org/pkg/kpfonts>).

Bonne lecture et bon courage!

- 2.1 Avant-propos
- 2.2 Codage des entiers
- 2.3 Nombres à virgule flottante
- 2.4 Codage des caractères
- 2.5 Performances d'un algorithme
- 2.6 Trier
- 2.7 Chercher
- 2.8 En résumé, à quoi ça sert, tout ça ?

Principes fondamentaux

*I start in the middle of a sentence
and move both directions at once.*

John COLTRANE.

AVANT D'ABORDER les problématiques des bases de données à proprement parler, nous proposons dans ce chapitre une présentation de quelques notions fondamentales aujourd'hui intégrées dans tous les SGBD :

1. **Comment sont codées les informations** élémentaires : *quid* des techniques pour représenter des nombres, des chaînes de caractères, etc. C'est un sujet incontournable qui permet de comprendre toutes les subtilités autour de la *précision* des nombres — quelle erreur commet-on en stockant des nombres à virgule — et leur *étendue* — quel est le plus grand et le plus petit nombre utilisable. On découvrira également les joies de l'encodage d'un caractère aussi simple qu'un « É ».
2. **Comment sont optimisés les traitements** élémentaires : les logiciel de bases de données passent une bonne partie de leur temps à *chercher* et à *trier* des données. Nous verrons que ces deux traitements sont fortement dépendants l'un de l'autre et présentons ici les techniques classiques pour les réaliser de manière optimale. Cette partie de l'ouvrage est *indispensable pour comprendre* ce qu'est un ►index, mécanisme incontournable permettant, dans une certaine mesure, d'accélérer les requêtes d'extraction de données.

§ 7.3 ◀
p. 298

2.1 Avant-propos sur le codage

L'objectif de cette partie est de comprendre comment on peut représenter les *nombres* et les *caractères* sur un système informatique sous la forme d'une séquence de chiffres binaires (0 ou 1) correspondant à l'état des bascules électroniques composant la mémoire. Ces deux types de données (nombres et caractères) constituent la base de la plupart des autres, il est donc important d'en comprendre les mécanismes.

2.1.1 Base, nombre & chiffre

2

La suite de chapitre va faire appel intensivement à la base 2, nous proposons ici quelques rappels utiles sur ce qu'est et à quoi sert une base de numération. Les bases couramment utilisées en informatique sont :

Base 2 comportant deux chiffres : le 0 et le 1. Cette base est utilisée car c'est l'unité d'information élémentaire correspondant à l'état des bascules électroniques composant la mémoire ;

Base 16 dont les 16 chiffres sont : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. On verra qu'on utilise cette base car elle permet de représenter de manière rapide et compacte un nombre exprimé initialement en base 2¹. On appelle cette base la base *hexadécimale* ;

Base 10 parce qu'on compte sur les doigts de nos mains. Si l'être humain était né avec six doigts à chaque main, nous comptierions aujourd'hui naturellement en base 12 et les chiffres arabes comprendraient deux symboles de plus pour les 12 chiffres de la base.

Les deux chiffres de la base 2, que l'on appelle aussi chiffre binaire, ont donné en anglais le terme de *bit*, contraction de *binary digit* (chiffre binaire).

Un *nombre*, qui exprime une quantité, permet de *dénombrer*. Les chiffres qui composent un nombre sont des symboles qui permettent de désigner une quantité particulière dans une base donnée. On utilisera dans ce qui suit la notation indicée pour spécifier la base qu'on utilise. Ainsi ici (figure 2.1a) il y a *treize*, ou encore 13_{10} bâtons.

1. On utilise parfois la base 8 (dite *octale*) pour les mêmes raisons.

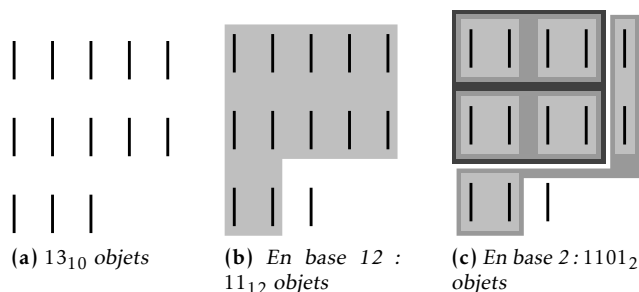


FIGURE 2.1 – Treize objets dans différentes bases

2

Si l'on veut écrire ce nombre treize en base 12, à chaque fois que l'on rencontre 12 bâtons, on crée une « dizaine² ». On a donc ici une « dizaine » et une unité, il y a donc 11_{12} bâtons (figure 2.1b).

En base 2, on fait des « dizaines » à chaque fois que l'on regroupe $2 = 2^1$ bâtonnets, des « centaines » à chaque fois que l'on groupe 2 dizaines (soit $4 = 2^2$ bâtonnets), etc. Ici, on a 1 « millier », 1 « centaine », 0 « dizaine » et 1 unité. Il y donc 1101_2 bâtonnets (figure 2.1c).

Finalement :

$$13_{10} = 11_{12} = 1101_2 = \dots$$

Il y a donc *plusieurs manières d'écrire le même nombre*, avec des symboles différents selon la base utilisée.

2.1.2 Conversion d'une base B vers la base 10

Remarquons tout d'abord que l'on peut écrire l'égalité suivante :

$$237_{10} = \underline{2} \times \boxed{10}^2 + \underline{3} \times \boxed{10}^1 + \underline{7} \times \boxed{10}^0$$

On retrouve la base élevée à des puissances croissantes, et les *chiffres* 2, 3 et 7 pondérant ces puissances. En base 2, on peut écrire une égalité analogue :

$$11101_2 = \underline{1} \times \boxed{2}^4 + \underline{1} \times \boxed{2}^3 + \underline{1} \times \boxed{2}^2 + \underline{0} \times \boxed{2}^1 + \underline{1} \times \boxed{2}^0 = 29_{10}$$

2. Les termes dizaine, centaine et millier s'entendent ici au sens de la *position* du chiffre.

De même en base 16 :

$$\begin{aligned}
 2AB_{16} &= \underline{2} \times \boxed{16}^2 + \underline{A}_{16} \times \boxed{16}^1 + \underline{B}_{16} \times \boxed{16}^0 \\
 &= 2 \times 16^2 + 10 \times 16^1 + 11 \times 16^0 \\
 &= 683_{10}
 \end{aligned}$$



De manière générale, et pour l'écrire de manière concise comme le font les mathématiciens, on obtient la valeur en base 10 d'un nombre exprimé dans une base B de la façon suivante :

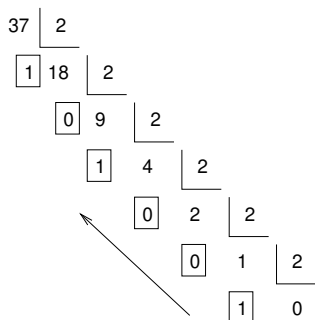
$$\text{valeur}_{10} = \sum_{i=0} c_i \times B^i$$

où c_i est le i^{e} chiffre du nombre en partant de la droite. La valeur de i commence à 0 et parcourt toutes les valeurs jusqu'au nombre de chiffres de valeur_{10}^3 .

2

2.1.3 Conversion de la base 10 vers une base B

Une méthode pour obtenir la représentation dans une base B d'un nombre exprimé en base 10 est la méthode dite des « divisions successives » Par exemple pour connaître la représentation binaire de 37_{10} on va le diviser par 2 ; on réitère l'opération avec le quotient, jusqu'à ce que ce quotient soit nul. Les différents restes des divisions forment le mot binaire.



On recompose le mot binaire en sélectionnant les restes des divisions en partant du dernier, ainsi on a :

$$37_{10} = 100101_2$$

De la même façon, pour convertir un nombre de la base 10 vers une base B , on va procéder à des divisions successives par B . Par exemple :

$$\begin{aligned}
 &25 \text{ divisé par } 16 : \text{« il y va 1 fois et reste 9 »} \\
 &\text{Donc } 25_{10} = 19_{16}
 \end{aligned}$$

3. Oui, moins 1, en effet.

2.1.4 Conversion de la base 2 vers la base 16

On peut exprimer un nombre binaire en base 16 (et inversement) par la méthode suivante⁴ :

1. regrouper les bits 4 par 4 en partant de la droite ;
2. combler avec des zéros à gauche si nécessaire ;
3. coder chaque bloc indépendamment les uns des autres en hexadécimal (un bloc de 4 bits peut prendre 16 valeurs de 0 à F).

Par exemple :

$$100101_2 = \boxed{00}10\boxed{0101}_2$$

et comme $0010_2 = 2_{16}$ et $0101_2 = 5_{16}$, on a $100101_2 = 25_{16}$. Autre exemple :

$$\left| \begin{array}{c} 1100 \\ 12_{10} \\ C_{16} \end{array} \right| \left| \begin{array}{c} 1010 \\ 10_{10} \\ A_{16} \end{array} \right|$$

On a donc $11001010_2 = CA_{16}$.



Cette technique nous sera particulièrement utile pour appréhender ce qui se cache derrière les mécanismes d'encodage et décodage des chaînes de caractères en UTF-8.

2.2 Codage des entiers

Dans la mémoire d'un ordinateur (mémoire centrale, registres du processeur, etc.), les entiers sont codés à partir de leur représentation en binaire et à l'aide d'un *nombre fini* de bits. Derrière l'*architecture 64bits* se cache par exemple l'idée que les entiers sont stockés à l'aide de 64 informations binaires. Le fait que ce nombre soit fini implique que les entiers représentés ont une *étendue*. C'est-à-dire qu'ils prendront des valeurs dans un intervalle de type $[\min, \max]$ (min pouvant être négatif)⁵. Du nombre fini de bits pour la représentation découle que les calculs effectués sur les entiers peuvent conduire à des *dépassements de capacité*, lorsqu'on obtient un résultat se situant en dehors de cet intervalle.

4. La méthode pour passer de la base 2 à la base 8 est similaire, en regroupant les bits par blocs de 3, ce qui donne 8 combinaisons possibles, correspondant à des valeurs de 0 à 7. Ainsi $11001010_2 = |011|001|010|_2 = 312_8$

5. Les mathématiciens disent que cet intervalle est inclus dans l'ensemble \mathbb{Z} , l'ensemble des entiers relatifs.

2.2.1 Entiers non signés

Les entiers non signés sont des entiers inclus dans \mathbb{N} , leur étendue est un intervalle de type $[0, \max]$. Supposons que l'on utilise N bits pour coder un entier :

$$\boxed{b_{N-1} \quad b_{N-2} \quad \dots \quad b_1 \quad b_0}$$

La valeur de ce nombre est :

$$\text{valeur}_{10} = \sum_{i=0}^{N-1} b_i \times 2^i$$

Par exemple, un entier codé sur 8 bits, c'est-à-dire sur un *octet*⁶ :

$$\boxed{1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1} = 129_{10}$$

La valeur maximale d'un entier non signé est atteinte lorsque tous les bits du nombre sont à 1, on a alors la valeur :

$$2^0 + 2^1 + \dots + 2^{N-2} + 2^{N-1}$$

On peut montrer que cette valeur, et donc l'étendue des entiers non signés codés sur N bits est :

$$\boxed{[0, 2^N - 1]}$$

Et, pour avoir un ordre de grandeur :

1 octet	8 bits	$[0, 255]$
2 octets	16 bits	$[0, 65\,535]$
4 octets	32 bits	$[0, 4\,294\,967\,295]$
8 octets	64 bits	$[0, 18\,446\,744\,073\,709\,551\,616]$

Remarquons ce qui se passe lorsqu'on ajoute deux entiers non signés sur 8 bits dont la somme dépasse l'étendue du codage (208_{10} et 92_{10} par exemple) :

$$\begin{array}{r}
 \begin{array}{cccccccc}
 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
 + & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
 \hline
 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0
 \end{array}
 \end{array}$$

la dernière retenue ne peut être stockée (il faudrait un 9^e bit) : on a un dépassement de capacité. L'unité centrale positionnera alors un indicateur d'état (parfois appelé *carry*⁷ *flag*) et le résultat du calcul sera 00101100_2 soit 44_{10} au lieu de 300_{10} ⁸...

6. En anglais *byte* que l'on prononce « baïte », à ne pas confondre avec *bit*.

7. Pour « retenue ».

8. C'est-à-dire 300 modulo 256 pour les plus observateurs d'entre vous.

2.2.2 Entiers signés

Pour coder les entiers signés (qui ont un intervalle de représentation $[\min, \max] \subset \mathbb{Z}$) on utilise un des bits du code pour stocker le signe :

s	b_{N-2}	...	b_1	b_0
-----	-----------	-----	-------	-------

Le $N - 1^{\text{e}}$ bit est ici noté s pour « bit de signe ». La valeur d'un tel entier est alors :

$$\text{valeur}_{10} = -s \times 2^{N-1} + \sum_{i=0}^{N-2} b_i \times 2^i$$

En reprenant l'exemple précédent, on a :

$$\boxed{1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1} = -1 \times 2^7 + 2^0 = -127_{10}$$

La valeur maximale de tels entiers est atteinte lorsque le bit de signe est 0 et les autres à 1, on a alors la valeur $2^{N-1} - 1$. La valeur minimale est quant à elle atteinte lorsque tous les bits sont à 0 sauf le bit de signe, la valeur est alors -2^{N-1} . Finalement l'étendue d'un entier signé sur N bits est :

$$\boxed{[-2^{N-1}, 2^{N-1} - 1]}$$

Et, pour information :

1 octet	8 bits	$[-128, 127]$
2 octets	16 bits	$[-32768, 32767]$
4 octets	32 bits	$[-2147483648, 2147483647]$
8 octets	64 bits	$[-9223372036854775808, 9223372036854775807]$



Un aspect intéressant de cette technique de codage des nombres négatifs (basée sur le complément à 2) — que nous choisissons de ne pas développer ici — est qu'elle est conçue pour faire fonctionner l'arithmétique binaire (les calculs) comme les entiers non signés.



Avec les entiers signés, on commence à percevoir la différence entre la représentation d'un nombre en base 2 et son codage.

Un exemple frappant est le nombre -1 en base 10 :

- il a pour représentation en base 2 : -1
- il a pour codage sur 8 bits : 11111111
- il a pour codage sur 16 bits : 1111111111111111

2.3 Nombres à virgule flottante

Nous examinons ici les nombres « avec des virgules » qu'on nomme en informatique les *nombres à virgule flottante* car la virgule semble flotter en des positions différentes dans la représentation du nombre. On notera que les anglophones emploient le terme de *floating point number* car c'est le point et non la virgule qui est utilisé comme séparateur des décimales.

2.3.1 Remarque préliminaire

2

Pour comprendre la suite, on peut déjà remarquer que lorsqu'on écrit un nombre à virgule, les chiffres après celle-ci correspondent à des puissances négatives de 10. Ainsi, par exemple :

$$12,345_{10} = \underline{1} \times \boxed{10}^1 + \underline{2} \times \boxed{10}^0 + \underline{3} \times \boxed{10}^{-1} + \underline{4} \times \boxed{10}^{-2} + \underline{5} \times \boxed{10}^{-3}$$

De manière analogue en base 2, les chiffres après la virgule correspondront à des puissances négatives de 2, c'est-à-dire : $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, etc.

$$10,011_2 = \underline{1} \times \boxed{2}^1 + \underline{0} \times \boxed{2}^0 + \underline{0} \times \boxed{2}^{-1} + \underline{1} \times \boxed{2}^{-2} + \underline{1} \times \boxed{2}^{-3}$$

De plus, nous savons — depuis notre plus tendre enfance — que décaler la virgule en base 10 correspond soit à diviser par 10 soit à multiplier par 10, selon le sens de déplacement. On peut faire la même manipulation en base 2, mais dans ce cas le déplacement correspondra à une division ou une multiplication par 2 :

$$\begin{aligned} 11000_2 &= 24_{10} \\ 1100_2 &= 12_{10} \\ 110_2 &= 6_{10} \\ 11_2 &= 3_{10} \end{aligned}$$

Ou encore :

$$\begin{aligned} 11_2 &= 3_{10} \\ 1,1_2 &= 1,5_{10} \\ 0,11_2 &= 0,75_{10} \\ 0,011_2 &= 0,375_{10} \end{aligned}$$

Pour finir, nous savons tous que les nombres en base 10 peuvent être écrits comme un produit avec une puissance de 10 :

$$12,345_{10} = 1,2345_{10} \times 10^1$$

donc, sans surprise, on peut écrire la même chose en base 2 :

$$10,011_2 = 1,0011_2 \times 2^1$$



On peut même dire qu'il est toujours⁹ possible d'écrire un nombre binaire non nul sous la forme :

$$1, \dots \times 2^x$$

Cette opération s'appelle la normalisation, on dit alors que le nombre en question est normalisé. On se servira de cette propriété pour le codage des nombres à virgule.

2.3.2 Norme IEEE-754

2

L'organisme de normalisation IEEE (Institute of Electrical and Electronics Engineers) a défini en 1985 puis en 2008 un standard pour coder plusieurs types de nombres avec virgule flottante :

1. simple précision (sur 4 octets, soit 32 bits);
2. double précision (sur 8 octets, soit 64 bits);
3. quadruple précision (sur 16 octets, soit 128 bits).

Ce codage — nommé IEEE 754 — est largement adopté, en particulier sur les micro-processeurs de nos ordinateurs. Il est basé sur le principe : signe / mantisse / exposant. Nous allons étudier le cas de la simple précision. On peut représenter les 32 bits d'un flottant IEEE 754 en simple précision¹⁰ comme suit :

s	e_7	...	e_0	m_{22}	m_{21}	...	m_0
---	-------	-----	-------	----------	----------	-----	-------

où :

- s est le bit de signe;
- $e_7 \dots e_0$ sont les 8 bits de l'exposant (qui s'entend ici comme une puissance de 2), ils forment un nombre entier E tel que :

$$E = \sum_{i=0}^{i=7} e_i 2^i = e_0 \times 2^0 + e_1 \times 2^1 + \dots + e_7 \times 2^7$$

9. Pour certains de ces nombres on aura bien sûr besoin d'une infinité de chiffres après la virgule.

10. Sachant que pour les autres précisions le mécanisme est rigoureusement identique.

- $m_{22}...m_0$ sont les 23 bits de la mantisse, on peut les imaginer comme les chiffres d'un nombre M tel que :

$$M = \sum_{i=22}^{i=0} m_i 2^{i-23} = m_{22} \times 2^{-1} + m_{21} \times 2^{-2} + \dots + m_0 \times 2^{-23}$$

La valeur du nombre ainsi constitué est alors :

$$(-1)^s \times (1 + M) \times 2^{E-127}$$

Bon, nous sentons que vous êtes perdus par tous ces Σ donc, nous vous proposons un exemple de décodage. Imaginons qu'au détour d'un de nos voyages dans la matrice, nous tombions sur la séquence de 32 bits suivante :

|0|10000000|101000000000000000000000|

Cette séquence, si on tente de la décoder en tant que 32 bits IEEE 754 simple précision, se décompose en :

- $s = 0$
- $E = 1 \times 2^7 + 0 \times 2^6 + \dots + 0 \times 2^0 = 128$
- $M = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0,5_{10} + 0,125_{10} = 0,625_{10}$

et le valeur du nombre en question est donc :

$$(-1)^0 \times (1 + 0,625) \times 2^{128-127} = 1,625 \times 2 = 3,25$$



Ici aussi, on pourra constater qu'il y a une différence entre la représentation d'un nombre en base 2 et son codage. Dans l'exemple précédent, le nombre 3,25 :

- a pour représentation en base 2 : $11,01_2$ (c'est-à-dire $3 + \frac{1}{4}$ suivez un peu, quoi...)
- est codé |0|10000000|101000000000000000000000| sur 32 bits selon la norme IEEE 754.



La norme IEEE-754 définit des séquences de bits particulières pour permettre de représenter des nombres particuliers :

Zéro : est défini en positionnant les bits de la mantisse et de l'exposant à 0 (peu importe la valeur du bit de signe) ;

?00000000000000000000000000000000

Infini : par convention si tous les bits de la mantisse sont à 0 et ceux de l'exposant à 1, le nombre représente l'infini qui peut alors être exploité dans certaines circonstances, comme résultat de calcul ;

?11111111000000000000000000000000

NaN pour Not a Number une séquence particulière (tous les bits de l'exposant à 1 et au moins un bit à 1 dans la mantisse) qui par convention définit un résultat de calcul incohérent d'un point de vue mathématique ($\sqrt{-2}$ ou $\log(-1)$ par exemple) ;

2.3.3 Précision et étendue

Les nombres à virgule flottante sont caractérisés par leur étendue et leur précision. On peut noter :

Valeur absolue maximale : obtenue lorsque tous les bits de la mantisse sont à 1 et que l'exposant atteint sa valeur maximale :

$$1,111...1_2 \times 2^{128} \approx 3 \times 10^{38}$$

Valeur minimale : obtenue avec la version « dénormalisée ¹¹ » :

$$0,000...1_2 \times 2^{-126} \approx 1.4 \times 10^{-45}$$

Précision donnée par la plus petite mantisse :

$$2^{-23} \approx 10^{-7}$$

ce qui signifie que les flottants en simple précision ont environ **7 chiffres significatifs**.



Dans le cas des nombres à virgule flottante il ne faut pas confondre la précision et l'étendue ; ainsi par exemple, même si on peut représenter des nombres autour de 10^{38} , qui est un nombre plutôt « grand » (composé de 38 chiffres en base 10, tout de même), on n'aura que 7 chiffres significatifs pour ces nombres. Cette précision est étendue à :

- environ 16 chiffres pour les flottants codés sur 64 bits ;
- environ 35 chiffres pour les flottants codés sur 128 bits.

2.3.4 Calcul d'un flottant IEEE-754

Pour comprendre ce qu'est la précision d'un flottant, il est intéressant de se pencher sur le codage lui-même : le calcul, à partir d'un nombre en base 10, de la représentation interne d'un flottant

11. Nombre particulier lâchement passé sous silence ici, permettant de s'approcher plus de zéro. Lorsque tous les bits de l'exposant sont à 0 et qu'au moins un bit de la mantisse est à 1, la valeur codée est $(-1)^s \times M \times 2^{-126}$

IEEE-754. Pour cela, on peut procéder en utilisant la méthode des « multiplications successives ». On analysera cette méthode sur deux exemples.

Soit le nombre $5,75_{10}$. On peut exprimer la partie entière en base 2 : $5_{10} = 101_2$. Pour la partie décimale 0,75, on procède comme suit :

$$\begin{array}{rclclcl}
 0,75 & \times & 2 & = & 1,5 & \geq 1 & \rightarrow \text{on retient} & \boxed{1} \\
 1,5 - 1 = 0,5 & \times & 2 & = & 1 & \geq 1 & \rightarrow \text{on retient} & \boxed{1} \\
 1 - 1 = 0 & \times & 2 & = & 0 & & \text{stop} &
 \end{array}$$

On peut donc écrire que $5,75_2 = 101,11_2 = 1,0111_2 \times 2^2$. On a donc tous les éléments pour définir s , E et M et on obtient le code suivant :

$$|0|10000001|0111000000000000000000|$$

Soit maintenant le nombre 0,3. Les multiplications successives donnent :

$$\begin{array}{rclclcl}
 0,3 & \times & 2 & = & 0,6 & < 1 & \rightarrow \text{on retient} & \boxed{0} \\
 0,6 & \times & 2 & = & 1,2 & \geq 1 & \rightarrow \text{on retient} & \boxed{1} \\
 1,2 - 1 = 0,2 & \times & 2 & = & 0,4 & < 1 & \rightarrow \text{on retient} & \boxed{0} \\
 0,4 & \times & 2 & = & 0,8 & < 1 & \rightarrow \text{on retient} & \boxed{0} \\
 0,8 & \times & 2 & = & 1,6 & \geq 1 & \rightarrow \text{on retient} & \boxed{1} \\
 1,6 - 1 = 0,6 & \times & 2 & = & 1,2 & \geq 1 & \rightarrow \text{on retient} & \boxed{1} \\
 \dots & & & & & & & \dots
 \end{array}$$

Le processus boucle donc :

$$0,3_{10} = 0,01001100110011001100110011001..._2$$

mais on ne dispose que de 23 bits pour la mantisse, on va donc utiliser le nombre :

$$1,00110011001100110011001 \times 2^{-2}$$

dont on pourra déduire le codage :

$$|0|01111101|00110011001100110011001|$$

et qui vaut très exactement $0,2999999821186065673828125_{10}$.



Par conséquent un nombre aussi « anodin » que 0,3 ne peut être représenté avec un tel système de codage, c'est-à-dire par exemple dans les registres flottants basés sur l'IEEE 754 des microprocesseurs, même les plus modernes !

2.3.5 Y-aurait-il pas moyen de calculer « juste »

Vous aurez raison de penser que pour certaines applications — par exemple celles qui manipulent des données financières — il est nécessaire de faire appel à des représentations exactes. On ne peut pas se permettre, dans une gestion financière et comptable, d'annoncer lors du bilan que les calculs sont à prendre avec uniquement 10 chiffres justes !

La technique utilisée pour stocker « autrement » un nombre à virgule, consiste à mémoriser le codage de chaque chiffre de ce nombre en binaire. Ce principe est connu sous le nom de *décimal codé binaire*. Dans ce système on codera le nombre 1234,56 grâce à la séquence :

1	2	3	4	5	6
0001	0010	0011	0100	0101	0110

Et le nombre 48151,62342 :

4	8	1	5	1	6	2	3	4	2
0100	1000	0001	0110	0001	0110	0010	0011	0100	0010

Chaque chiffre est donc représenté par une séquence de quatre bits. Ce principe induit quelques remarques :

1. le nombre 0.3 **pourra être stocké de manière exacte** : 0000 0011 ;
2. il faut trouver un codage pour le signe ;
3. il faut décider de l'emplacement implicite de la virgule ;
4. il prend plus de place que nécessaire (rappelez vous que 4 bits permettent de stocker les nombres de 0 à 15) ;
5. les calculs ne pourront pas être fait nativement par le processeur, par conséquent, toute l'**arithmétique sera plus lente qu'avec des flottants**.



Les systèmes de gestion de base de données proposent de stocker des nombres grâce à ces représentations exactes via un type nommé *numeric* ou *decimal*. Nous vous proposons au chapitre 6 consacré aux traitements stockés et en particulier au paragraphe 6.4.11 page 235, quelques routines de calcul mettant en évidence les différences notables (espace mémoire, performances de calculs, précision) entre flottant et nombre décimal codé en binaire.



Notons pour finir que si les caractéristiques des flottants découlent du nombre de bits utilisés pour les stocker, les types permettant une arithmétique exacte proposés par les Sgbd auront eux aussi des limitations imposées par l'espace mémoire qu'ils peuvent occuper au maximum. Cet espace sera généralement exprimé par le nombre de chiffres maximal avant et après la virgule.

2.4 Codage des caractères

2

Cette partie montre quelles sont les techniques utilisées pour stocker des caractères dans un système informatique. L'idée générale est qu'on finit toujours par stocker des nombres. Le principe du codage des caractères repose donc sur deux idées :

1. chaque caractère est représenté par un nombre : son *code*
2. ce nombre correspond à une entrée dans une *table* (ci-contre)

...	...
88	X
89	Y
90	Z
...	...
97	a
98	b
99	c
...	...

Il faut donc se convaincre que lorsque des programmes s'échangent des caractères, ils se transmettent des nombres. Pour être plus précis ces nombres sont des *octets*, unité élémentaire de transmission et de stockage. Pour éviter d'avoir à joindre la table des caractères à chaque transmission, il a fallu rapidement définir des *normes* internationales de codage des caractères.

Avant l'apparition du premier standard, chaque constructeur avait sa propre table. Fort heureusement à cette époque la tendance n'était pas à l'inter-opérabilité, et les systèmes informatiques n'étaient que très peu enclin voir incapables de s'échanger des informations. Donc ça n'était pas un drame si le mot « bonjour » écrit avec une séquence de codes (nombres) sur un système était décodé en « connard » sur un autre système, par un malheureux hasard des tables de caractères...

Mais suivons sans plus attendre l'histoire de cette normalisation...

2.4.1 La table ASCII (1964)

La première normalisation a été la table ASCII (pour American Standard Code for Information Interchange) en 1964 dont les caractéristiques sont les suivantes :

- codage des caractères sur 7 bits : la table comporte donc 128 caractères ; on échange des octets (8 bits) mais on conserve le 8^e bit pour contrôler que le transfert des informations se déroule correctement.
- le standard est conçu par des anglophones, par conséquent la question des caractères accentués est tout bonnement occultée.

Chaque système d'exploitation a ensuite proposé un codage des caractères sur 8 bits, correspondant à une table composée de la table ASCII et d'une extension qui lui était propre :

ASCII	0		ASCII	0		ASCII	0		ASCII	0		
	97	a		97	a		97	a		97	a	
	127			127			127			127		-----
UNIX	128		DOS	128		MAC	128		DOS2	128		
	180	È		180	é		180	ù		180	Ç	
	255			255			255			255		

La communication entre les différents systèmes devenait alors très difficile si on ne se limitait pas aux caractères de la table ASCII. En effet, comme le montre la figure ci-dessus, un message/fichier qui aurait contenu l'octet 180 aurait été interprété¹² comme le caractère È sur un système Unix, ù sur un système Mac, etc.

2.4.2 ISO 8859 ou le début de la normalisation

L'ISO (International Standard Organisation) a défini entre 1987 et 2001 une norme baptisée ISO 8859 permettant d'étendre la table ASCII pour les langues utilisant des caractères accentués, avec :

- un codage des caractères sur 8 bits (table de 256 caractères) ;
- une extension en fonction des zones du globe.

On a par exemple :

- ISO 8859-1 (Latin1) pour l'Europe de l'ouest : allemand, français, italien...
- ISO 8859-2 pour l'Europe de l'est : croate, polonais, serbe...

12. Ceci n'est pas la réalité mais un exemple montrant l'idée générale.

- ISO 8859-5 pour l'alphabet cyrillique : bulgare, russe...
- ISO 8859-6 qui contient les caractères arabes
- ISO 8859-15 (Latin9), proche du Latin1, mais avec en plus le « € » et le « œ », entre autres.

Mais il faut changer de table si on veut communiquer entre deux pays ne partageant pas la même extension. Et certains systèmes d'exploitation utilisent d'autres tables, comme MacRoman ou Windows-1252 (qui contiennent toutes deux tout ou partie de la table ISO 8859-1)...

2



Il peut être utile de savoir que les extensions incluses dans les tables ISO 8859 sont codées de 160 à 255 (en décimale). Il y a par conséquent un « no man's land » de 32 caractères (entre 128 et 159) que certaines tables, comme celle utilisée sous Windows, comblent avec des caractères. Il faut alors garder à l'esprit qu'un fichier qui contiendrait un caractère de cette zone, comme par exemple le '...' codé 85₁₆, ne sera pas parfaitement portable d'un système à l'autre.

2.4.3 Vers l'uniformisation : Unicode et ISO 10646

Depuis 1991, le consortium Unicode collabore avec l'ISO pour définir un codage des caractères *universel*, qui couvre l'ensemble des caractères du monde. La norme ISO 10646 définit un jeu universel de caractères, en anglais *Universal Character Set* (UCS). Il s'agit d'un ensemble de symboles, lettres, nombres, idéogrammes, logogrammes issus de langues, systèmes d'écriture, traditions du monde entier. Chaque caractère est identifié par un nom unique (un en anglais et un en français) et associé à un nombre entier positif appelé son point de code (ou position de code).



Unicode et ISO 10646 ne traitent que de caractères dit abstraits, par exemple « le a minuscule ». Il n'est pas question ici du dessin des caractères. Cette problématique relève du domaine des fontes.

Il peut être intéressant de noter que dans le projet Unicode, on distingue plusieurs couches :

Abstract character repertoire : ensemble des caractères nommés, en français et en anglais.

Coded character set : les *points de code* de chaque caractère, notés U+xxxx avec xxxx en hexadécimal. Quelques exemples :

Caractère	Nom en anglais	Code
A	latin capital letter A	U+0041
È	latin capital letter E with grave	U+00C8
€	euro sign	U+20AC
Œ	latin capital ligature OE	U+0152
ד	hebrew letter dalet	U+05D3

Character encoding form : le *code* proprement dit, c'est-à-dire la représentation physique du caractère, en mémoire, dans un fichier ou un message. Cette représentation n'est pas nécessairement une copie directe du point de code.

La dernière couche est finalement la plus délicate et c'est à cause d'elle que la migration n'est toujours pas achevée, alors que le projet a plus de vingt années d'existence. Le point dur est le suivant :

l'assertion « 1 caractère = 1 octet » n'est plus vraie.

Pour comprendre pourquoi ceci est un frein à une migration rapide, il faut s'imaginer qu'une quantité astronomique de programmes et de bibliothèques de programmation a été conçue en s'appuyant sur cette hypothèse. La remettre en cause demande à reprendre beaucoup de bibliothèques communément utilisées par un grand nombre de logiciels. Plusieurs solutions s'offrent naturellement à nous pour coder les plusieurs centaines de milliers de caractères qui constituent le corpus mondial :

- encoder tous les caractères sur 2 octets (65536 caractères possibles);
- idem sur 4 octets (environ 4 milliards de caractères encodables);
- trouver un système à taille variable, étant donné que les deux solutions précédentes ont le défaut de doubler ou quadrupler la taille des fichiers, et plus généralement, des informations stockées.

2.4.4 UTF-8 : l'encodage émergent

La transformation UTF-8 est la plus commune pour les applications Unix et Internet. Son codage de taille variable lui permet d'être en moyenne moins coûteuse en occupation mémoire. Mais cela ralentit nettement les traitements faisant intervenir des opérations complexes sur des chaînes (par exemple, extraction de sous-chaînes). Le principe peut se résumer en ces deux points :

- chaque entrée unicode U+xxxx est encodée grâce à une séquence composée de 1 à 4 octets ;
- chaque octet est affublé d'une séquence de bits de contrôle permettant d'aider et de contrôler le décodage (cf. plus bas).

Voyons maintenant le principe à partir de trois exemples. Pour commencer, un caractère inclus dans la table ASCII, le caractère 'X'. Dans ce cas, l'encodage est simplement l'ASCII :

Caractère	Code	Encodage	
		Binaire	Hexadécimal
X	U+0058	01011000	58

2

Dans le cas d'un caractère dont le point de code en binaire fait au moins 8 bits (et jusqu'à 11) on devra procéder de la manière suivante : soit par exemple le caractère é, dont le point de code est U+00E9. En binaire ceci correspond à la séquence :

$$E9_{16} = 11101001_2$$

On devra faire rentrer cette séquence de 8 bits dans les 11 bits disponibles (matérialisés ci-dessous par des '_') des deux octets suivants :

110_____ 10_____

En calant tout ce petit monde (je veux parler des bits correspondant au code E9) sur la droite on obtient :

110__11 10101001

En positionnant les bits inutilisés à 0, on obtient finalement :

Caractère	Code	Encodage UTF-8	
		Binaire	Hexadécimal
é	U+00E9	11000011 10101001	C3A9

De manière générale, l'encodage UTF-8 d'un caractère produira de 1 à 4 octets selon le principe suivant :

Taille	Représentation binaire	Points de code
1 octet	<u>0</u> _____	U+0000 à U+007F
2 octets	<u>110</u> _____ 10_____	U+0080 à U+07FF
3 octets	<u>1110</u> _____ <u>10</u> _____ 10_____	U+0800 à U+FFFF
4 octets	<u>11110</u> _____ <u>10</u> _____ <u>10</u> _____ <u>10</u> _____	à partir de U+10000

Un dernier pour la route, le caractère pour l'euro : son point de code est U+20AC soit 11 0000 1010 1100 en binaire. Dans ce cas on doit faire rentrer la séquence de bits dans :

1110 ---- 10 ----- 10 -----

On cale tout vers la droite, on a donc :

1110 __11 10 000010 10 101100

On obtient finalement :

Car.	Code	Code binaire	Hexa
€	U+20AC	11100010 10000010 10101100	E2 82 AC

2.4.5 J'Ã©cris en UTF-8, du moins j'essaie...

Un des syndromes intéressant est celui qui survient lorsqu'un programme pensant lire de l'ISO-8859, décode une séquence UTF-8. Ceci peut se produire lorsque le programme qui lit les données ne connaît pas l'encodage qui a été utilisé pour produire la séquence d'octets. Il faut en effet comprendre que lorsqu'un programme encode la chaîne « été » en UTF-8 il produit la séquence¹³ :

é	t	é
C3A9	74	C3A9

Et **rien dans cette séquence** ne précise qu'il s'agit de l'UTF-8. Si bien qu'un programme qui ne serait pas mis au courant pour x raisons, s'il décode cette séquence en pensant qu'il s'agit de ISO-8859-1, produirait un caractère par octet :

Ã	©	t	Ã	©
C3	A9	74	C3	A9


Car C3 et A9 sont respectivement les codes ISO-8859-1 des caractères « Ã » et « © ». Inversement un programme encodant la chaîne « été » en ISO-8859-1 produirait¹⁴ :



é	t	é
E9	74	E9

Et si par un malheureux hasard, c'est un décodeur UTF-8 qui tente de lire cette séquence, il agirait ainsi :

13. L'encodage en UTF-8 du « é » est expliqué un peu plus haut.

14. E9 est le code du caractère « é » dans la table ISO-8859-1.

1. humm, voyons le premier octet : E9, soit 11101001 en binaire
2. quelle séquence de contrôle contient cet octet ? 11101001
3. bien, il s'agit donc d'un encodage UTF-8 sur trois octets
4. je vais donc lire les deux octets suivants, voyons le premier : 74 soit 01110010 en binaire
5. j'ai bien l'impression qu'il y a un problème : 01110010. Je devrais trouver 10 comme séquence de contrôle, et j'ai 01...
6. puisque c'est comme ça, j'émetts un joli  indiquant que le flot d'octets n'est pas de l'UTF-8
7. et je passe à l'octet suivant, finalement le décodeur affichera :

	t	
E9	74	E9

2.5 Estimer les performances d'un algorithme

Pour le reste de ce chapitre nous allons présenter quelles sont les techniques connues pour rendre les plus efficaces possible les *traitements élémentaires* et pour cela nous étudierons deux grandes familles d'algorithmes : le *tri* et la *recherche*. Si on y réfléchit quelques instants, on prendra conscience du fait que de nombreux logiciels que nous utilisons — pour certains quotidiennement — passent beaucoup de temps à trier ou rechercher des données. Depuis les années 60, les mathématiciens, algorithmiciens, informaticiens se sont interrogés sur la manière la plus efficace de trier des données ou de rechercher une information parmi un ensemble. Les algorithmes mis au point sont aujourd'hui implémentés dans les SGBD si bien que lorsqu'on effectue une requête de type tri ou recherche parmi les données d'une base, on peut être certain que le traitement se fera dans un temps optimal. Nous proposons ici d'exposer quels sont ces algorithmes et comment on peut être sûr qu'ils répondent à la question efficacement en termes de temps de calcul.

Le terme utilisé en algorithmie pour qualifier les performances d'un algorithme, est *complexité*, et plus précisément la complexité en *temps*. Il ne s'agit pas d'estimer le fait qu'un algorithme soit complexe ou compliqué, mais d'estimer ses *performances*.

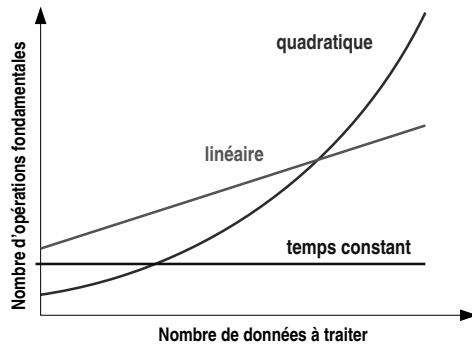


FIGURE 2.2 – Différentes complexités. Constante : l'algorithme nécessitera le même temps quel que soit le nombre d'opérations fondamentales N . Linéaire : le temps croît proportionnellement à N , quadratique selon le carré de N , etc.

2.5.1 Notion d'opération fondamentale

Pour caractériser un algorithme, on détermine une ou plusieurs *opérations fondamentales* en fonction du nombre de données. Par exemple :

- pour des algorithmes de tris : les opérations fondamentales sont les opérations de comparaison ;
- pour des algorithmes de multiplication de matrices : ce sont la multiplication et l'addition ;
- ...

On suppose donc, de manière implicite, que ce sont les opérations fondamentales qui déterminent principalement le temps d'exécution des algorithmes et que le temps d'exécution est fonction du nombre d'opérations fondamentales (voir la figure 2.2). Cette approche ne permet donc de comparer, entre-eux, que des algorithmes travaillant sur le même type de problème (on ne pourra comparer une méthode de tri avec une méthode de multiplication de matrices).



Il est important de comprendre ici que les courbes de complexité ne donnent pas une mesure du temps d'exécution, mais une idée de l'évolution de ce temps. On pense parfois à tort que l'étude des complexités est inutile étant donné la puissance des ordinateurs d'aujourd'hui. Ceci est bien entendu une idée fausse. Même s'il est vrai qu'il y a de grandes disparités en termes de puissance de calcul entre différentes machines, car :

- on demande aux ordinateurs de traiter de plus en plus de données,

par exemple : une image numérique d'un appareil photo datant de 1993 contenait 320×200 pixels, soit presque 200 fois moins qu'un appareil d'aujourd'hui¹⁵ ;

- quelle que soit la puissance, une complexité quadratique implique que vouloir traiter 10 fois plus de données demandera $10^2 = 100$ fois plus de temps.

Dans la suite de ce chapitre nous utiliserons les notations suivantes¹⁶ :

- $O(1)$: pour les complexités en temps constant : quel que soit le nombre de données à traiter, on considérera que l'algorithme exigera le même temps ;
- $O(N)$: la complexité *linéaire* : le temps augmente alors proportionnellement au nombre de données N ;
- $O(N^2)$: la complexité *quadratique* : on verra en particulier que dans le cadre du tri, cette complexité est considérée comme très mauvaise ;
- $O(\log(N))$: complexité *logarithmique*. On verra plus loin qu'on parle ici du logarithme en base 2.

Il est parfois difficile de donner un ordre de grandeur du temps nécessaire au déroulement d'un algorithme, dans tous les cas de figure. C'est pourquoi on calcule ou on estime généralement :

- la complexité **en moyenne**, c'est-à-dire le temps moyen que mettrait l'algorithme pour s'exécuter si on lui présentait des jeux de données correspondant à l'ensemble des configurations possibles.
- la complexité **dans le pire des cas**, c'est le temps mis par l'algorithme lorsqu'on lui présente un jeu de données le plus défavorable, c'est-à-dire le jeu de données pour lequel il doit effectuer le plus grand nombre d'opérations.

2.5.2 Ordre d'idée de l'évolution des temps

Pour avoir maintenant une idée de ce que représente une complexité linéaire $O(N)$ par rapport à une complexité logarithmique $O(\log_2(N))$, ou quadratique $O(N^2)$, supposons :

1. qu'on dispose d'un ordinateur capable d'effectuer 10 millions (10^8) d'opérations par seconde ;

15. Début du xxi^e siècle, vers 2018...

16. Cette notation n'est pas rigoureuse d'un point de vue mathématique, mais est utilisée ici sciemment par souci de clarté.

- 2. que nous ayons un problème pour lequel le traitement de 1000 données demande l'exécution de 100000 opérations, soit un millièème de secondes.

Progression avec un algorithme linéaire

Avec un algorithme linéaire, l'augmentation du nombre de données à traiter entraînera une augmentation dans le même rapport, du temps de calcul. Par exemple, en multipliant par 10 progressivement le nombre de données :

pour 10^3 données	on a	10^{-3} secondes de calcul
pour 10^4 données	on aura	10^{-2} secondes de calcul
pour 10^5 données	on aura	10^{-1} secondes de calcul
pour 10^6 données	on aura	1 seconde de calcul
...

Finalement, pour 10^9 (1 milliards de) données, on aura 1000 secondes, soit environ un quart d'heure.¹⁷

Progression avec un algorithme quadratique

Avec un algorithme quadratique le temps est augmenté proportionnellement au carré du nombre de données. Donc multiplier par 10 le volume de données à traiter conduit à un temps d'exécution multiplié par 10^2 soit 100. Par conséquent la progression sera :

10^3 données	10^{-3} secondes	
10^4 données	10^{-1} secondes	
10^5 données	10 secondes	
10^6 données	100 secondes	environ 2 minutes
10^7 données	200 minutes	soit environ 3h30
10^8 données	environ 350 heures	c.-à-d. environ 15 jours

Finalement, pour le traitement d'un milliard de données il faudrait donc 1500 jours soit environ 4 ans.

Progression avec un algorithme en $O(N \log N)$

On verra que l'algorithme de référence pour le tri, le « tri rapide » a une performance optimale¹⁸ en $O(N \log N)$. Pour ce type

¹⁷. À la louche.
¹⁸. Vous ne trouverez pas d'algorithme de tri avec une complexité meilleure que celle-ci.

d'algorithme, à chaque fois que le volume de données sera multiplié par 10, le temps de traitement sera multiplié par $10 \log 10$ soit environ 33,2. Si on reprend notre progression :

10^3 données	10^{-3} secondes	
10^4 données	$\frac{33}{1000}$ de secondes	
10^5 données	environ 1 seconde	
10^6 données	environ 30 secondes	
10^7 données	environ 900 secondes	environ $\frac{1}{4}$ d'heure
10^8 données	environ 33 quarts d'heure	environ 8 heures

Pour comparer avec les autres complexités, à 1 milliard de données le temps de calcul atteint 33×8 heures, soit environ 11 jours.



Ce qu'il faut retenir de tout ceci c'est que, entre un algorithme de complexité quadratique et un algorithme de complexité $O(N \log N)$, un utilisateur :

- ne verra pas de différence notable entre attendre 33 millièmes de secondes ou attendre un dixième de seconde ;
- pourra éventuellement accepter d'attendre 2 minutes pour trier 1 million d'éléments, mais appréciera peut-être d'attendre plutôt 30 secondes ;
- commencera sans doute à s'impatienter s'il faut attendre 3h30 alors que le collègue/concurrent attend, lui, pas plus d'un quart d'heure ;
- lorsqu'il aura à lancer un calcul intensif, n'hésitera pas une seconde entre un programme exigeant un temps de traitement de 11 jours et un autre nécessitant 4 ans.

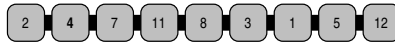
2.6 Trier

La problématique du tri consiste à mettre dans l'ordre un ensemble d'éléments. Il a été démontré qu'un algorithme ne peut pas trier N éléments avec une complexité meilleure que $O(N \log(N))$. Nous proposons dans cette partie de vous montrer ce qu'est un « mauvais » algorithme de tri, puis de présenter quelques-uns des « bons » algorithmes.

2.6.1 Tri par insertion : un mauvais élève

Le tri par insertion est la méthode que l'on utilise pour trier un jeu de cartes : on prend la première, et on insère chacune des suivantes à sa place. Dans le cadre de l'algorithmie, cela revient à

chercher à insérer une valeur parmi une séquence déjà triée. Considérons la séquence suivante :



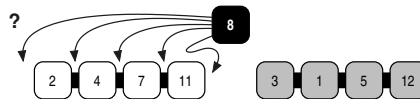
On commence par supposer que la première valeur est en place, et on traite la deuxième, c'est-à-dire qu'on se demande s'il faut la placer avant ou après la première :



À chaque étape de l'algorithme de tri par insertion, une partie de la séquence a été triée :



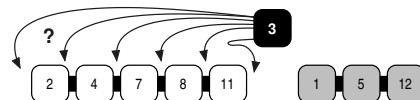
On se demande ensuite où l'on va insérer la valeur courante (ici 8) :



Une fois l'emplacement trouvé, on insère la valeur à sa place :



Et on reprend le même processus avec la valeur suivante (ici 3) à insérer :



Si on considère que les opérations fondamentales sont l'échange et la comparaison, on peut dire que l'algorithme du tri par insertion est en $O(N^2)$ dans le pire des cas et en moyenne. Pour avoir un ordre d'idée du temps qu'implique cette complexité, on peut examiner les diagrammes de la figure 2.3 page suivante. On notera que ces courbes s'approchent de paraboles comme prévu par la complexité théorique. Il est important de noter (cf. figure 2.3) que ce tri est performant pour des séquences « presque » triées. Cette propriété qui peut être utilisée dans certaines applications, est exploitée dans le tri Shell (cf. § 2.6.2). On retiendra que :

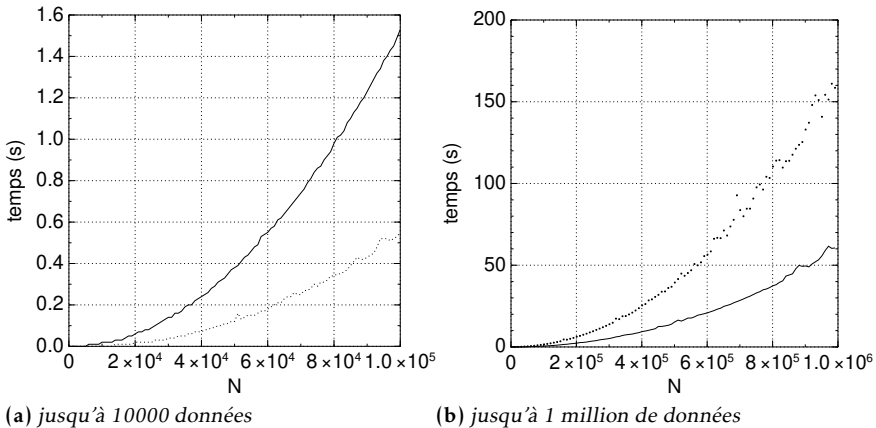


FIGURE 2.3 – Temps nécessaire pour le tri par insertion sur un processeur Intel Core i5®. En abscisse le nombre de données à traiter, en ordonnée le temps en secondes. Les courbes du bas montre le temps nécessaire pour trier des données « presque » triées.

La complexité du tri par insertion est de :

- $O(N^2)$ dans le pire des cas et en moyenne ;
- $O(N)$ pour des séquences quasiment triées.

2.6.2 Tri Shell

Le tri Shell doit son nom à D. L. SHELL qui l'a conçu en 1959. L'idée de ce tri repose sur la constatation que le tri par insertion est efficace si les données sont « presque triées. » SHELL a donc proposé d'effectuer un certain nombre de « pré-tris » rapides avant d'effectuer un tri par insertion.

À ce jour les avis sont partagés sur la complexité du tri Shell. Il y a en réalité deux *conjectures* pour la complexité en moyenne : $O(N \log_2 N)$ ou $O(N^{1.25})$. Il s'agit d'hypothèses qui n'ont pu être démontrées mathématiquement. Nous illustrons le fonctionnement du tri Shell en étudiant un exemple. Soit la séquence :



On trie d'abord les éléments pris tous les h_1 (ici 6) en ignorant tous les autres. On utilise un simple tri par insertion pour effectuer ce tri :



Même chose, en se décalant d'un cran à droite :



Encore un cran vers la droite : ici rien à faire la sous-séquence est triée.



On trie ensuite les éléments pris tous les h_2 (ici 3) en ignorant tous les autres :



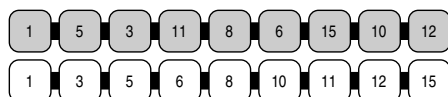
Même chose un cran à droite :



Même chose un autre cran à droite :



Enfin, on effectue un tri par insertion ($h_3 = 1$) :



La suite des h_i introduite par SHELL n'est pas celle présentée — par souci de simplicité — dans l'exemple précédent, mais :

..., 1093, 364, 121, 40, 13, 4, 1.

Ou comme diraient les mathématiciens, de façon récurrente :

$$\begin{cases} h_0 &= 1 \\ h_n &= 3h_{n-1} + 1 \end{cases}$$

2.6.3 Tri rapide

Le tri rapide ou *quick sort* a été conçu par C. A. R. HOARE en 1960. Il est plus difficile à implémenter que le tri Shell. On notera qu'il donne de bons résultats en moyenne et c'est pour cela qu'il est souvent implanté dans les bibliothèques de programmation (la librairie standard du langage C propose par exemple une fonction permettant de trier des données avec cet algorithme).

Complexité

Le tri rapide a une complexité en :

- en $O(N \log_2 N)$ en moyenne;
- en $O(N^2)$ dans le pire des cas.

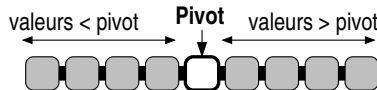
2

Principe

On choisit une valeur *pivot* permettant de *partitionner* la séquence en deux sous-séquences :

1. d'un côté les valeurs supérieures au pivot;
2. de l'autre les valeurs inférieures au pivot.

Une fois la séquence partitionnée de cette manière, on met la valeur pivot à sa place :

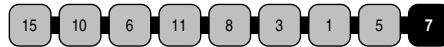


On réitère le processus sur chaque sous-séquence. Toute la subtilité de l'algorithme du tri rapide, réside dans un partitionnement rapide autour du pivot. Idéalement il faudrait choisir comme pivot *la médiane* de la séquence traitée, de manière à créer deux sous-séquences de tailles égales. Cependant le calcul de cette médiane *exigerait de trier la séquence*, ce que nous sommes précisément en train de faire! On prend donc un pivot « au hasard ». Voici un exemple de déroulement de l'algorithme du tri rapide. Soit la séquence :



On choisit comme pivot la dernière valeur de la séquence (ici la valeur 7¹⁹) :

19. Ouh le vilain qui a choisi comme par hasard, la médiane de la séquence...



On réalise le partitionnement de la séquence autour de la valeur pivot en effectuant deux *parcours*, l'un de l'avant dernière valeur vers le début de la séquence, l'autre de la première valeur vers la fin de la séquence.



Partitionnement lors du tri rapide

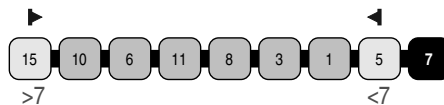
Voici comment se déroule le partitionnement. On démarre les deux parcours :

1. de la gauche vers la droite (symbolisé par ►) à la première valeur de la séquence;
2. de la droite vers la gauche (symbolisé par ◄) à l'avant-dernière valeur de la séquence.

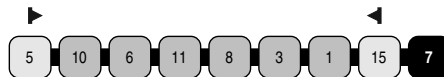
On stoppe les parcours :

- dès que ► rencontre une valeur supérieure au pivot (7);
- dès que ◄ rencontre une valeur inférieure au pivot (7).

Par conséquent dans notre exemple, ► s'arrête sur 15 qui est supérieur à 7, et ◄ sur 5 qui est inférieur à 7 :

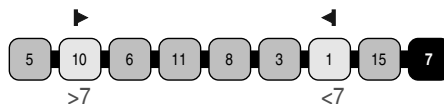


On échange les deux valeurs sur lesquelles les parcours se sont interrompus :

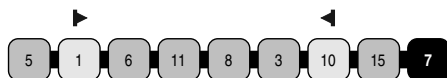


On reprend les deux parcours, qui s'arrêtent ensuite sur :

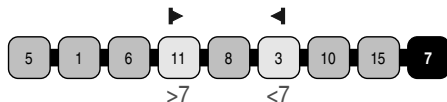
- la valeur 10 supérieure au pivot 7 pour ►;
- la valeur 1 inférieure au pivot 7 pour ◄;



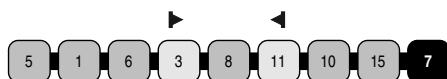
On échange alors 10 et 1 :



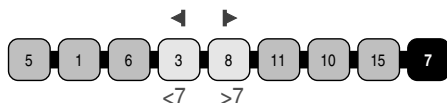
On reprend les parcours ► et ◄ qui s'interrompent respectivement sur les valeurs 11 et 3 :



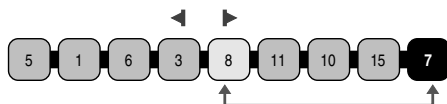
Valeurs qu'on échange :



On reprend ensuite les parcours qui finissent par se croiser :



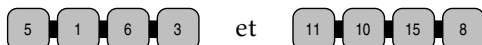
On finit le partitionnement en plaçant le pivot à sa place. Il suffit pour cela d'échanger la valeur pivot avec la valeur sur laquelle s'est arrêté le parcours ► :



On notera qu'après cette étape, l'élément contenant la valeur pivot (7) est en place pour ce qui concerne le tri :



Le schéma ci-dessus illustre également que le partitionnement a pour but de mettre chacune des valeurs de la séquence « du bon côté » par rapport à la valeur pivot. Il reste alors à trier chaque sous-partie. Ce qui est réalisé en *réitérant* le partitionnement sur les deux *sous-séquences* :



2.6.4 Pour en finir avec les algorithmes performants

Pour se rendre compte de l'efficacité des algorithmes performants de tri nous proposons d'étudier les courbes de la figure 2.4 page suivante qui montrent les temps nécessaires pour trier jusqu'à 10 millions de données sur notre machine test (Intel Core i5). On pourra faire quelques remarques :

1. le tri rapide porte bien son nom, puisqu'il apparaît comme le meilleur des algorithmes performants ;
2. il est intéressant de noter que le tri par insertion demandait environ 160 secondes pour traiter 1 million valeurs. La complexité de ce tri étant quadratique, le temps mis pour traiter 10 fois plus de valeurs sera donc 100 fois plus important. On atteindra donc 100×160 secondes, c'est-à-dire pratiquement quatre heures et demie ! Nous vous invitons à « méditer » sur le rapport entre ces quatre heures et la poignée de secondes dont ont besoin le tri Shell et le tri rapide, et ce *même si pour de petites quantité de données, un utilisateur ne verra pas de différence*.
3. dans le même ordre d'idée, il faut se persuader qu'il n'est pas possible de représenter sur un même graphique les trois tris (insertion, Shell et rapide), tellement le premier est peu performant ;
4. La figure montre également que les courbes de temps sont quasiment linéaires, ce qui est le cas de la fonction $f(x) = x \log_2(x) \dots$

2

2.7 Chercher

Dans cette partie nous nous intéresserons aux algorithmes fondamentaux de *recherche*. La recherche est une tâche réalisée extrêmement fréquemment par la plupart des logiciels. Par exemple, lorsque que votre logiciel de traitement de texte souligne en rouge un mot pour indiquer qu'il est mal orthographié, il vient d'effectuer une recherche parmi les quelques dizaines de milliers de mots de son dictionnaire.

Le problème de base que résolvent les algorithmes de recherche peut s'énoncer simplement : étant donné un ensemble de données, est-ce que la valeur \mathcal{V} est présente dans cet ensemble ? En examinant différentes méthodes de recherche nous verrons que la structure de

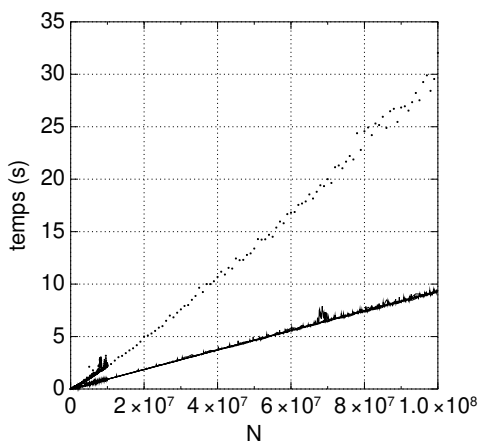


FIGURE 2.4 – Performances relatives du tri Shell (environ 30 secondes pour trier 10 millions de données sur notre machine test) et du tri rapide (environ 10 secondes pour la même quantité de données). Les artefacts s'expliquent par un processus système lancé pendant les tests.

données sous-jacente, c'est-à-dire l'*organisation* des données, permet d'effectuer des recherches de façon plus ou moins efficace. Comme lorsque vous cherchez un livre dans votre bibliothèque, vous mettrez énormément de temps à trouver un exemplaire si vos livres sont rangés sans aucune logique et vous mettrez rapidement la main sur un cd²⁰ de Frank ZAPPA si vous avez classé vos disques par ordre alphabétique des auteurs.

2.7.1 Chercher dans une séquence



On suppose ici que la structure de donnée « séquence » est une collection homogène de données à laquelle on accède via un numéro de case. Cet accès est supposé se faire en temps constant, c'est-à-dire via le même temps quelle que soit la case accédée.

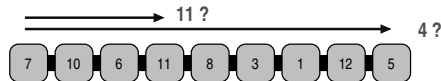
Dans une séquence non triée

Dans un premier temps on fait l'hypothèse que les données sont organisées dans une *séquence non triée*. Par exemple :



20. Souvenez-vous, ces petites galettes en plastique avec un trou au milieu.

Si la séquence est non triée, il n'y a pas d'autre alternative que de parcourir chaque élément à partir du début de la séquence et de le comparer avec la valeur qu'on recherche. On arrêtera le parcours dès qu'on la trouve ou qu'on arrive en fin de séquence. La figure ci-dessous montre deux recherches, l'une — fructueuse — de la valeur 11, l'autre — infructueuse — de la valeur 4 :



Pour analyser la complexité d'un tel algorithme, on choisit d'estimer le nombre de comparaisons. On peut noter que dans le pire des cas, il faudra parcourir toute la séquence (cas d'une recherche infructueuse). En outre, pour chercher une valeur V prise au hasard on peut dire dans une séquence composée de valeurs aléatoires, on fera en moyenne de $(N + 1)/2$ comparaisons. On peut donc dire que :

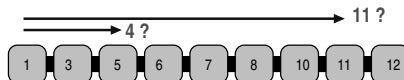
La complexité de la recherche dans une séquence non triée est $O(N)$ en moyenne et dans le pire des cas.

Dans une séquence triée

Dans une séquence triée comme celle-ci :



on pourra chercher une valeur V en effectuant un parcours de la séquence et en s'arrêtant dès que l'on rencontre la valeur V , ou la fin de la séquence, ou une valeur supérieure à V . On pourrait penser que le fait que la séquence soit triée diminue le temps de la recherche, mais il suffit que la valeur V soit située en fin de séquence pour qu'on ait à parcourir la quasi-totalité des données :



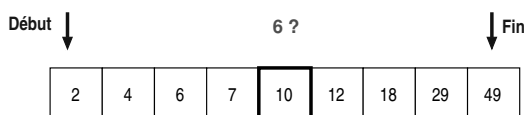
On s'intéressera ici aussi à estimer le nombre de comparaisons. On peut montrer que :

La complexité de la recherche dans une séquence triée est en $O(N)$ en moyenne et dans le pire des cas.

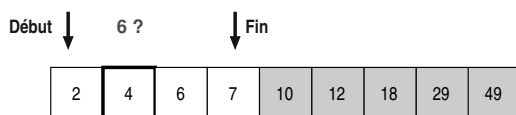
Recherche dichotomique

La recherche dichotomique est la méthode utilisée dans le jeu où un joueur fait deviner un nombre secret (entre 1 et 100 par exemple) à un autre joueur. Ce dernier propose le nombre qu'il veut et le premier joueur lui répond soit qu'il s'agit d'un nombre trop grand, soit trop petit, soit du nombre secret. Dans le cas d'un nombre secret entre 1 et 100, il est évident que la manière de deviner le *plus rapidement possible* le nombre consiste à choisir d'abord le nombre 50, puis 75 ou 25 selon le cas, etc.

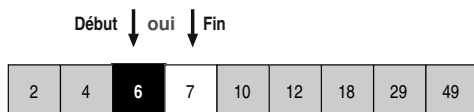
Voyons le cas d'une recherche fructueuse, par exemple si on cherche la valeur 6 :



La valeur 6 est inférieure à la valeur du milieu (10), on réitère la recherche dans la partie droite :



La valeur 6 est supérieure à la valeur du milieu (4), on cherche donc dans la partie droite, et on trouve. La recherche s'arrête :



Dans le cas d'une recherche infructueuse, les deux pointeurs de case début et fin finiraient par se « croiser ».

On peut montrer, et on le comprend intuitivement, que la complexité de la recherche dichotomique est en $O(\log_2 N)$ dans le pire des cas et en moyenne.



Une complexité logarithmique — qui peut paraître au premier abord comme une formule mathématique obscure — implique que le nombre de comparaisons effectuées se comporte logarithmiquement (c'est-à-dire, comme « l'inverse²¹ » d'une puissance de 2.). Ainsi :

- pour 1024 données, on fera 10 comparaisons car $2^{10} = 1024$
- pour 1048576 données (en gros un millions), on fera 20 comparaisons, car $2^{20} = 1048576$.

Si bien qu'au lieu de comparer séquentiellement (c'est-à-dire, une donnée puis une autre) dans plusieurs milliards de données, on ne fera que quelques dizaines²² de comparaisons !

2.7.2 Recherche par table de hachage

On appelle *table de hachage* une structure de données particulière permettant de réaliser des recherches de manière très efficace. Le principe général consiste à ranger la valeur recherchée \mathcal{V} dans un tableau à la case dont le numéro est $h(\mathcal{V})$, où h est appelée *fonction de hachage*. Nous verrons que la difficulté de la mise en place d'une table de hachage est précisément la définition de la fonction h .

Nous présentons ici une étude de cas pour comprendre les mécanismes entrant en jeu dans la table de hachage. Supposons par exemple que l'on veuille effectuer des recherches sur des chaînes de caractères.

Comme l'illustre la figure ci-dessus, on a :

- $h(\text{jean}) = 0$
- $h(\text{jeanne}) = 2$
- $h(\text{serge}) = 3$
- $h(\text{maurice}) = 6$

Donc si on veut savoir si la chaîne « jean » est dans la table, on calcule $h(\text{jean})$ qui donne 0 et en examinant le contenu de la case 0, on trouvera la chaîne. C'est donc une recherche fructueuse. À l'inverse, si on cherche la chaîne « albert », la fonction $h(\text{albert})$ donnera par

0	jean
1	
2	jeanne
3	serge
4	
5	
6	maurice
7	
8	
9	

21. Les profs de maths qui liront vont crier au scandale, certes ça n'est pas l'inverse mais la réciproque.

22. Pour info $2^{30} = 1073741824$

exemple 8, et dans cette dernière case, aucune chaîne n'est présente. « albert » n'est donc pas dans la table.

En supposant que la fonction h associant une valeur à chaque données soit bijective²³, le seul coût entrant en jeu dans la recherche ou l'insertion est l'accès à une case d'un tableau. Par conséquent :

Les opérations de recherche et d'insertion dans une table de hachage sont réalisées en temps constant (!).

2

Nous allons voir par la suite, qu'il est impossible de trouver une fonction de hachage bijective dans la plupart des cas. Une « bonne » fonction de hachage doit respecter certaines propriétés pour justifier l'utilisation d'une table de hachage. Elle doit être :

- rapide à calculer ;
- uniforme pour limiter les *collisions*.

Pour comprendre ce qu'est une collision, supposons que l'on dispose d'une fonction de hachage $h(\text{chaîne})$ telle qu'après avoir attribué une valeur à chaque lettre ($a=1$, $b=2$, etc.);

1. on fasse la somme des lettres de chaîne ;
2. on ajoute le nombre de lettres de chaîne ;
3. on calcule le reste de la division par 13.

Dans cas, le nombre d'entrées dans la table de hachage est 13, et :

- $h(\text{jean}) = 8$
- $h(\text{serge}) = 7$
- $h(\text{jane}) = 8 \leftarrow \text{collision}$

Dans la mesure où les chaînes « jean » et « jane » prennent la même valeur au travers de la fonction de hachage, on dit qu'il se produit une *collision* et il va falloir trouver une astuce pour faire face à cette situation, puisqu'on ne peut stocker deux valeurs dans la même case de la table de hachage.

Une astuce possible pour gérer les collisions consiste à stocker dans la case n une séquence contenant toutes les chaînes c telles que $h(c) = n$, c'est-à-dire prenant la même valeur n au travers de la fonction h . Comme l'illustre la figure 2.5 page ci-contre, on a :

- $h(\text{jean}) = h(\text{jane}) = h(\text{juan}) = 0$
- $h(\text{jeanne}) = h(\text{jennifer}) = 2$
- $h(\text{serge}) = 3$

23. C'est-à-dire qu'à une donnée correspond une seule valeur au travers de h et inversement.

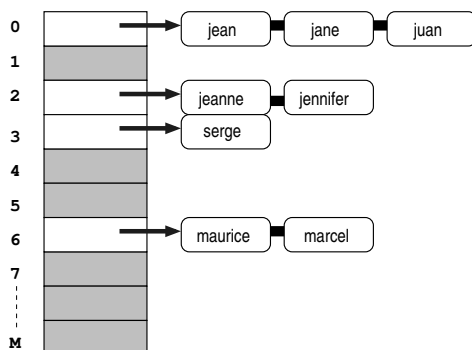


FIGURE 2.5 – Table de hachage avec collisions

$$- h(\text{maurice}) = h(\text{marcel}) = 6$$

En d'autres termes, « jennifer » et « jeanne » prennent la même valeur 2 au travers de la fonction h , on construit donc dans la case 2 de la table de hachage, une séquence contenant ces deux chaînes. On fait la même chose à la case 6 pour « maurice » et « marcel », et pour « jane » « juan » et « jean » à la case 0.

Il faut bien évidemment modifier les algorithmes de recherche et d'insertion pour tenir compte des collisions. Ainsi pour chercher une valeur \mathcal{V} , il faudra si nécessaire chercher dans la séquence associée à la case $h(\mathcal{V})$. Pour ce qui est de la complexité :

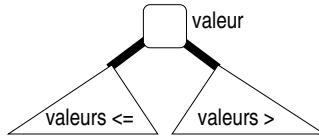
Dans une table de hachage avec gestion de collision par séquence, la recherche et l'insertion ont une complexité de :

- $O(1)$ en moyenne ;
- $O(N)$ dans le pire des cas.

On comprend aisément que la complexité en moyenne sera d'autant meilleure que les données seront uniformément réparties dans la table. En outre le *volume* de la table de hachage ne doit pas être trop important pour que la table puisse effectivement être exploitée en mémoire.

2.7.3 Recherche par arbre binaire

Nous présentons dans ce paragraphe une méthode de recherche s'appuyant sur une structure de données particulière : *l'arbre binaire de recherche*. Un arbre binaire de recherche (ABR) est une structure de données abstraite dérivée de l'arbre binaire telle que :



2

C'est-à-dire, telle qu'en chaque sommet N , le sous-arbre gauche contient des valeurs inférieures à celle contenue dans N , et le sous-arbre droit des valeurs supérieures.



Dans l'arbre complet de la figure 2.6a page suivante, chaque fils possède ses deux nœuds. En notant N le nombre total de nœuds et h la hauteur de l'arbre, on a :

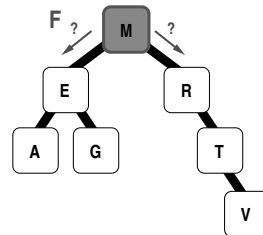
- $N = 2^h - 1$
 - $h = \log_2 N + 1$
- où \log_2 est le logarithme en base 2.

Les opérations associées à cette structure sont :

- Ajout et suppression d'une valeur ;
- Recherche d'une valeur.

La figure 2.6 page ci-contre montre deux arbres binaires de recherche contenant tous deux les mêmes données (A, B, ... , G). L'ordre utilisé est l'ordre lexicographique (A est inférieur à B).

Pour savoir si une valeur V est présente dans l'arbre binaire de recherche, on va effectuer un parcours de l'arbre en partant de la racine. Si la valeur contenue dans la racine n'est pas V on continue la recherche *de la même manière*²⁴ à gauche ou à droite selon que V est inférieure ou supérieure à la valeur contenue dans la racine.



Par exemple pour une recherche (infructueuse) de la valeur F, on commence donc par comparer F à la valeur de la racine M

24. Les algorithmiciens disent *récurivement*.

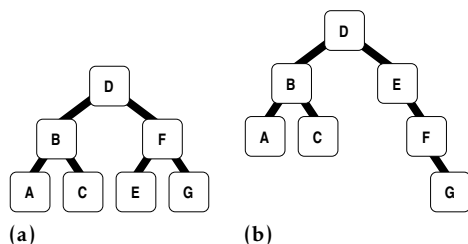
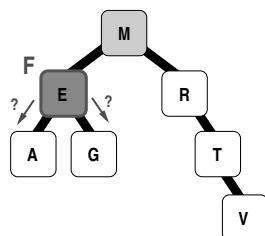


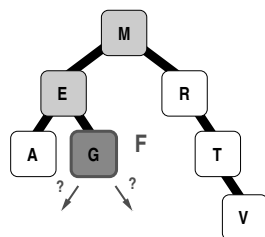
FIGURE 2.6 – Exemples d’arbres binaires de recherche : dans ces deux arbres, pour chaque nœud, le sous-arbre de gauche contient des valeurs inférieures, celui de droite des valeurs supérieures.

2

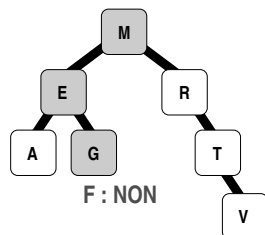
La valeur F est inférieure à M, on réitère donc la recherche à partir du fils gauche contenant la valeur E



On « arrive » donc sur le sommet contenant la valeur G et on compare cette valeur à F



La valeur F est inférieure à G on cherche donc à partir du fils gauche du sommet contenant G

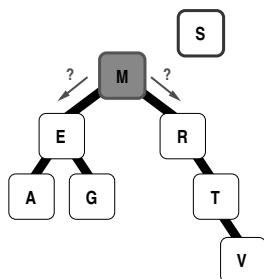


Ce sommet n’existe pas, la recherche s’arrête donc ici, sans avoir trouvé la valeur F.

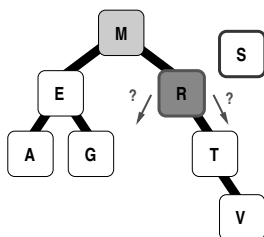
Insertion dans un ABR

Le principe de l'insertion est analogue à celui de la recherche d'une valeur ne se trouvant pas dans l'arbre. Nous avons vu précédemment que lorsqu'une recherche était infructueuse, le parcours s'arrêtait sur une feuille de l'arbre (c'est-à-dire un sommet terminal). L'opération d'insertion devra alors insérer un nouveau sommet en guise de nouveau fils de ce sommet terminal.

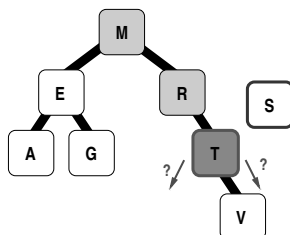
Pour comprendre le principe de l'insertion dans un arbre binaire de recherche, nous étudierons un exemple. On veut insérer la valeur S dans l'arbre binaire utilisé dans les exemples précédents. On commence par comparer cette valeur à la racine



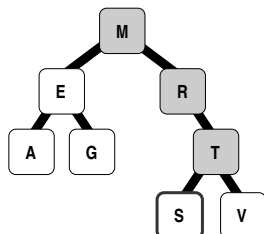
Étant donné que la valeur S est supérieure à la valeur se trouvant dans la racine, on continue à droite au sommet contenant la valeur R



De la même manière, on « tourne à droite » vu que S est supérieur à R



Arrivé au sommet contenant la valeur T, on va réitérer le processus avec le fils gauche (puisque S est inférieur à T). Or ce fils n'existe pas, on choisit donc cet emplacement pour greffer un nouveau sommet contenant la valeur S.



Complexité

Pour estimer la complexité des opérations d'insertion et de recherche sur un ABR, il faut dans un premier temps observer que pour ces deux opérations on effectue un parcours lors duquel on rencontre à peu près autant d'éléments qu'il y a de niveaux dans l'arbre. Nous avons vu au début du § 2.7.3 page 58, que le nombre de niveaux d'un arbre binaire complet composé de n sommets est $\log_2(n+1)$. Par conséquent :

Pour un arbre binaire de recherche contenant N données (c.-à-d. N sommets), les opérations d'insertion et de recherche sont en :


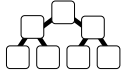
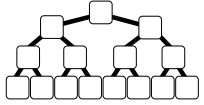
- $O(\log_2 N)$ en moyenne;
- $O(N)$ dans le pire des cas (arbre filiforme).

2

Pourquoi la complexité \log_2 est-elle « tip-top » ?

À l'instar de la recherche ▶dichotomique, la recherche dans un arbre binaire, s'il est complet, sera de l'ordre de $\log_2(N)$ dans le pire des cas, le cas d'une recherche infructueuse. N est ici le nombre de données de l'arbre, c'est-à-dire le volume de données parmi lesquelles on cherche. Le tableau ci-dessous montre la correspondance entre hauteur H et nombre de nœuds N :

§ 2.7.1 ◀
p. 52

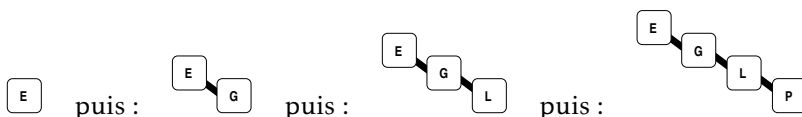
Arbre	N	hauteur H
	3	$2 = \log_2(3+1)$
	7	$3 = \log_2(7+1)$
	15	$4 = \log_2(15+1)$
...
...	1023	$10 = \log_2(1023+1)$
...	1048575	$20 = \log_2(1048575+1)$
...	1073741823	$30 = \log_2(1073741823+1)$



Comme pour la recherche dichotomique, il faut donc se persuader que si vous aviez un milliard de données et vous étiez capable de les ranger dans un arbre binaire complet (ce qui n'est pas une opération simple), vous n'auriez qu'au pire 30 comparaisons à faire pour répondre à la question : « cette valeur est-elle parmi mon milliard d'informations » !

2.7.4 Autres arbres

Si vous ne vous êtes pas endormi jusqu'ici, vous aurez compris que le défaut majeur d'un arbre binaire de recherche est qu'il peut être réduit à une simple séquence si les données se présentent « mal » pendant la construction. À titre d'exemple, les données E, G, L, P, présentées dans cet ordre, donneraient en utilisant l'algorithme d'insertion dans un ABR (§ 2.7.3 page 60) :



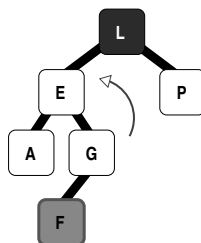
On comprend donc que l'arbre ainsi créé n'a plus aucun intérêt puisque la recherche d'un élément demande alors de parcourir tous ses nœuds. La complexité devient alors $O(N)$, en d'autres termes celle de la recherche séquentielle dans une séquence triée.

Arbres AVL

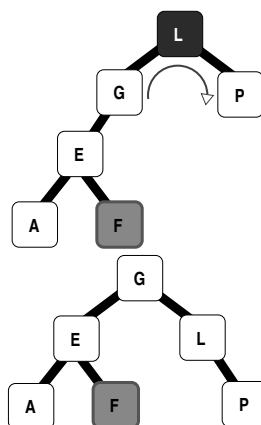
C'est la raison pour laquelle Georgii ADELSON-VELSKY et Evguenii LANDIS ont conçu au début des années 60, des arbres portant leurs noms : les arbres AVL. Ces arbres ont la particularité de se « ré-équilibrer » automatiquement pendant les opérations d'insertions (et de suppression).

Bien évidemment ces équilibrages ne doivent pas être coûteux en calcul. Dans le cas contraire cela n'aurait plus d'intérêt d'utiliser la structure. Les deux compères cités plus haut ont réussi le tour de force de réaliser les opérations de ré-équilibrage en $O(\log_2(N))$.

À titre d'exemple, nous montrons ici un arbre AVL dans lequel on vient d'insérer la valeur F. On constate que le sous-arbre gauche de la racine est plus haut que le sous-arbre droit. On va donc procéder à un « rotation gauche » autour du nœud G.



Le résultat de cette rotation est indiqué ci-contre. On réalise alors une « rotation droite » autour du nœud L.



Rotation qui donne l'arbre ci-contre.

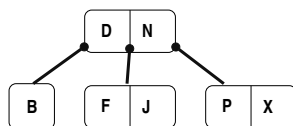
B-Arbres

D'autres arbres sont nés de l'imagination des chercheurs en algorithmie et parmi ceux-là les *B-arbres* qui sont souvent utilisés pour l'indexation des données dans un système de gestion de base de données. On ne rentrera pas dans le détail de ces arbres, dont le principe est dû à Rudolf BAYER au début des années 70, mais donnerons quelques grands principes :

- les B-arbres ne sont pas des arbres binaires mais des arbres dont les nœuds peuvent avoir plusieurs fils ;
- les nœuds eux-mêmes peuvent contenir plusieurs valeurs ;
- les B-arbres sont *constamment équilibrés*, c'est ce qui rend la recherche dans ceux-ci particulièrement efficaces.

À titre d'exemple, dans le B-arbre ci-contre, le nœud racine contient deux valeurs (D et N) et ses fils sont des nœuds contenant respectivement des valeurs :

- inférieures à D
- comprises entre D et N
- supérieures à N



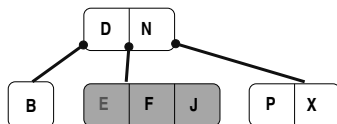
On peut imaginer qu'il s'agit d'une sorte de généralisation de l'arbre binaire de recherche et que pour effectuer une recherche dans une telle structure, on procédera comme suit :

- si la valeur recherchée est dans la racine on renvoie « c'est bon, j'ai trouvé! » ;
- sinon on la cherche dans le fils adapté, c'est-à-dire celui contenant des valeurs dans un intervalle encadrant la valeur recherchée ;

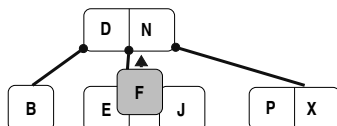
- et ainsi de suite, récursivement, avec les nœud de niveaux inférieurs.

Par exemple si on cherche à savoir si la valeur G est dans l'arbre, on traversera successivement la racine puis le nœud contenant F, J. Le processus s'arrêtera puisque ce dernier nœud n'a pas de fils comportant des valeurs comprises entre F et J, où aurait pu se trouver la valeur recherchée G.

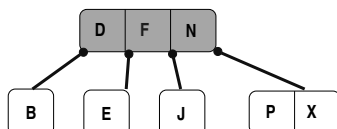
Pour insérer dans un B-arbre, par exemple la valeur E, on commencera par l'insérer comme expliqué dans le précédent paragraphe.



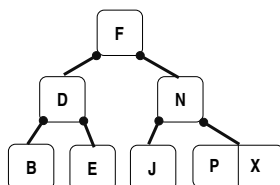
Puis, pour conserver le degré du B-arbre, c'est-à-dire, s'assurer que chaque nœud contient un maximum de 2 valeurs, on fera remonter la valeur centrale vers le niveau supérieur.



Il est ensuite probable d'avoir à répéter cette opération de « remonter » pour le niveau supérieur.



On obtiendra finalement le B-arbre ci-contre, après avoir fait remonter la valeur centrale du nœud arrivé ici à saturation.



2.8 En résumé, à quoi ça sert, tout ça ?

D'une part à vous convaincre que les **informations sont toutes élaborées à partir de séquences de bits**. Par exemple, ces 64 bits :

```
1010000101000100011011110110110001110000011010000111100100100001
```

pourront **indifféremment** être interprétés comme :

- une séquence d'octets en hexadécimal : A1446F6C70687921
- un entier non signé : 11620535450055768353₁₀
- un entier signé : -6826208623653783263₁₀
- un nombre à virgule flottante en double précision :
 $\approx 1.987064769750108 \times 10^{147}$
- la chaîne de caractère « ;Dolphy! » en ISO 8859-15.

Décider de la manière d'interpréter cette séquence de bits, c'est, dans le cadre des SGBD, choisir un domaine (équivalent du *type de donnée* en programmation).

Comprendre le codage des informations c'est **être capable de faire les bons choix pour le stockage des nombres** (entiers ou à virgule, avec l'étendue et la précision associée) et c'est aussi *maîtriser les subtilités autour des caractères accentués*.

Enfin, en un mot comme en 100 : **les arbres de recherche et les tables de hachage sont implémentés aujourd'hui dans les SGBD pour accélérer les recherches sous la forme d'un objet spécial appelé *index***. Vous serez à même d'appréhender le concept d'index grâce au lumineux exposé de ce chapitre. Cela vous sera d'autant plus utile, que les index sont absolument incontournables pour faciliter le travail des SGBD lors des diverses opérations d'extraction de données. Ces index vous sont présentés au chapitre 7 et plus précisément à la section 7.3 page 298.



- 3.1 Qu'est-ce qu'une donnée ?
- 3.2 Modèle conceptuel des données
- 3.3 Difficultés rencontrées
- 3.4 Micro-études de cas
- 3.5 Étude de cas : la discographie de FZ

Modèle conceptuel des données

*Information is not knowledge. Knowledge is not wisdom.
Wisdom is not truth. Truth is not beauty.
Beauty is not love. Love is not music.
Music is the best.*

Frank ZAPPA.

LA DÉMARCHE que nous présentons ici a pour but, à partir d'un ensemble d'informations existant, d'aboutir à une modélisation de celui-ci. En d'autres termes, de dresser, grâce à un *formalisme particulier*, une sorte de « plan » des données. L'ensemble des informations dont il est question est appelé *système d'information*. Il est constitué :

1. de l'ensemble des **données** qui transitent dans le système ;
2. de l'ensemble des **traitements** réalisés sur ces données.

Une des définitions de l'informatique étant « la science du traitement de l'information » nous voyons ici que nous sommes au cœur du domaine ! Données & traitements. Traitements & données. Comment stocker tout cela sur un système informatique ? Si ce chapitre traite uniquement de la modélisation des données, on verra que même sans aborder de formalisme qui nous permettrait de modéliser les traitements, ce sont toujours eux qui définissent le périmètre du modèle des données.

3.1 Qu'est-ce qu'une donnée ?

Nous proposons pour répondre à cette question, deux assertions :

- Frank ZAPPA et Terry Bozzio sont des *musiciens* ;
- Terry Bozzio joue de la *batterie* et Frank ZAPPA de la *guitare*.

De manière intuitive nous pouvons tirer de ces deux phrases plusieurs constatations. La première est qu'il semble que la notion de musicien soit un *concept* émergeant des données dont il est question ici. Il semblerait également que les sieurs Bozzio et Zappa soient des matérialisations¹ de ce concept. On pourra en dire autant du concept d'instrument de musique dont la batterie et la guitare sont des dignes représentants.

La deuxième phrase porte également une information importante : il semblerait qu'il y ait un lien entre un musicien et un instrument (deux matérialisations de chacun des concepts introduits). La donnée est donc ici *constituée de ce lien*.

Dans le jargon des bases de données — et dans le cadre du formalisme que nous utiliserons pour la modélisation :

1. le terme de concept est remplacé par **entité** et chaque matérialisation d'une entité se nomme *occurrence* de l'entité ;
2. les liens possibles que nous observerons entre les entités seront appelés **associations**.

3.2 Modèle conceptuel des données

Le formalisme *entité/association* permet de dresser ce qu'on appelle un *modèle conceptuel des données* (MCD). Ce qu'a de conceptuel ce modèle, c'est qu'il a pour but de représenter les différents concepts et les liens entre ces concepts. Le formalisme entité/association est un formalisme comme peut l'être un langage algorithmique qui décrit une solution. Ici le modèle entité/association permet de formuler le résultat de l'analyse et donc la modélisation du système pour ce qui est des données (cf. figure 3.1 page ci-contre).

Si notre système d'information à examiner est la discographie de Frank Zappa, il y a un concept qui émerge naturellement : la notion de musicien. Imaginons maintenant que ces musiciens constituent un *ensemble* au sens mathématique du terme : une *collection d'objets de même nature sans ordre particulier*.

1. Dans le monde de l'informatique on dit parfois *instanciations*.

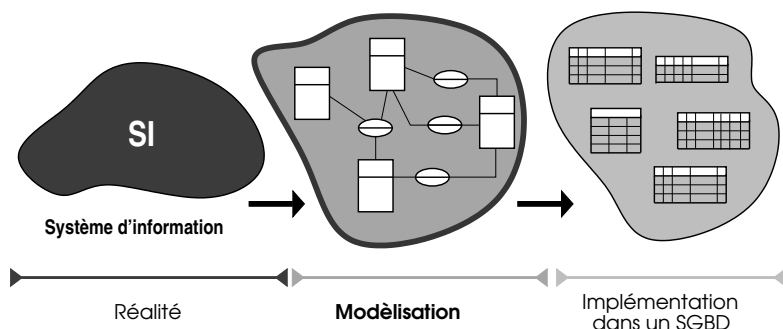


FIGURE 3.1 – Modélisation d'un SI : le système d'information a un périmètre qui circonscrit les données et les traitements qui y transitent. La modélisation conceptuelle des données — au centre — consiste à adopter un formalisme pour dresser le plan des données. La phase suivante mènera à l'exploitation de ce plan pour procéder au stockage de ces données dans une base à l'aide d'un SGBD.

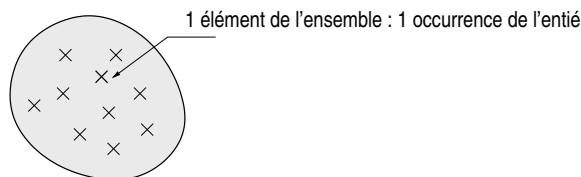


FIGURE 3.2 – Représentation ensembliste : la notion de musicien est ici représentée sous la forme d'un ensemble au sens mathématique du terme. Chaque élément est doté de plusieurs attributs qui le caractérisent, il est également appelé occurrence de l'entité.

Si nous examinons chaque élément de cet ensemble nous verrons qu'étant de même nature, ils arborent tous un certain nombre d'*attributs* : un nom, un prénom, une date de naissance, etc. Si nous avons à dresser un plan des données nous pourrions bien entendu nous contenter du modèle en « patate » de la figure 3.2 ; cependant en continuant l'examen du système et au fur et à mesure des données rencontrées nous risquerions de surcharger quelque peu la représentation.

3.2.1 Entité

Dans notre formalisme nous représenterons la notion de musicien sous la forme d'une *entité* dans un rectangle comme suit :

Musicien
Nom
Prénom
Naissance

On exprime ici que chaque musicien est doté de trois attributs, son nom, son prénom et sa date de naissance. Il est important de comprendre que ce « rectangle » désigne non pas un musicien, mais le *concept* de musicien, ou encore l'*ensemble* des musiciens. Un autre façon de voir les choses en utilisant le jargon de la programmation serait de dire que cette entité :

- correspond au *type* musicien pour lequel les *variables* représentent des instanciations ;
- correspond à la *classe* musicien pour laquelle les *objets* sont des instanciations.

De la même manière, on pourra représenter le concept d'instruments comme ceci :

Instrument
Libellé
Abrégé

indiquant l'existence dans les données qu'on manipule de la notion d'*instruments*. On supposera que chaque instrument est doté :

- d'un nom via l'attribut Libellé, par exemple « saxophone soprano » ;
- d'une version réduite de ce nom via l'attribut Abrégé qui pourra prendre la valeur « s. sax », par exemple.

3.2.2 Association

Quid maintenant des phrases suivantes :

« Terry Bozzio sait jouer de la batterie et chante. »

« Frank Zappa sait jouer de la guitare, de la basse et chante »

Un manière de représenter cette idée est de dire qu'il existe des liens entre des éléments de l'ensemble des musiciens et des éléments de l'ensemble des instruments. Chaque lien matérialise le

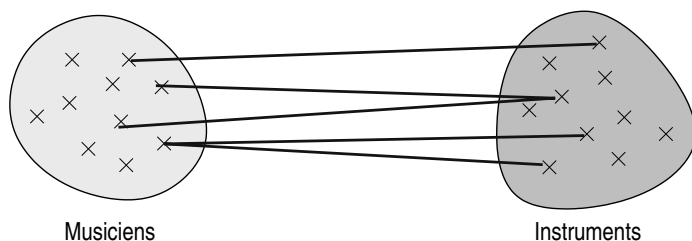


FIGURE 3.3 – Association au sens ensembliste : des liens entre les éléments des deux ensembles.

fait qu'un musicien sait jouer d'un instrument. L'ensemble de ces liens constitue ce qu'on appelle une *association* entre les deux entités Musicien et Instrument. La figure 3.3 illustre ces liens qui pourraient exister entre les éléments de l'ensemble de musiciens et ceux des instruments.

Par souci de concision, qui est un des objectifs de tout formalisme de modélisation, on représente une telle association comme suit :



Cette jolie ellipse est la forme utilisée pour indiquer dans le modèle qu'il existe une *association* entre l'entité Musicien et l'entité Instrument.

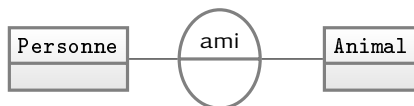


On verra par la suite que le schéma ci-dessus est incomplet car il y manque les ►cardinalités, symboles indiquant « en quelle quantité » interviennent les éléments de chaque ensemble.

§ 3.2.3 ◀
p. 74

Pas « d'orientation »

En premier lieu on notera qu'une association n'est pas « orientée », c'est-à-dire que le lien entre les deux entités peut être lu *dans les deux sens*. Ainsi l'association :



peut être lue comme :

- soit « une personne est amie avec un animal »
- soit « un animal est ami avec une personne »

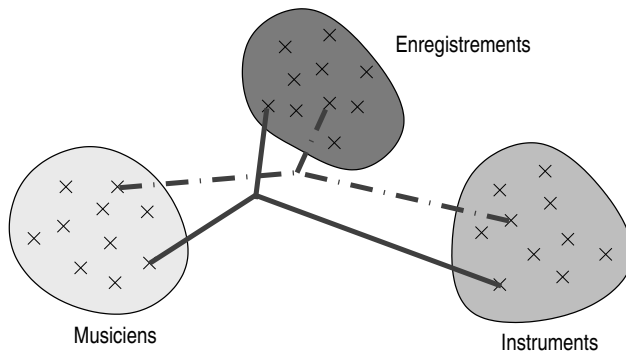
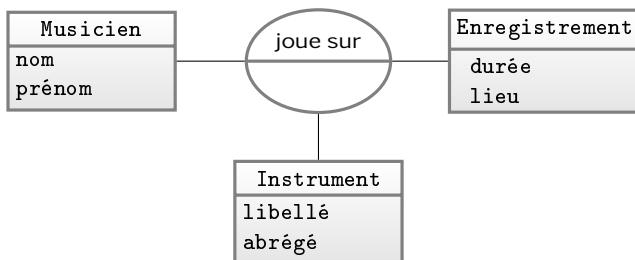


FIGURE 3.4 – Une association ternaire : chaque occurrence de l'association est une « étoile à 3 branches » allant piocher dans chacun des trois ensembles impliqués.

3

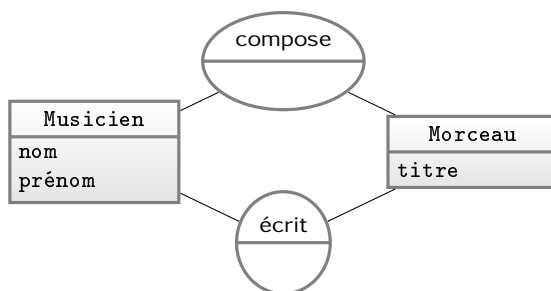
Pas de limitation à deux entités

Une association peut faire intervenir *plus de deux entités*. Par exemple si on souhaitait mémoriser le fait qu'un musicien joue d'un (ou plusieurs) instrument(s) sur un enregistrement on aurait d'un point de vue ensembliste un schéma ressemblant à celui de la figure 3.4. Du point du vue du formalisme entité/association on aurait :



Plusieurs associations possibles

Deux entités peuvent être reliées par plus d'une association. Par exemple dans le modèle ci-dessous on indique qu'un morceau et un musicien peuvent être associés par la notion de *composition* (création de la musique) et par la notion d'*écriture* (paroles du morceau le cas échéant) :



Lorsqu'on ajoute une association dans le modèle conceptuel, on est souvent confronté au choix du verbe ou du nom. Dois-je mettre « compositeur » ou « compose » ? Même si on peut bien entendu utiliser l'un ou l'autre, dans la pratique, il vaudra sans doute mieux, autant que faire se peut, se tenir à l'un des deux (toujours le verbe ou toujours le nom) pour conserver une certaine homogénéité dans la lecture du modèle.

3

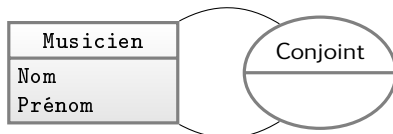
Avec des attributs

Une association *peut avoir des attributs*. Par exemple, en reprenant l'idée qu'un musicien sait jouer d'un ou plusieurs instruments, on peut imaginer qu'on veuille indiquer un ordre de « savoir-faire » pour ces instruments : 1 pour l'instrument le mieux maîtrisé, 2 pour le suivant, etc. On peut l'indiquer dans le modèle en tant qu'attribut de l'association :



S'associer « soi-même »

Une association peut lier une même entité, on parle alors d'association réflexive :



On établit alors des liens entre les éléments d'un même ensemble. L'implémentation de ce type d'association est présentée au paragraphe 4.7.4 page 130 du chapitre suivant.

3.2.3 Cardinalités

Reprenons le modèle présenté plus haut, associant les musiciens et les instruments qu'ils maîtrisent :



Dans ce modèle nous allons ajouter deux couples de symboles de part (α, β) et d'autre (γ, δ) de l'association pour indiquer dans quelle proportion sont reliés les musiciens et leurs instruments.



Ces symboles : 0, 1 ou n se nomment les cardinalités. Un modèle conceptuel des données ne veut rien dire sans celles-ci. Sans les cardinalités, il manquera une information pour construire les relations dans la base de données. En outre, un changement dans celles-ci pourra entraîner une modification importante dans la structure des relations de la base, il faudra donc prendre un soin particulier lors de leur définition.

Les cardinalités répondent à quatre questions pour chaque association. Dans l'exemple ci-dessus, il s'agit de :

- α** pour n'importe quelle occurrence de l'entité musicien, combien y a-t-il au **minimum** d'instruments qu'il maîtrise? À cette question on peut répondre 0 (aucun) ou 1 (au moins 1);
- β** pour n'importe quelle occurrence de l'entité musicien, combien y a-t-il au **maximum** d'instruments qu'il maîtrise? À cette question on peut répondre 1 (au plus 1) ou n (plusieurs);
- γ** pour n'importe quelle occurrence de l'entité instrument, combien y a-t-il au **minimum** de musiciens qui le maîtrisent? À cette question on peut répondre 0 (aucun) ou 1 (au moins 1);
- δ** pour n'importe quelle occurrence de l'entité instrument, combien y a-t-il au **maximum** de musiciens qui le maîtrisent? À cette question on peut répondre 1 (au plus 1) ou n (plusieurs).

On placera sur le schéma du modèle (voir ci-dessous) les deux premières cardinalités (min et max) à gauche de l'association (c'est-à-dire « du côté » de l'entité musicien), les deux dernières à droite de l'association, du côté de l'entité instrument. Tentons maintenant de répondre aux questions :

- 1. *pour un musicien, combien d'instruments maîtrise-t-il au moins ?* Réponse au moins 1, sinon, il ne serait pas musicien. Mais si on entend par musicien, également les auteurs ou compositeurs, on pourrait dire : au minimum 0, donc le symbole 0 ;
- 2. *pour un musicien, combien d'instruments maîtrise-t-il au plus ?* Ici pas de problème : plusieurs, donc le symbole n ;
- 3. *pour un instrument, au moins combien de musiciens le maîtrisent ?* Si on imagine que dans notre base l'ensemble des instruments sera constitué uniquement de ceux maîtrisés par les musiciens de la base, alors on pourra dire : au moins 1, donc le symbole 1 ;
- 4. *pour un instrument, au plus combien de musiciens le maîtrisent ?* On indiquera ici : plusieurs donc le symbole n.

Finalement on complètera notre MCD comme suit :



! Les réponses aux questions des cardinalités (« combien y a-t-il au plus / au moins ») ne se trouvent pas automatiquement. Dans « la vraie vie », c'est-à-dire lorsque vous aurez à étudier et comprendre un système d'information, ce sont les spécialistes du métier de ce SI qui vous donneront les réponses, et non vous.

3.3 Difficultés rencontrées

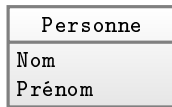
Les écueils qui surgissent le plus souvent lorsqu'on modélise un SI sont décrits dans les paragraphes qui suivent.

3.3.1 Sémantique

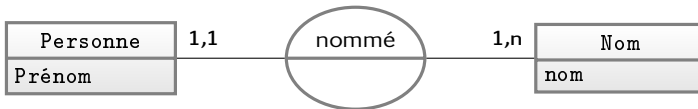
Lorsque je décide que ceci est une entité quel est le *sens* (la signification) exact du mot que j'inscris dans le rectangle ? Par exemple lorsque je crée une entité *musicien*, ce terme désigne-t-il une personne jouant d'un instrument ? Désigne-t-il également les compositeurs de musique ? Les paroliers ?

3.3.2 Attribut ou entité

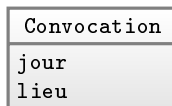
Mémoriser des personnes dans une base de données implique souvent de créer une entité :



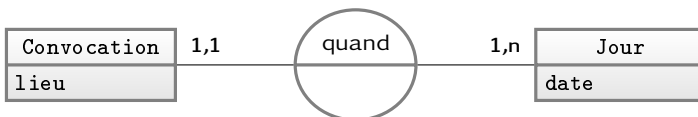
En revanche, dans le cadre d'un logiciel de *généalogie*, peut-être serions-nous amenés à exprimer les concepts suivants :



Les noms ici constitueraient un ensemble auquel seraient associées des personnes. De même il ne viendrait pas à l'idée de contester que l'entité :

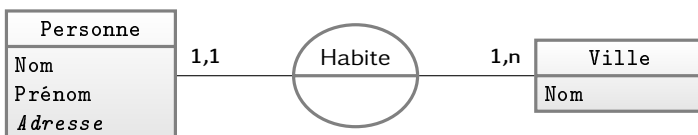


permet de modéliser une convocation à un jour donné. Pourtant :



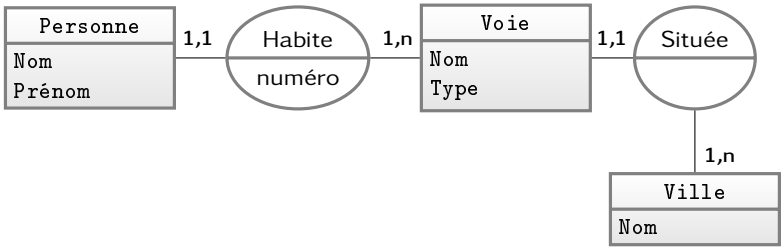
est un modèle exprimant que les convocations sont toutes reliées à un jour parmi un ensemble de jours bien définis.

Il peut également paraître délicat de décider si une information est une *entité* ou l'*attribut* d'une entité. Dans le modèle ci-dessous, l'adresse postale d'une personne est un attribut :



Ce modèle implique que l'on considère dans nos données *deux ensembles* : les villes et les personnes et que l'adresse au même titre

que le nom est un attribut des personnes. Dans une application dans laquelle des traitements exigent qu'on distingue clairement les voies (rue, boulevard, etc.) de chaque ville, ainsi que la position des habitations dans celles-ci, on pourra avoir recours au modèle suivant :



Dans ce cas la voie est considérée comme une entité à part entière. En d’autres termes, l’avenue de la Libération de Saint-Étienne est un élément distinct de l’avenue de la Libération de Clermont-Ferrand. L’adresse est alors un *numéro* dans une voie. Ce type de modélisation est indispensable pour les applications qui exigent de distinguer les différentes voies d’une ville, comme les applications autour de la gestion du territoire. Inversement si l’on a uniquement besoin d’imprimer l’adresse des personnes sur une enveloppe, il n’est pas nécessaire de modéliser les rues aussi finement.



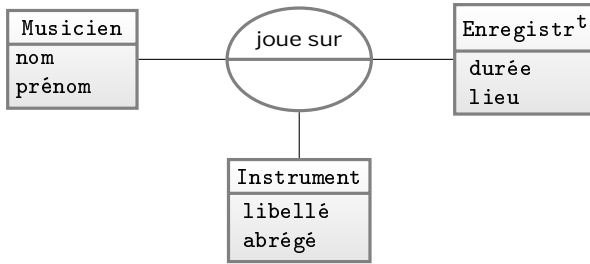
De manière générale, c’est toujours l’application et donc les traitements qui vont définir le degré de finesse de la modélisation. C’est elle qui vous guidera lorsqu’il faudra choisir si une information doit être distinguée comme un ensemble à part entière ou si on peut la cantonner dans le rôle d’attribut. Il faut savoir que dans ce dernier cas, les traitements qu’on pourra effectuer sur celles-ci seront plus limités que dans le premier cas.

3.3.3 Entité ou association

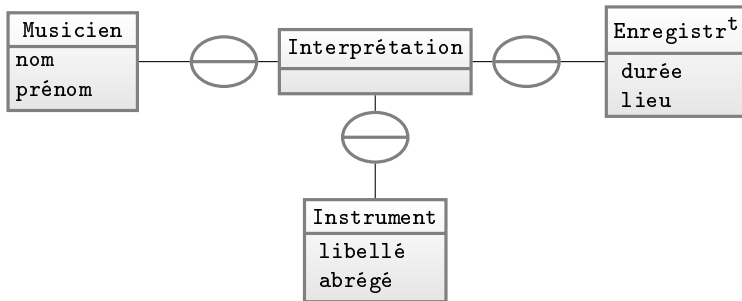
Choisir entre une entité ou une association peut également être délicat dans certaines situations.

Interprétation d’un musicien sur un enregistrement

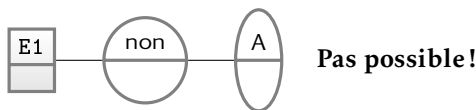
En effet, le modèle ci-dessous (déjà rencontré plus haut), traduit l’idée que l’interprétation d’un musicien avec son instrument sur un enregistrement est une association :



Mais on peut également envisager d'imaginer l'interprétation d'un enregistrement par un musicien à l'aide d'un instrument, comme une entité :



Ces deux modèles ne sont pas fondamentalement différents. Par contre seul le deuxième laisse la possibilité d'associer l'interprétation à une autre entité si le besoin s'en faisait sentir. Il n'est en effet pas autorisé dans le formalisme entité/association, d'associer une entité et une association :

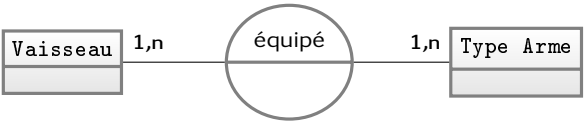


Des vaisseaux, des agents et des armes

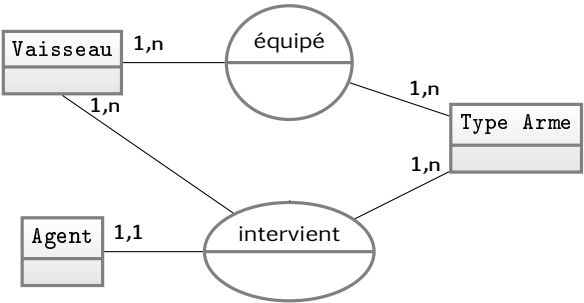
Autre cas intéressant où il n'est pas évident de distinguer entité et association, inspiré d'un film de science-fiction :

1. des vaisseaux sont équipés de différents types d'armes ;
2. des agents interviennent sur ces vaisseaux avec un type d'arme précis.

Pour le premier point on écrira :

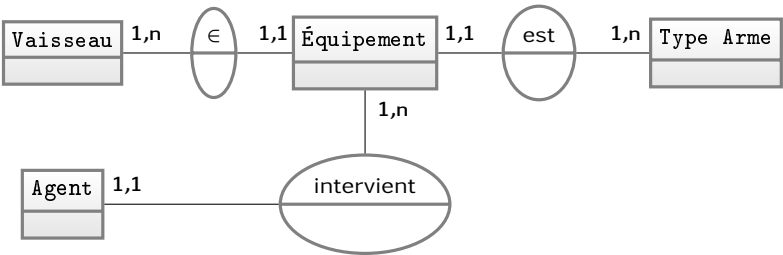


qui exprime le fait que chaque vaisseau contient plusieurs armes différentes et chacune des armes peut se retrouver dans plusieurs vaisseaux. Puis pour faire intervenir les agents sur un vaisseau et un type d’arme, on pourra dire :



3

L’association ternaire reliant un élément de chaque ensemble (arme, vaisseau, agent) permet d’exprimer le fait que chaque agent est relié à un couple (vaisseau/arme) et un seul (le paragraphe 3.5.7 page 91 explique comment trouver les cardinalités d’une association ternaire). On pourrait aussi dire que chaque vaisseau contient des *équipements* et que chaque agent intervient donc dans les vaisseaux via ces équipements :

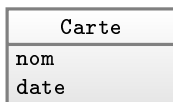


Nous montrons au paragraphe 4.7.4 page 130, comment le passage au modèle relationnel et les contraintes d’intégrité donnent un éclairage intéressant sur la légitimité de l’une ou l’autre des deux approches de modélisation proposées ci-dessus.

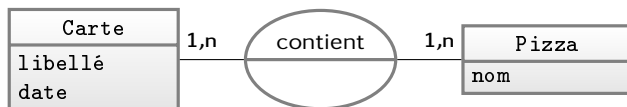
3.4 Micro-études de cas

3.4.1 Pizzas

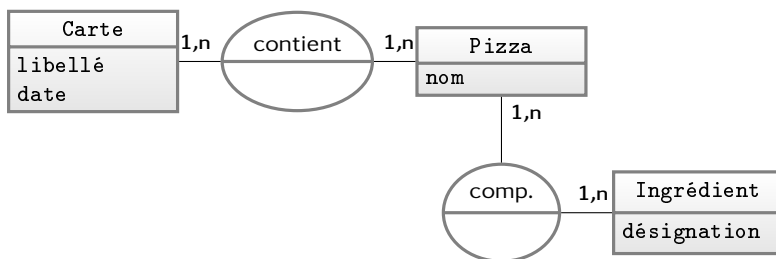
Chaque année ou tout du moins, régulièrement, la pizzeria « chez Jacky » restructure sa carte des pizzas :



Cette carte est bien entendu composée de pizzas avec pour chacune d'elles un nom :



De manière à pouvoir générer les menus, dans lesquels est affichée la composition de chacune des pizzas, on mémorise dans notre base le concept d'ingrédients entrant en ligne de compte dans la fabrication de chaque pizza :

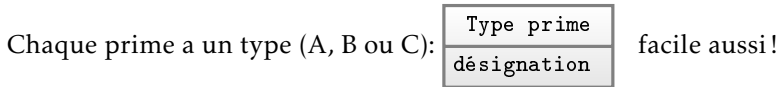


Le modèle ainsi créé garantit donc que chaque carte comporte plusieurs pizzas pouvant elles-mêmes appartenir à plusieurs cartes. Et que chaque pizza est composée de plusieurs ingrédients, eux-mêmes pouvant apparaître dans plusieurs pizzas.

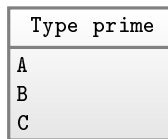
3.4.2 Primes

Nous vous proposons maintenant d'étudier la construction d'un micro-modèle autour du petit problème suivant :

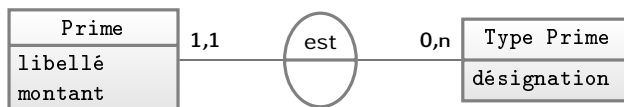
« On souhaite affecter des primes de types différents à des employés selon leur catégorie ».



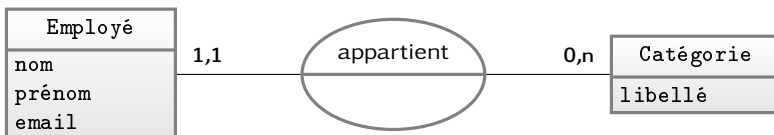
Au passage une erreur² qu'on peut être amener à commettre consiste à confondre attributs et valeurs de ces attributs et ainsi définir l'entité Type prime comme ceci :



Pour exprimer le fait qu'une prime est d'un type et un seul on écrira :



On arriverait au même type de modèle pour mémoriser l'idée qu'un employé appartient à une catégorie et une seule :

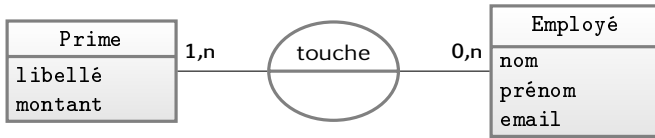


On pourra exprimer le fait qu'à chaque catégorie d'employé peuvent correspondre plusieurs primes, grâce au MCD ci-dessous :



2. Qui n'en n'est pas une dans d'autres formalismes comme UML pour ne pas le nommer.

Il reste alors à modéliser le fait qu'un employé peut toucher des primes :



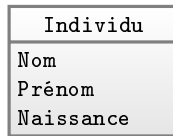
Est laissé à votre charge le soin de « recoller les morceaux » de ces différents modèles pour n'en faire qu'un seul. La façon d'implémenter ce MCD dans un SGBD est étudiée au chapitre 4 consacré au modèle relationnel. Nous ne pouvons que vous inviter vivement à le lire.

► § 4.7.4
p. 132

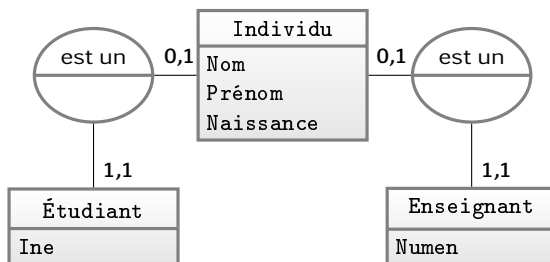
3

3.4.3 Étudiants & enseignants

Dans un établissement scolaire, on peut rencontrer des êtres humains sous forme de personnes physiques, nommons-les *individus* :



Si l'on s'approche d'un peu plus près, on se rendra compte que certains font partie du personnel enseignants, d'autres sont des étudiants. Cette idée peut être modélisée comme suit :



Ici on formalise le fait :

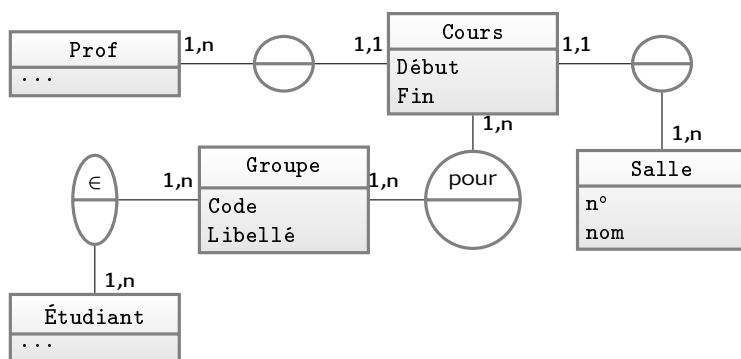
- qu'un étudiant *est un individu* (cardinalité 1,1) et qu'un individu peut être un étudiant (au plus) ou ne pas en être du tout (cardinalité 0,1) ;

- qu'un enseignant *est un individu* (cardinalité 1,1) et un individu peut être un enseignant (au plus) ou ne pas en être un du tout.

On notera que ce type de modélisation se rapproche en partie de l'approche « objet » dans le sens où, grâce aux cardinalités telles qu'elles sont définies ici, l'entité *Étudiant* va hériter les attributs de l'entité *Individu*. Ainsi, les attributs nom, prénom et naissance sont en quelque sorte communs à tous les individus. En revanche, les attributs *Ine* et *Numen*³ sont propres à chacune des entités, *Étudiant* et *Enseignant*, respectivement.

3.4.4 Emploi du temps d'un étudiant/enseignant

Nous disons en français : « *un enseignant fait cours devant un ou plusieurs groupes d'étudiants, pour une matière donnée, dans une salle de donnée, à partir d'une certaine heure et pour une durée déterminée.* ». On arrivera donc aisément à construire :



Ce modèle dit en substance :

- un cours a lieu dans une salle et une seule avec un prof et un seul ;
- il peut être dispensé à plusieurs groupes et au moins un seul ;
- chacun de ces groupes contient plusieurs étudiants ;
- etc.



La conception d'un modèle de données permettant de gérer un établissement scolaire et bien entendu certainement plus complexe que ce qui est présenté ici.

3. Identifiant national étudiant, et numéro Éducation nationale.

3.5 Étude de cas : la discographie de FZ

Pour illustrer la construction d'un MCD, nous vous proposons d'étudier le système d'information suivant : « *la discographie de Frank ZAPPA* ». Ceci paraît sans doute un peu simpliste au premier abord, mais — les aficionados de ZAPPA le savent — étudier la production discographique de ce musicien peut s'avérer ardu étant donné le volume produit (une soixantaine d'albums de son vivant) et le nombre de musiciens qui ont constitué les différentes formations. En outre, ZAPPA tenait à conserver une *conceptual continuity*, c'est-à-dire une continuité dans ses œuvres qui se sont étalées de la fin des années 60 au début des années 90. Cette continuité conceptuelle s'est notamment traduite par la reprise de certaines compositions à plusieurs années d'intervalle, avec un titre identique ou légèrement différent.

3



L'exercice auquel nous nous livrons maintenant peut être fait sur n'importe quel système d'information. La démarche à mettre en œuvre et qui va aboutir à un « plan » des données, est rigoureusement la même pour n'importe quel ensemble d'informations que vous auriez devant les yeux. Il n'y a pas de méthode miracle pour modéliser, nous vous proposons ici un guide pour y arriver. L'expérience que vous aurez acquise après avoir dressé quelques MCD vous aidera grandement pour les suivants.



Lors de la phase où l'on souhaite modéliser un SI, deux mondes se rencontrent :

- celui de l'informaticien qui a une vision traitement et stockage des informations numériques ;
- celui du spécialiste du « métier » dont on souhaite informatiser une partie.

Ces deux personnes ne parlent pas le même langage. Le rôle de l'informaticien dans cette phase, sera d'arriver à ce que le spécialiste sache livrer son savoir sans dévier vers des considérations sur (sa vision⁴ de) l'informatique. Dans cette discussion, l'informaticien aura quant à lui une difficulté à surmonter : appréhender et comprendre parfaitement les traitements et les données manipulées dans le métier.



Enfin, la finalité du MCD, en plus d'être un document dans un langage universel, à destination d'un être humain, est de préparer l'implémentation dans une base de données relationnelle. On verra

4. Souvent erronée, car les interlocuteurs ont du mal à imaginer que l'informatique — science du traitement de l'information — c'est aussi analyser, sur une feuille de papier.

en effet au chapitre suivant, qu'une fois le MCD dressé, il existe des règles qui permettent de transcrire la grande majorité de ses éléments dans le modèle dit relationnel.

3.5.1 But de la modélisation

Ce qui motive la modélisation des données d'un système d'information, ce sont *les traitements que l'on veut réaliser sur ces données*. On ne modélise pas pour modéliser. Dans notre cas, les traitements sont les questions que tout amateur de FZ souhaiterait poser sur son musicien préféré. Par exemple :

- qui a joué de la batterie sur cet album ?
- combien y a-t-il eu de versions de ce morceau ?
- sur combien d'albums ce musicien joue-t-il de la clarinette basse ?
- ...

On va donc plus loin que les simples informations basiques (mais suffisantes pour d'autres situations) contenues dans les lecteurs de fichiers musicaux (album, titre, etc.).

3.5.2 Inventaire des données

La phase suivante dans la modélisation consiste à faire *l'inventaire des données* dont on dispose. Ce qui nous intéresse ici, ce sont les différentes « sortes » de données (entité ou association) que l'on veut recenser plutôt que l'inventaire des valeurs possibles de ces données. Dans notre cas, discographie d'un musicien, on pourra réaliser cet inventaire en étalant sur une table les « pochettes » des disques et lire les informations qui y sont inscrites. On pourra par exemple noter en vrac, les informations que l'on trouve :

date	artiste	lieu du concert	musicien
instrument	pochette	paroles	album
noms des musiciens	durée d'un morceau	pistes du cd	



Tout le travail qui est à faire maintenant consiste à se demander : parmi ces mots que je viens d'écrire :

- quels sont ceux qui représentent une entité ?
 - quels sont ceux qui désignent une association ?
 - quels sont ceux qui sont des attributs d'une relation ou d'une entité ?
- certaines de ces questions induisent une réponse évidente : par exemple un musicien est clairement une entité. D'autres sont moins immédiates à traiter et c'est là toute la difficulté...

3.5.3 Construction du MCD (version 0)

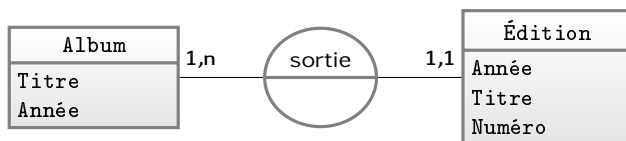
La première constatation évidente est qu'une discographie est un *ensemble d'albums*. Personne ne peut le nier... Ceci se formalise grâce à une entité judicieusement nommée album :



Ce qui peut « complexifier » quelque peu cet embryon de modélisation est qu'un même album peut être paru selon les années sous plusieurs versions, avec par exemple des morceaux bonus. On peut modéliser cela en distinguant :

1. l'entité album conçu une année donnée ;
2. l'édition d'un album parue une année donnée.

Cela peut être modélisé comme ceci :



Ici encore, c'est l'application qui décidera si vous devez aller jusqu'à ce degré de finesse ou non. Dans ce qui suit, on utilisera la forme simple (on ne distinguera pas l'album de ses différentes éditions)

3.5.4 Construction du MCD (version 1)

Poursuivons notre étude et énonçons gaillardement : « *un album contient des morceaux* »...



Ça a l'air de coller sauf qu'il faut s'interroger plus précisément sur ce qu'est un « morceau⁵ », en gardant à l'esprit ce qu'est une entité : un ensemble d'éléments, ici de morceaux de musique. Aussi, pour

5. Nous sommes ici clairement face à une des difficultés rencontrées pendant la modélisation : la signification du mot que l'on retient pour une entité.

les différentes versions d'un même « morceau » il serait gênant (pour les traitements futurs) d'avoir autant d'éléments dans l'ensemble. On ne peut imaginer avoir dans un ensemble :

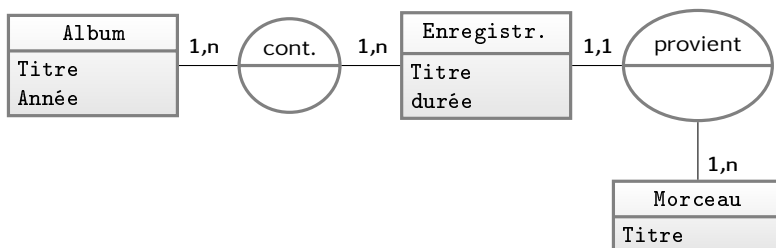
- un élément pour la version originale en studio ;
- un élément pour la version en live à Paris ;
- un élément pour la version enregistrée en concert à Tokyo ;
- ...

Après moult réflexions, on doit arriver à la conclusion qu'il faut séparer le concept de morceau en tant que *composition* et de morceau en tant qu'*enregistrement* d'une composition. On appellera donc :

Le morceau : la composition créée à une certaine date ;

L'enregistrement : la captation par des moyens idoines d'une composition.

Un album contient donc un ensemble d'enregistrements, chacun étant l'enregistrement d'un seul morceau :

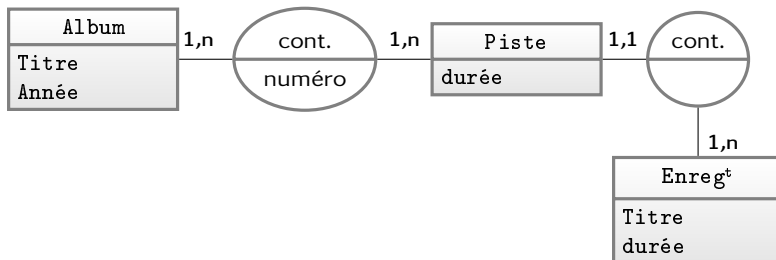


Quelques remarques importantes sur les cardinalités :

- le 1,1 entre enregistrement et provient indique qu'un enregistrement correspondant à une version d'un morceau et d'un seul ;
- le 1,n entre provient et morceau indique qu'un morceau peut donner lieu à plusieurs versions et au moins une ;
- le 1,n entre contient et enregistrement précise qu'un enregistrement (une version particulière d'un morceau) peut se trouver sur plusieurs albums différents. C'est le cas d'un enregistrement se trouvant sur un album et sur une compilation, par exemple.

3.5.5 Construction du MCD (version 2)

Un mot passé sous silence jusqu'ici est « piste ». Cette chose qui porte un numéro sur le CD⁶ de l'album. Lançons-nous : « *un album contient des pistes numérotées et chaque piste contient un enregistrement...* » (l'association entre enregistrement et morceau reste vraie par ailleurs).



Dans certains disques, notamment ceux contenant des concerts, on trouve des pistes nommées « medley » ou « pot-pourri ». Sur ces pistes, l'orchestre enchaîne plusieurs morceaux, généralement ceux grandement attendus par le public. Dans l'état actuel des choses, notre modèle ne permet pas de retranscrire ce type de morceaux puisque d'après les cardinalités :

- une piste contient un enregistrement et un seul
- un enregistrement est la version d'un morceau et d'un seul

Il faut donc modifier le MCD pour garder trace de ceci. Deux options s'offrent à nous qui consistent à agir sur les cardinalités :

- conserver le fait qu'une piste contienne un enregistrement et dire que l'enregistrement peut être la version de plusieurs morceaux ;
- conserver le fait qu'un enregistrement est la version d'un morceau et d'un seul et stipuler que la piste contient plusieurs enregistrements.

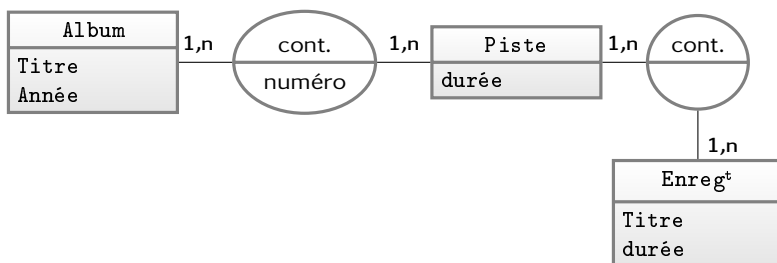
Nous choisirons la deuxième option qu'on peut également formuler de la manière suivante : « *une piste est généralement constituée d'un seul enregistrement, mais dans certains cas, de plusieurs (par exemple pour un medley). En outre une même piste peut se trouver sur plusieurs disques sous des numéros différents (par exemple dans le cas des compilations).* »

6. Nous vous rappelons que le CD est un objet inventé dans les années 80 permettant de stocker de la musique et nécessitant un rayon laser pour en obtenir la lecture.

Autrement dit, on appellera :

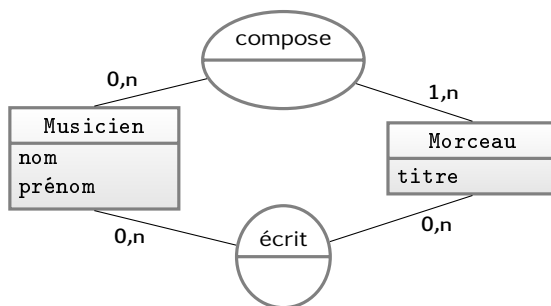
Piste une concaténation d'enregistrements clairement identifiée, pouvant se trouver à des positions différentes selon l'album.

Conceptuellement cela donne :



3.5.6 Construction du MCD (version 3)

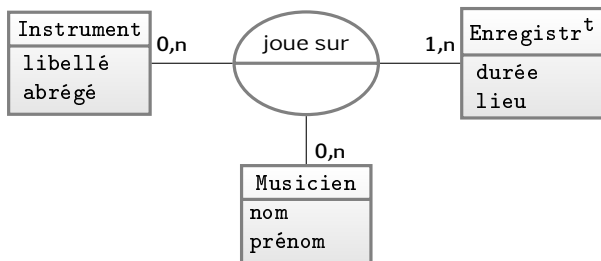
Nous n'avons pas encore inséré les musiciens dans le modèle. Comme expliqué plus haut, il faut définir précisément quel est le sens que l'on accorde au mot « musicien ». Pour simplifier on appellera musicien, toute personne participant de près ou de loin à la production de la musique : auteur, compositeur, interprète, etc. On peut donc d'ores et déjà reprendre l'extrait de modèle rencontré plus haut :



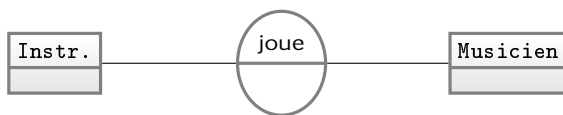
Explications des cardinalités :

- un musicien peut n'avoir composé (musique) aucun morceau ou n'avoir été auteur (paroles) d'aucun morceau, d'où les cardinalités 0,n à gauche ;
- un morceau a au moins un compositeur (1,n) et peut ne pas avoir de paroles, donc aucun auteur.

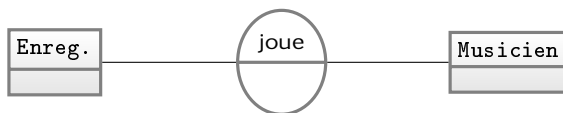
Passons maintenant aux musiciens participant à chaque enregistrement. Sur une pochette de disque est la plupart du temps inscrit, pour chaque piste, qui joue et de quel instrument. On a ainsi trois entités qui entrent en jeu : instrument, musicien et enregistrement :



Il s'agit bien d'une association ternaire et non de plusieurs associations binaires. Par exemple l'association :



n'exprimerait que l'idée : « un musicien joue d'un ou plusieurs instruments (et réciproquement) » sans préciser sur quel enregistrement. Même en y ajoutant l'association :



On ajouterait ici une autre information : « un musicien interprète des enregistrements » mais on ne préciserait pas avec quel instrument. On pourrait continuer le raisonnement avec une association entre instrument et enregistrement. En vain.

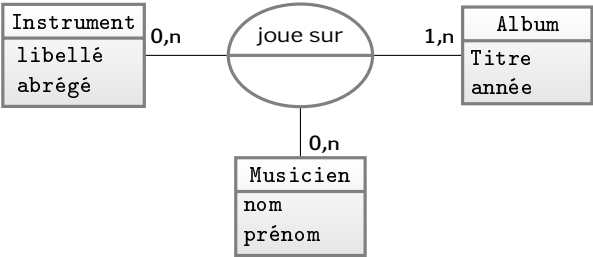
Nous n'avons pas encore eu à définir des cardinalités pour une association ternaire. Elles se définissent en se posant les questions suivantes :

1. *pour un instrument*, combien y a-t-il de couples (enregistrement/musicien) au plus et au moins ? Si l'on dispose d'une base avec tous les instruments, il est probable que certains instruments ne sont utilisés dans aucun enregistrement par un musicien. D'où la cardinalité 0,n ;

- 2. *pour un musicien*, combien y a-t-il de couples (instrument/enregistrement) qui le concernent? Ici aussi 0,n puisque certains musiciens de notre base ne sont que des auteurs ou des compositeurs, ils n'apparaîtront donc sur aucun enregistrement;
- 3. *pour un enregistrement*, combien y a-t-il de couples (musicien/instrument)?. Ici c'est 1,n puisqu'il faut au moins un musicien et son instrument pour produire un enregistrement⁷.

3.5.7 Construction du MCD (version 4)

Sur certains pochettes d'album sont uniquement indiquées les interventions des musiciens sans préciser sur quelles pistes. On a donc pour cela le modèle suivant :



3

On pourrait également ne conserver que l'association « joue sur » précédente (liant instrument/musicien/enregistrement) et faire en sorte que si l'information n'est pas présente, alors on suppose que les musiciens indiqués jouent de tous les instruments sur toutes les pistes. Ceci est tout à fait implémentable grâce à une routine (voir le chapitre 6 page 207).

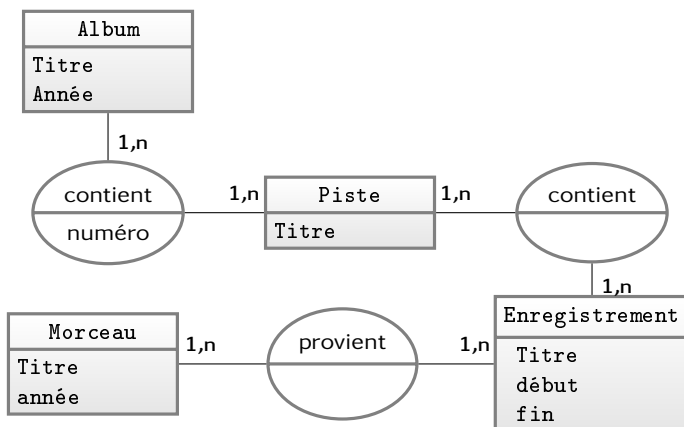
3.5.8 Construction du MCD (version 5)

Une partie des albums de Frank ZAPPA contient des pistes très particulières dans lesquelles :

- on débute par une version de 1974 d'un morceau *M* pendant quelques minutes;
- puis le morceau continue mais dans une version de 1984;
- puis de nouveau la version de 1974.

7. Les esprits taquins me rétorqueront qu'il existe probablement une composition de John Cage où un réveille-matin joue de l'escalator, donc c'est 0,n...

Le tout paraît être le même enregistrement car la musique est parfaitement enchaînée, une oreille attentive entend seulement un changement de « couleur » de l'orchestration. Notre modèle répond correctement à cette situation, car une piste peut contenir plusieurs enregistrements d'un même morceau. On peut même s'offrir le luxe d'indiquer — pour coller à ce type de piste hybride — le minutage :



Quelques remarques :

- l'entité Piste contient un attribut « titre » ce qui permettra de nommer le pot-pourri s'il y a lieu ;
- l'entité Enregistrement contient également un « titre ». Il n'est pas rare de trouver des morceaux dont le titre subisse quelques variations avec les versions et les années ;
- cette même entité contient maintenant deux attributs « début » et « fin » (en secondes écoulées depuis le début de la piste) permettant de positionner l'enregistrement dans la piste et également d'en calculer la durée.

3.5.9 MCD final (?)

Le modèle de la figure 3.5 page ci-contre est une reconstitution de toute notre discussion autour des subtilités que nous souhaitons mémoriser autour de la discographie de Frank ZAPPA.



Savoir si ce modèle est valide ou non revient à se poser la question de savoir si les traitements qu'on envisage sur ces données sont possibles ou non.

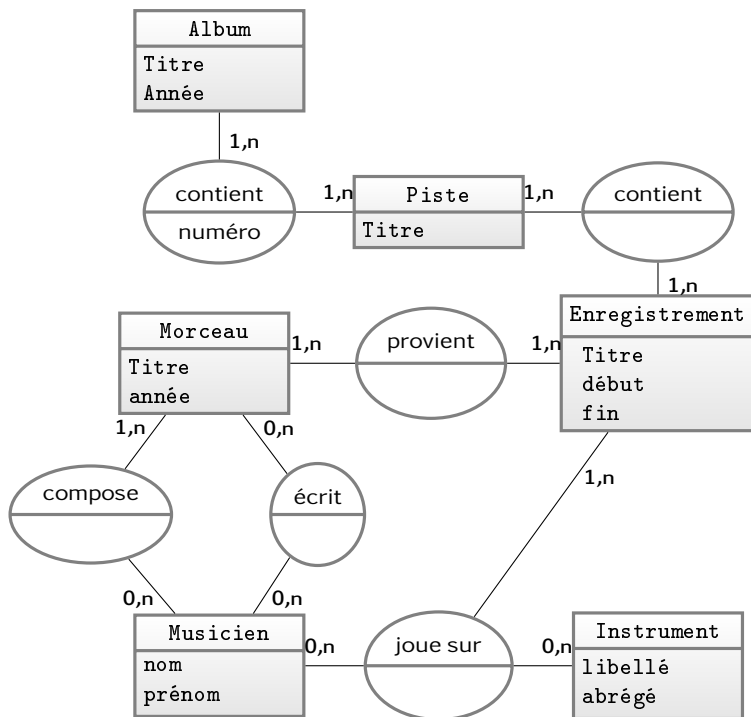


FIGURE 3.5 – Modèle conceptuel des données résultant de toutes nos cogitations.

En examinant le modèle, on doit pouvoir répondre oui à toutes les questions qui nous ont motivés à l'élaborer (entre autres, les questions du paragraphe 3.5.1 page 85).

- qui joue de la batterie entre 1972–1976 ?
- sur quel autre album se trouve aussi la piste 7 de cet album ?
- combien y a-t-il de versions de ce morceau ?
- ...

What's next ?

La suite du livre vous présente comment, grâce au langage SQL, nous allons pouvoir **donner vie aux MCD** dans les logiciels spécialisés que sont les systèmes de gestion de bases de données, et ce grâce au **langage SQL**.



- 4.1 Introduction
- 4.2 Ceci n'est pas une table
- 4.3 MCD → MR : « 1 à plusieurs »
- 4.4 « C'est pas faux »
- 4.5 Comment identifier un tuple
- 4.6 Un peu plus loin avec les contraintes
- 4.7 MCD → MR : « plusieurs à plusieurs »
- 4.8 Gestion des dates
- 4.9 Tas d'octets
- 4.10 Normalisation du modèle relationnel

Bâtir les données

*And so castles made of sand,
Fall in the sea, eventually.*

Jimi HENDRIX.

CE CHAPITRE aborde les outils du modèle relationnel et de SQL pour construire les données à partir d'un ►MCD. Le choix pédagogique que vos humbles serviteurs assument pleinement est de présenter le modèle relationnel par le truchement du langage SQL. Vous pourrez trouver dans d'autres ouvrages une approche purement mathématique de ce modèle si vous en sentez le besoin. Nous guidons donc le lecteur, dans les méandres de ce chapitre, vers les notions de *relation*, de *clés primaires et étrangères*, de *contraintes d'intégrité*, dans l'objectif de pouvoir implémenter les modèles conceptuels élaborés aux chapitres précédents.

Nous montrerons également comment doivent être mémorisées les *dates* dans une base de données, ainsi qu'un moyen de stocker des *informations brutes* (comme des fichiers image, son, etc.). Enfin, sont données quelques pistes pour comprendre les *formes normales*, outils de validation du modèle relationnel.

Chap. 3 ◀
p. 67

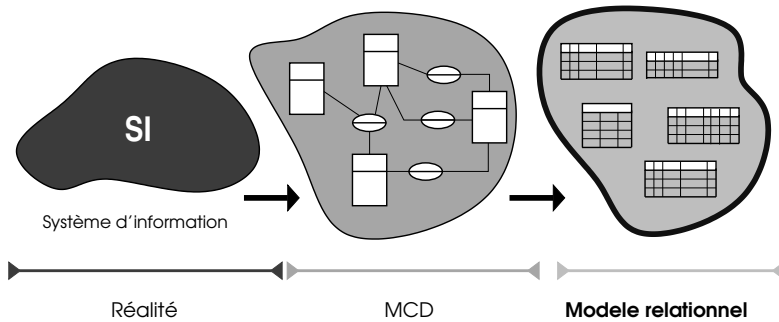


FIGURE 4.1 – Création d’une base de données relationnelle : à partir d’un modèle conceptuel des données (MCD), on va utiliser un ensemble de règles de conduite qui vont nous permettre, grâce au langage SQL, de bâtir les données de notre base sous forme de relations.

4

4.1 Introduction

Nous partons ici du principe que vous avez digéré les grands principes de la modélisation conceptuelle des données abordée au chapitre précédent. L’objectif ici est de vous exposer comment on peut, à partir d’un MCD, structurer une base de données en créant un ensemble d’objets (nommés *relations*), tout ceci grâce à ce merveilleux langage qu’est SQL (cf. figure 4.1).

4.1.1 Structured Query Language

L’**intégralité** des concepts mis en œuvre dans ce chapitre consacré à la structuration des données *peut être exprimée grâce au langage SQL*. Comme évoqué dans l’historique de ce langage de requête en introduction, il est particulièrement intéressant de noter que SQL est né dans le but d’*implémenter* les idées d’Edgar Frank Codd autour de sa *théorie de l’algèbre relationnelle*. Il n’est en effet pas si courant dans le domaine de la science et de la technologie, que la mise en œuvre vienne immédiatement après voire pendant la théorie.

► § 1.4
p. 7

4.1.2 Introduction aux contraintes d’intégrité



On appelle contrainte d’intégrité une condition posée par le concepteur de la base sur une partie des données et dont le Sgbd garantira l’inviolabilité.

On verra qu'il est par exemple possible d'imposer que :

- la valeur de cet attribut doit être une chaîne composée de trois caractères exactement ;
- il n'est pas possible de ne pas avoir de nom pour les personnes ;
- la valeur indiquée pour le prix ne doit pas être nulle ;
- il ne pourra pas y avoir deux fois la même référence pour les articles ;

La motivation première est la **cohérence des données** : en effet, les contraintes d'intégrité ont pour objet cette cohérence et c'est ce qui constitue l'une des fondations de la conception d'une base. Le SGBD est le logiciel « cerbère » qui permet de mettre leur mise en œuvre.

4.2 Ceci n'est pas une table¹



Une erreur fréquemment commise lorsqu'on aborde les systèmes de gestion de base de données relationnelles est de confondre les tables avec les feuilles de calcul d'un tableur. Il faut donc vous enlever cette idée de la tête : une table est une relation et par conséquent peut être vue comme un ensemble au sens mathématique du terme.

Une autre façon de voir les choses est de s'imaginer qu'un ensemble est un « sac », ce sac contient des *éléments*, chacun possédant des attributs. Je peux piocher un élément ou plusieurs dans ce sac, il n'y pas d'ordre pré-établi. De la même manière si j'ausculte un de ces éléments je considérerai les attributs indépendamment d'un ordre quelconque. Ainsi **les deux relations** ci-dessous **sont identiques** :

nom	prenom
A	B
C	D
E	F

prenom	nom
F	E
D	C
B	A

La conception des données dont il est question dans ce chapitre va consister à définir la forme commune des éléments de ce sac.



Même si certains Sgbd les gèrent correctement, nous ne mettons volontairement ni accent ni espace dans les noms des relations et de leurs attributs. Les espaces risqueraient d'induire en erreur l'interpréteur SQL qui compte sur eux pour repérer les éléments « mots » de votre requête. Unpeucommenousleshumains.

1. Ni une pipe d'ailleurs...

4.2.1 Définir une relation et ses attributs

La syntaxe de Sql pour créer une table est la suivante :

```
create table
    musicien(nom varchar(50), prenom varchar(30))
```

Ceci crée la table, qui peut également être vue ici comme un ensemble vide :

Musicien	
nom (chaîne)	prenom (chaîne)

On impose ici le nom et le type des attributs, chaque SGBD en proposant plusieurs que nous découvrirons au fur et à mesure de ce fantastique manuel.



Vous pouvez doré et déjà accepter le fait que la liste des types disponibles dans chaque Sgbd n'est pas standardisée. Si bien que vous rencontrerez selon l'outil que vous utiliserez des noms qui pourront être différents. C'est le cas des types chaînes de caractères : dans PostgreSQL, on rencontre le type **varchar** avec une indication de taille maximale et le type **text** sans avoir à préciser la taille (la documentation vous indiquera la taille max autorisée). C'est également le cas pour les types numériques qui portent des noms différents selon les logiciels.

On peut éventuellement ajouter ou supprimer des attributs, par exemple :

```
alter table personne add taille int
alter table musicien drop prenom
```



Dans la console psql on peut consulter la structure de la relation grâce à la commande \d. Ainsi :

```
bacasasable=> \d musicien
```

affichera :

Table "public.musicien"		
Column	Type	Modifiers
nom	character varying(50)	
prenom	character varying(30)	
taille	integer	

Peut-être souhaitez vous un jour « annoter » les objets que vous créerez dans votre base. Ceci est possible grâce à la commande **comment** qui permet d'associer une description à tous les objets, en particulier les colonnes :

```
comment on musicien.nom is 'nom_du_musicien'
```

Cette description ne sera visible dans la console de PostgreSQL qu'avec la variante \d+ de la commande \d.



L'interpréteur SQL ne fera pas de distinction entre minuscules et majuscules, ni pour les mots réservés du langage (**alter**, **create**, etc.), ni pour le nom des objets que vous créerez pas plus que pour leurs attributs. Vous pourriez donc écrire :

```
aLTER Table PERSonne aDD TAille iNt
```

Pour ce qui est données dans les tables elles-mêmes, le Sgbd fera en revanche la distinction.

On aura à plusieurs reprises dans les « phrases » exprimées en SQL, à référencer les attributs d'une relation. On utilisera dans certain cas la notation suivante, qui consiste à *préfixer l'attribut par le nom de la table* :

```
musicien.nom  
musicien.taille  
etc.
```



Il est important de noter qu'on s'interdira, dans le modèle relationnel, de créer des attributs non atomiques, c'est-à-dire qui pourraient être divisés en plusieurs valeurs. Par exemple :

Musicien		
nom (chaîne)	prenoms (chaîne)	telephones (chaîne)
Zappa	Frank, Vincent	7540,8435
Hendrix	Jimi, Marshall	6522,7830

Nous ne disons pas ici qu'il est impossible de stocker plusieurs téléphones ou prénoms pour une même personne, mais que ça n'est pas la bonne manière de faire dans le cadre du modèle relationnel. Les différentes recherches qu'on pourrait vouloir faire sur ces attributs seraient très difficiles à exprimer en SQL.



Notons enfin, pour clore ce court paragraphe introductif, que le type des attributs (chaîne de caractères, nombres, etc.) s'appelle le domaine dans le jargon du modèle relationnel². Imposer qu'un attribut (par ex. `musicien.taille`) soit de type entier est une première forme de contrainte d'intégrité et se nomme l'intégrité de domaine. Le Sgbd garantira en effet que l'attribut `musicien.taille` ne pourra contenir autre chose que des nombres entiers.

4.2.2 Une table contient des « tuples »

Dans le jargon du modèle relationnel, un *tuple* désigne à la fois :

- une ligne d'une table ;
- un élément si on considère la table comme un ensemble.

Une relation dotée de deux attributs contiendrait des *couples*, celle qui aurait trois attributs, des *triplets*, etc. Pour n attributs les francophones disent des n -uplets et les anglophones des t -uples ou *tuples*.

Si l'on revient à cette notion d'ensemble, il est important de comprendre qu'on doit pouvoir distinguer chacun des éléments, par conséquent, on ne devrait pas se trouver dans la situation suivante :

Musicien	
prenom	nom
Zappa	Frank
Van Vliet	Don
Zappa	Frank

car rien ne distingue les deux « Frank Zappa » de cet ensemble.

4.2.3 Chaque table doit avoir une clé primaire

Comme introduit ci-dessus, étant donné que dans l'algèbre relationnelle cette table représente un ensemble et qu'il ne peut y avoir deux éléments identiques dans un ensemble, **il ne peut donc y avoir deux lignes identiques dans une table**. Ainsi dans une relation qui contiendrait :

2. Certains vont d'ailleurs même jusqu'à énoncer qu'une relation est un sous-ensemble du produit cartésien de ses domaines. Nous vous laissons cogiter sur cette assertion, en ayant pris connaissance de § 5.3.1 page 178.

Musicien	
nom	prenom
Van Vliet	Don
Van Vliet	Don

On serait incapable³ d'agir sur un des deux tuples sans agir sur l'autre. Ceci est une façon de nous amener à la première règle sur laquelle on peut s'appuyer aveuglément pour concevoir les données.



Toute relation doit être munie d'une clé primaire. C'est-à-dire qu'il doit exister un attribut (ou un ensemble d'attributs) dont la valeur est unique pour chaque ligne et qui ne peut pas être vide.

Dans une table comme celle-ci :

Musicien	
prenom	nom
Zappa	Frank
Van Vliet	Don
Bozzio	Terry

l'attribut nom pourrait être la clé primaire, puisque chaque nom existe et est unique. Mais vous comprendrez aisément que l'ajout de Dweezil ZAPPA rendrait invalide notre assertion « tous les noms sont uniques ». Qu'à cela ne tienne, décidons que nom, prenom en tant que *couple de valeur*, soit la clé primaire. Cela nous permet d'ajouter Dweezil. Par contre, l'insertion des deux musiciens homonymes Bill EVANS (l'un jouant du piano, l'autre du saxophone) rendrait une fois encore caduque notre supposition sur l'unicité.

C'est la raison pour laquelle, pour le cas qui nous préoccupe, nous allons ajouter un attribut de type entier à la table musicien qui nous permettra de différencier chaque tuple :

```
alter table musicien add idmus int
```

Puis nous allons poser la contrainte de clé primaire. En Sql ceci peut être exprimé grâce à l'incantation vaudou suivante⁴ :

3. Si vous cherchez bien, vous pourriez certes y arriver, mais au prix de manipulations assez peu intuitives qui consisteraient toutes à tenter d'identifier chacun des tuples grâce à un attribut « système ».

4. Que vous pouvez bien entendu n'écrire que sur une seule ligne dans votre interpréteur.

```
alter table musicien
  add constraint pk_musicien
  primary key(idmus)
```

où `pk_musicien` est le petit nom⁵ donné à la clé primaire. Ce petit nom pourra servir à supprimer cette contrainte :

```
alter table musicien drop constraint pk_musicien
```

Voici à quoi pourrait ressembler le contenu de la table une fois la contrainte de clé primaire posée :

Musicien		
<i>idMus</i>	prenom	nom
1	Zappa	Frank
2	Van Vliet	Don
5	Bozzio	Terry

4



La contrainte « clé primaire » est donc la deuxième contrainte d'intégrité rencontrée. Il s'agit d'un « verrou » posé par le concepteur de la base, sur un ou plusieurs attributs (ici `idMus`). Une fois cette contrainte posée, il devient impossible par quelque biais que ce soit :

1. d'avoir deux tuples ayant la même valeur de `idMus`
2. d'avoir un tuple sans valeur pour `idMus`

Moult questions surgissent maintenant dans votre esprit. Les quelques remarques qui suivent vont tenter d'y répondre, partiellement pour le moment :

- on confond souvent l'attribut et la contrainte posée sur l'attribut, on dit donc abusivement « *idMus est la clé primaire de musicien* » on devrait dire « *il a été posé une contrainte de clé primaire sur l'attribut idMus* » ;
- la clé primaire est ici une clé dite artificielle ou de substitution (*surrogate* en anglais), on l'a ainsi fabriquée de toutes pièces pour associer à chaque musicien un numéro unique, ce numéro n'a pas de signification dans le modèle de données ; on verra :

5. Il n'y a aucune contrainte sur le nom utilisé exceptées les règles habituelles (pas d'espace⁶, de tiret '-',...).

6. En fait les espaces sont possibles, mais mieux vaut éviter d'en mettre pour simplifier l'écriture des requêtes. Faites-nous confiance⁷.

7. Vous pouvez sans crainte⁸.

8. Même si vous êtes en droit de douter de l'intégrité intellectuelle de personnes qui tiennent absolument à avoir moult notes en bas d'une même page.

1. qu'il existe d'autres clés possibles pour cette table, le paragraphe 4.5 page 112 ouvre une discussion sur les avantages et inconvénients des clés de substitution ;
 2. que les SGBD proposent tous des mécanismes permettant de générer automatiquement les clés dites « surrogates ».
- lorsqu'on dit qu'un attribut clé primaire doit obligatoirement avoir une valeur on fait référence à la notion ►d'absence de valeur ;
 - la contrainte d'unicité induite par la clé primaire ne concerne que l'attribut sur lequel est posé la clé, ainsi on pourrait très bien avoir dans la table *musicien* :

§ 4.4 ◀
p. 109

Musicien		
idMus	prenom	nom
1	Zappa	Frank
2	Van Vliet	Don
7	Zappa	Frank

Ici les deux « Zappa » seraient considérés comme des éléments distincts de l'ensemble, et dans ce cas deux personnes homonymes.



Dans la console psql il est possible de voir la clé primaire grâce à la commande \d :

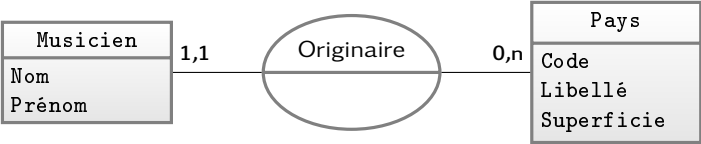
```
yahozna=> \d musicien
          Table "public.musicien"
  Column |          Type          | Modifiers |
  -----+-----+-----+-----+
  idmus  | integer                | not null  |
  nom    | character varying(50)  |           |
  prenom | character varying(30)  |           |
Indexes:
    "pk_musicien" PRIMARY KEY, btree (idmus)
```

La rubrique ►Indexes (sur laquelle nous apportons un éclairage au chapitre 7) fait apparaître notre clé primaire. On notera également en face du champ idmus l'indication « not null » rappelant que la valeur de cet attribut ne peut être vide.

§ 7.3 ◀
p. 298

4.3 MCD → MR : « association 1 à plusieurs »

Imaginons la situation où dans notre modèle conceptuel existe une association dite de « 1 à plusieurs » c’est-à-dire que de part et d’autre de l’association en question, les *cardinalités maximales* sont 1 d’une part et *n* de l’autre. Voici un exemple :



L’approche triviale⁹ serait de créer une table comme celle-ci :

Musicien					
idMus	prenom	nom	code	pays	superficie
1	Zappa	Frank	US	États-Unis	9834
2	Van Vliet	Don	US	États-Unis	9834
3	Ponty	Jean-Luc	FR	France	644

Cette solution « marche » mais comporte un inconvénient majeur qu’est la *duplication des données* du pays, ainsi :

- une modification de « États-Unis » en « United States of America » devra être répercutée dans tous les tuples concernés ;
- s’il l’on souhaitait stocker d’autres informations sur le pays (par exemple stocker le libellé en plusieurs langues), il faudrait également dupliquer ce surplus d’information dans chacun des tuples.

Par conséquent, en *dupliquant les informations* (nom et superficie du pays), on rend fastidieuses les mises à jour qui nécessiteront des calculs inutiles dans le but de **conserver la cohérence des données** (si je dois modifier la superficie, il faut que je le fasse pour chaque musicien de ce pays). La technique utilisée dans cette situation consiste à créer une table pour l’entité pays :

```
create table pays(code char(2), nom text,
                 superficie int)
```

9. Pour ne pas dire « tableau »...

Pays		
code (chaîne)	nom (chaîne)	superficie (nombre)

On applique alors sauvagement la première règle concernant les contraintes — toute table doit avoir une clé primaire ¹⁰ — et on dote la table de sa clé primaire :

```
alter table pays
add constraint pk_pays primary key(code)
```



L'iso (international standard organisation), de laquelle émane un grand nombre de normes y compris pour l'informatique, a affecté un code et un libellé aux 260 pays du globe. La norme en question porte le numéro 3166 et propose un code de pays sur deux caractères.

Voici un extrait de notre table ainsi initialisée :

Pays		
code	nom	superficie
US	United States	9834
GB	United Kingdom	242
FR	France	644
...	...	

Il s'agit maintenant de *matérialiser l'association* reliant l'entité musicien et l'entité pays, association permettant de mémoriser le pays de naissance d'un musicien. La technique utilisée pour y parvenir est la suivante :

```
alter table musicien add origine char(2)
```

qui nous donne :

Musicien			
idMus	prenom	nom	origine
1	Zappa	Frank	US
2	Van Vliet	Don	US
3	Ponty	Jean-Luc	FR

10. Suivez un peu quoi...



Avant de détailler les mécanismes qui vont garantir que les deux relations ainsi créées sont reliées de manière sûre via une clé étrangère, nous attirons votre attention sur le fait que si nous « scindons » maintenant les données pour les répartir dans plusieurs tables, il faudra un jour ou l'autre les « réunir » pour les présenter à l'utilisateur final. Ceci est l'objet des jointures◀.

► § 5.3
p. 177

4.3.1 Intégrité référentielle: clé étrangère



Créer un attribut `musicien.origine` de type `char(2)` garantit uniquement l'intégrité de domaine (il n'y aura dans ce champ que des valeurs constituées de caractères uniquement). Mais il est par contre tout à fait possible de positionner la valeur de champ à un code de pays inexistant. Cette situation entraînerait une perte dans la cohérence des données, c'est la raison pour laquelle a été créée la contrainte d'intégrité référentielle : la clé étrangère.

Pour garantir que les valeurs que prendra l'attribut `origine` de la relation `musicien` seront obligatoirement présentes dans la table `pays`, on dit (en SQL) :

```
alter table musicien
add constraint fk_musicien_origine
foreign key(origine) references pays(code)
```

ce qui (en français) veut dire : il existe dorénavant une contrainte de clé étrangère imposant que la valeur de `musicien.origine` fasse *référence* (c'est le sens du mot *references*, conjugaison du verbe *to reference* à la troisième personne du singulier en anglais) à l'attribut `pays.code`.



Une fois qu'une contrainte de clé étrangère (`foreign key` en anglais) est posée¹¹, vous pouvez dormir sur vos deux oreilles, le Sgbd vous garantira le contrôle des données dans trois situations :

1. insertion ou mise à jour du champ `musicien.origine` : comme indiqué ci-dessous, il n'est pas possible que cet attribut prenne une valeur non présente dans la table `pays` ;
2. suppression dans la relation `pays`
3. mise à jour dans la relation `pays`

Nous allons voir en détails les deux dernières situations.

11. Par le concepteur, il faut le répéter : les SGBD ne posent pas eux-mêmes les contraintes d'intégrité, c'est vous.



En passant, on notera que la commande magique `\d` de la console de `psql` affiche des informations claires sur la contrainte de clé étrangère¹² :

```
yahozna=> \d musicien
...
Foreign-key constraints:
    "fk_musicien_origine" FOREIGN KEY (origine)
                                REFERENCES pays(code)
```



Dans une situation très simple — comme par exemple, un exercice d'école — où la table `pays` ne serait référencée que par la seule table des musiciens, alors la table `pays` pourrait être réduite à une liste de noms (pas besoin ni de code ni de superficie). Dans ce cas, le nom (qui existe et est unique) est une clé naturelle et suffit donc par définition, à identifier chaque pays. Cet attribut (le nom du pays) serait alors celui qui migrerait dans la table des musiciens.

4.3.2 Ce que contrôle une clé étrangère

4

Suppression dans la table cible : avec les données définies ci-avant, la suppression du pays « France » dans la table `pays` entraînera une situation catastrophique puisque le célèbre violoniste Jean-Luc PONTY se retrouverait originaire d'un pays inexistant ! *le SGBD interdira donc cette opération*. Il existe cependant deux autres possibilités :

Suppression en cascade : dans ce cas la suppression de la France entraîne la suppression de tous les tuples pour lequel l'origine est positionnée à 'FR'. On exprime ceci, *au moment de poser la contrainte* :

```
alter table musicien
add constraint fk_musicien_origine
foreign key(origine)
references pays(code)
on delete cascade
```

Vidange de l'attribut : dans ce cas la suppression entraîne l'effacement des valeurs de `musicien.origine` pour chacun des tuples concernés. En Sql châtié on dira :

12. Nous n'indiquons ici que ce qui la concerne.

```
alter table musicien
  add constraint fk_musicien_origine
  foreign key(origine)
  references pays(code)
  on delete set null
```

On peut également noter ici que la « vidange » peut également consister à mettre la valeur par défaut◀ de l'attribut s'il en existe une. On écrira alors :

```
on delete set default
```

Mise à jour dans la table cible : il peut exister des situations dans lesquelles on sera amené à modifier le code du pays¹³. Sans intervention particulière le SGBD interdira cette opération puisqu'on se retrouverait alors avec des données non cohérentes (code modifié dans la table cible pays — par ex. 'FF' — et valeur inconnue — 'FR' — dans la table source). Si l'on souhaite que la modification dans la table pays se répercute dans la ou les tables qui y font référence, on écrira *au moment de poser la clé étrangère* :

```
alter table musicien
  add constraint fk_musicien_origine
  foreign key(origine)
  references pays(code)
  on update cascade
```



Finalement, plusieurs remarques ou propriétés concernant les clés étrangères qui, nous le savons, vous travaillent depuis que vous avez pris conscience de leur puissance :

- comme pour les clés primaires on a tendance à la métonymie¹⁴ à savoir : dire que `musicien.origine` est une clé étrangère, alors qu'on devrait dire qu'il s'agit d'un attribut sur lequel est posée une contrainte de clé étrangère ;
- une clé étrangère possède un attribut source et un attribut cible et il doit y avoir une contrainte d'unicité sur l'attribut cible (un attribut faisant office de clé primaire remplit donc cette condition) ;

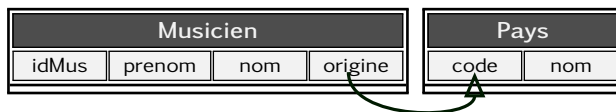
13. Peu probable je vous l'accorde, mais faites un effort d'imagination avec une autre table cible d'une clé étrangère.

14. Dire « boire un verre » ou « percer un trou », alors que je vous le rappelle, on boit le contenu du verre et on perce le mur.

- un attribut sur lequel est posé une contrainte de clé étrangère accepte de n'avoir aucune valeur. Dans notre exemple, ceci implique que la contrainte de clé étrangère n'impose pas qu'un individu doit avoir une origine, le champ `musicien.origine` peut donc être vide.

4.3.3 Représentation schématique

Il peut être nécessaire de se représenter schématiquement la notion de clé étrangère en imaginant qu'il existe un lien entre l'attribut sur lequel on a posé celle-ci et l'attribut cible. Ainsi pour notre exemple de `musicien` et d'`origine` on pourra « dessiner » les relations comme ceci :



Cette représentation graphique exprime le fait que les valeurs de l'attribut `musicien.origine` font référence à celles de `pays.code`. Elle exprime également l'idée que l'attribut `origine` « pointe sur » l'attribut `code`.

4

4.4 Traitement du vide : « c'est pas faux¹⁵ »

La notion de vide ou d'absence de valeur est spécifiée grâce à la valeur particulière **NULL**.



L'erreur des débutants qui ignorent l'existence de la valeur **NULL**, consiste à faire les choses suivantes :

- je ne connais pas la taille de cette personne donc je mets 0
- je ne connais pas le numéro de cette chose donc je mets -1
- je ne connais pas le tarif de cette prestation donc je mets 0
- je ne connais pas le truc de ce bidule donc je mets 9999

On verra que cette façon de faire peut mener à une exploitation très difficile des données et qu'elle est à fuir absolument.

L'algèbre relationnelle et le langage SQL introduisent donc une valeur particulière appelée **NULL** désignant le vide. Ce qui est très intéressant c'est que cette valeur est traitée dans les expressions booléennes et arithmétiques de manière un peu surprenante au premier abord. En effet on peut dire qu'en gros :

*Toute expression contenant **NULL** renvoie la valeur **NULL***

15. Kaamelott, Livre I.

Et qu'en outre la logique booléenne de SQL est « ternaire », une expression de cette **logique** pourra donc **renvoyer les trois valeurs** :

- VRAI
- FAUX
- NULL, pour « ni VRAI ni FAUX » ou « je ne sais pas »

Illustrons cela en utilisant la relation suivante :

Personne		
nom (chaîne)	prenom (chaîne)	taille (chaîne)
Moondog	•	160
Tampion	Tar	152
Tel	Un	•

Le symbole • indique ici visuellement l'absence de valeur. Si l'on demande au SGBD d'avoir la liste des personnes dont la taille dépasse 155 cm on devra écrire ¹⁶ :

```
select * from personne where taille>155
```

Cette requête — on le verra plus loin — renvoie une relation contenant tous les tuples de la table personne pour lesquels l'expression : **taille>155** est vraie, c'est-à-dire :

nom	prenom	taille
Moondog	•	160

En effet, l'analyseur effectuera les tests suivants :

- pour Moondog : **160 > 155** ⇒ VRAI
- pour Tampion : **152 > 155** ⇒ FAUX
- pour Tel : **NULL > 155** ⇒ NULL ⇒ ni VRAI ni FAUX

et ne renverra que les tuples pour lesquels l'expression renvoie VRAI. Invertissons maintenant la condition, pour obtenir les individus dont la taille est inférieure à 155cm :

```
select * from personne where taille<=155
```

L'analyseur effectuera les tests suivants :

- pour Moondog : **160 ≤ 155** ⇒ FAUX
- pour Tampion : **152 ≤ 155** ⇒ VRAI
- pour Tel : **NULL ≤ 155** ⇒ NULL ⇒ ni VRAI ni FAUX

16. Nous prenons ici un peu d'avance sur le volet *manipulation des données* de SQL présenté au chapitre suivant.

Et renverra donc uniquement le tuple :

nom	prenom	taille
Tampion	Tar	152

4.4.1 Ce qu'il ne faut pas faire

Imaginons maintenant ce qu'il se serait passé si un utilisateur non éclairé par ce brillant exposé avait saisi la valeur -1 pour les individus dont la taille n'est pas connue :

Personne		
nom (chaîne)	prenom (chaîne)	taille (chaîne)
Moondog	•	160
Tampion	Tar	152
Tel	Un	-1

Dans ce cas la requête permettant d'obtenir les individus de taille inférieure à 1,55 m aurait renvoyé des valeurs erronées puisqu'elle aurait inclus la personne nommée « Un Tel » de taille -1 (-1 comme vous le savez est inférieur à 155). Il faudrait donc, avec une telle configuration, écrire toutes les requêtes (y compris celles calculant des statistiques) en tenant compte que certaines valeurs (ici -1) ont une signification particulière. Ce qui deviendrait vite un casse-tête inextricable...

4.4.2 Ce qu'il ne faut surtout pas faire

Une autre erreur grossière serait, dans le cas de notre exemple sur les clés étrangères, de saisir, pour les individus dont l'origine est inconnue, une valeur « générique » à la place de **NULL**. Par exemple comme ceci :

Personne			
id	nom	...	origine
1	Zappa	...	US
2	Van Vliet	...	XX
3	Ponty	...	FR

Pays	
code	nom
US	United States
GB	United Kingdom
FR	France
XX	Pays inexistant

Ceci est une erreur grossière car toutes les requêtes dont le but sera d'extraire des informations liant les personnes aux pays seront fausses à moins de tenir compte de ce pays *qui n'existe pas* et dont le code est XX : il faudra par exemple penser lors du calcul du nombre de pays, à retrancher 1 au résultat, etc. Par conséquent :

S'il n'y pas d'information (ici l'origine),
ne pas l'inventer, mais **ne rien stocker**
c'est-à-dire **saisir NULL**

4.5 Comment identifier un tuple

Nous avons vu que chaque tuple devait pouvoir être identifié. C'est-à-dire qu'il doit exister dans la relation sous-jacente, un ou plusieurs attributs dont les valeurs seront uniques pour chaque tuple.

4 4.5.1 Clés primaires et candidates

Dans la relation suivante :

Personne		
prenom	nom	email

on peut faire l'hypothèse que chaque personne dispose d'un email qui lui est propre. C'est pourquoi on dira que l'email est une *clé candidate*, c'est-à-dire un attribut candidat pour l'identification des tuples. On peut en revanche imaginer que cet email ne soit pas disponible pour chacune des personnes, cet attribut ne peut donc faire l'objet d'une clé primaire ◀ qui exige l'existence des valeurs. En outre il est également concevable que le SGBD sera ralenti lorsqu'il aura à effectuer des opérations de tri et de recherche sur les personnes, puisqu'il devra comparer les chaînes de caractères composant les emails.

C'est la raison pour laquelle on a généralement recours à une clé artificielle dite de substitution (*surrogate*) permettant de garantir :

- un traitement rapide : on compare des entiers et non pas des chaînes de caractères ;
- l'existence systématique de la valeur et donc la possibilité d'être utilisée comme clé primaire.



D'autres arguments importants concernant l'utilisation des surrogate keys sont exposés en § 4.5.3 page 116.

Finalement pour l'exemple ci-dessus on créera la relation comme suit :

Personne			
id_P (entier)	prenom (chaîne)	nom (chaîne)	email (chaîne)

L'attribut id_P fera office de clé primaire. Et on pourra imposer une contrainte d'intégrité garantissant l'unicité des valeurs de l'attribut email, comme suit :

```
alter table personne
add constraint email_unique unique(email)
```

unicité Notez bien que cette contrainte *n'impose pas l'existence* de l'email. Il est uniquement stipulé ici : « *si l'email est renseigné, il doit être unique sur toute la relation* ».

Le paragraphe suivant montre comment il est possible de générer les valeurs des clés de substitution automatiquement.¹⁷

4.5.2 Générer des clés grâce aux séquences

Certains SGBD, dont Postgresql et Oracle, proposent un objet particulier appelé *séquence* dont l'objet principal est de générer automatiquement les valeurs d'un attribut clé primaire.



La lecture de ce paragraphe impose d'avoir préalablement compris comment on peut insérer des tuples dans une table. Si ça n'est pas votre cas vous pouvez lire le chapitre en question sur la manipulation des données, page 149. Vous pouvez également sauter ce paragraphe en première lecture.

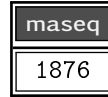


Les séquences sont des objets conçus pour s'inscrire dans le contexte des accès concurrents, c'est-à-dire les situations où plusieurs processus souhaitent accéder aux mêmes données en même temps. Nous montrons l'intérêt des séquences dans ce contexte, au paragraphe 6.7.2 page 259.

On peut voir une séquence comme une table constituée d'un tuple doté d'un seul attribut de type entier :

¹⁷. Quel suspense insoutenable...

voici une séquence :



Contrairement à une relation, une séquence ne réagit qu'à trois méthodes :

1. on peut l'initialiser avec la méthode `setval`;
2. on peut l'incrémenter avec `nextval`;
3. on peut en obtenir la valeur courante avec `currval`.

Il est important de comprendre qu'une séquence *n'est pas attachée* à une table, c'est à l'utilisateur de le spécifier.

Pour faire en sorte que la clé de substitution de notre table des musiciens puisse être générée automatiquement, on procédera d'abord à la création de la séquence :

```
create sequence seq_musicien;
```

le nom de cet objet étant bien entendu à la discrétion du concepteur de la base. Il est ensuite nécessaire de relier cette séquence avec la relation `musicien` :¹⁸

```
alter table musicien alter id_p
set default nextval('seq_musicien')
```

► § 4.6.2
p. 118

Cette incantation peut être traduite par : « *la valeur par défaut du champ `id_p` sera la valeur suivante de la séquence `seq_musicien`.* » Une fois ces deux opérations effectuées, on pourra insérer un nouveau tuple dans la relation `musicien` sans spécifier l'attribut `id_p` :

```
insert into musicien(nom,prenom,email)
values('Dolphy','Eric','dolphy@jazz.org')
```

Cette requête force le SGBD à faire appel à la valeur par défaut du champ `id_p`. Elle déclenchera donc un appel à la routine `nextval` qui renverra la valeur 1 pour le premier appel, 2 pour le suivant, etc. Si vous êtes curieux vous pouvez même tester vous-même ce que renvoie cette méthode, avec :

```
select nextval('seq_musicien')
```

Si jamais la relation `musicien` contenait déjà des données, il serait nécessaire d'initialiser la séquence à une valeur spécifique. Ceci peut être réalisé grâce à la commande suivante :

18. Avec Oracle, il est nécessaire de passer par un trigger...

```
select setval('seq_musicien',99)
```

Cette initialisation assurerait que la première insertion se fasse avec la valeur 100.



Une idée fausse couramment associée aux séquences est que la suite d'entiers générée par celles-ci est une suite d'entiers contigus. En réalité, on ne peut pas compter sur le fait que les entiers en question se suivent. Et ce, pour la simple et bonne raison que lorsqu'on procède à une insertion (par ex. valeur courante de la séquence : 128), le Sgbd incrémente la séquence (valeur 129), puis tente l'insertion. Si cette dernière échoue¹⁹ la séquence reste à sa valeur incrémentée. Par conséquent, lors de l'insertion suivante, à condition qu'elle ne génère pas d'échec, la valeur insérée sera 130 et la valeur 129 sera donc non utilisée.

Il faut par contre se persuader que cette « non-contiguïté » n'est pas un problème en soi, puisque l'objectif de départ est atteint : chaque tuple dispose d'un identifiant unique.

Pour finir avec les séquences, sachez qu'il est possible de régler plusieurs petites choses autour de celles-ci, notamment : la valeur de départ, le fait qu'elles puissent boucler ou non, le pas d'incrément, etc.



psql vous donnera quelques informations sur les séquences que vous avez créées. En particulier la liste de celles-ci grâce à :

```
yahozna=> \ds
```

Mais aussi, le détail d'une d'entre elles :

```
yahozna=> \d seq_musicien
          Sequence "public.seq_musicien"
   Column      | Type   | Value
-----+-----+-----
sequence_name | name   | musicien_idm_seq
last_value     | bigint | 4
...
increment_by   | bigint | 1
max_value      | bigint | 9223372036854775807
...
Owned by: public.musicien.idmus
```

19. Par exemple à cause d'une tentative de violation de contrainte.

Enfin l'attachement d'une séquence à un attribut d'une relation est visible lorsqu'on inspecte la structure de celle-ci, par exemple :

```
yahozna=> \d musicien
          Table "public.musicien"
  Column | Type          | Modifiers
  -----+-----+-----
  idmus  | integer       | not null
          |               | default nextval('seq_musicien')
  ...
```



Il est sans doute intéressant de noter que PostgreSQL peut vous épargner la création de la séquence ainsi que la définition de la valeur par défaut, grâce au recours au type `serial`.



Pour finir ce paragraphe sur les séquences, nous ajouterons que leur manipulation varie légèrement selon les Sgbd qui les proposent. On notera d'ailleurs la syntaxe d'Oracle est plus élégante, puisqu'on écrira :

```
select seq_musicien.nextval ...
```

4

4.5.3 Petit plaidoyer en faveur des surrogate keys

Nous avons introduit cette section sur l'identification des tuples en vous disant que l'email de la table ci-dessous :

Personne			
id_P (entier)	prenom (chaîne)	nom (chaîne)	email (chaîne)

bien qu'identifiant chaque tuple, *n'était pas un bon candidat pour devenir clé primaire*. Les raisons avancées étaient :

- l'éventualité de ne pas connaître l'email d'une personne ;
- la relative lenteur pour traiter une chaîne de caractères pour les recherches par exemple ;
- la taille du champ : sans clé de substitution cette taille sera répercutée dans toutes les relations qui feraient référence à la table personne.

Il y a deux autres raisons importantes qui nous poussent à opter pour la clé de substitution :

1. le fait que l'email d'une personne **peut changer**. Par conséquent toutes les relations qui feraient référence à une personne via une clé étrangère devraient être mises à jour en cas de modification de l'email ;
2. dans la mesure où une clé primaire crée implicitement un ►index, cet index sera d'autant plus volumineux que l'attribut sur lequel il s'appuie l'est aussi.

§ 7.3 ◀
p. 298

Pour terminer ce plaidoyer en faveur des clés de substitution, ajoutons que, de manière générale, une clé primaire **ne devrait jamais contenir d'information liée au tuple** à moins d'avoir la certitude absolue que cette information *ne sera jamais amenée à changer*.

4.6 Un peu plus loin avec les contraintes

Faisons un petit point sur les contraintes d'intégrité : jusqu'à maintenant nous en avons rencontré de plusieurs types :

- l'intégrité de *domaine* ;
- la clé *primaire* ;
- la clé *étrangère* ;
- la contrainte d'existence d'une valeur ;
- la contrainte d'*unicité*.

Et parmi les choses importantes à retenir au sujet des contraintes d'intégrité :

- elles sont à la charge du *concepteur* ;
- une fois posées elles sont *inviolables* ;
- on ne peut les poser que si les données déjà existantes dans la base, respectent les conditions des contraintes.

Il existe d'autres garde-fous proposés par la majeure partie des SGBD. Nous les passons en revue dans cette section.

4.6.1 Assurer l'existence d'une valeur

Nous avons vu au paragraphe 4.4 page 109 que les SGBD savaient traiter correctement la notion de vide. Il n'est pas donc pas étonnant qu'on trouve parmi les contraintes d'intégrité la possibilité de s'assurer qu'un attribut ne puisse pas être vide. Nous avons d'ailleurs déjà vu qu'une ►clé primaire assure entre autres choses, l'existence de la valeur.

§ 4.2.3 ◀
p. 100

Dans le modèle proposé au début de la section 4.3 page 104, nous indiquons qu'un musicien est originaire d'un pays et d'un seul.

En d'autres termes, les cardinalités mini et maxi 1,1 indiquent que pour toutes les occurrences d'un musicien est associée une et une seule (au moins une et au plus une) occurrence d'un pays. Ceci a été modélisé par la relation :

Personne			
id_personne	prenom	nom	origine

dans laquelle l'attribut *origine* est une clé étrangère pointant sur la clé de la table *pays*. Nous avons déjà noté lors de la présentation des clés étrangères que cette contrainte n'imposait pas l'existence d'une valeur. Ainsi, dans notre exemple, *rien n'empêche ce champ d'être vide*. En d'autres termes, dans l'état actuel des choses, un nouveau tuple peut être inséré dans la table *personne* avec un champ *origine* vide. Ce qui contredit le modèle conceptuel. C'est la raison pour laquelle pour implémenter de manière rigoureuse ce dernier on aura recours à l'incantation suivante :

```
alter table personne
    alter origine set not null
```

Si jamais vous changiez d'avis il vous suffirait de dire :

```
alter table personne
    alter origine drop not null
```

4.6.2 Valeur par défaut

Le concepteur d'une base peut imposer qu'un attribut d'une table ait une valeur par défaut. Cette valeur sera *automatiquement positionnée* dans les situations suivantes :

- au moment de l'insertion d'un nouveau tuple si la valeur de l'attribut n'est pas précisée ;
- au moment de la suppression en cascade si le comportement **on delete set default** a été choisi.

Dans PostgreSQL la commande permettant de définir une valeur par défaut est :

```
alter table musicien
    alter origine set default 'FR'
```

Ceci implique que tout musicien nouvellement inséré dans la relation se verra être rattaché à la France pour son origine, sauf si on précise son origine. Ainsi : ²⁰

20. La commande **insert** est détaillée au chapitre suivant.

```
insert into musicien(idmus,nom)
values (4,'Vander')
```

insère un individu dont le nom est 'Vander' sans préciser le pays.
Et :

```
insert into musicien(idmus,nom,origine)
values (5,'Coltrane','US')
```

insère un musicien en précision son origine. La table contiendra finalement :

Musicien			
idMus	nom	prenom	origine
...
4	Vander	•	FR
5	Coltrane	•	US
...

4.6.3 Vérifier que...

Le langage SQL propose une forme très générale de contrainte d'intégrité qui se traduit par « vérifier » que cet (ou ces) attribut(s) respecte(nt) telle condition. À titre d'exemple, dans notre relation contenant la taille des individus, on pourrait imposer que cette taille soit comprise entre deux bornes, par exemple 110 cm et 240 cm. Pour cela on écrira l'incantation suivante ²¹ :

```
alter table personne
add constraint taille_coherente
check ( taille >=110 and taille<=240 )
```

Une fois cette contrainte posée il sera impossible d'insérer un nouvel individu dont la taille serait hors de cet intervalle ou de modifier la taille d'une personne avec une valeur non compatible avec les bornes imposées.



Notons, qu'une contrainte check peut porter sur plusieurs attributs. On pourra par exemple écrire que la somme de deux attributs ne dépasse pas 100 :

```
add constraint bidule
check ( a + b < 100)
```

21. Attendre la pleine lune peut aider les sagittaires du 2^e décan.

4.6.4 Exemple d'intégrité de domaine : plaques minéralogiques



Un cas intéressant à étudier est celui des plaques d'immatriculation (vous pouvez passer directement au 4.7 page 124 si vous voulez souffler un peu). Il va nous permettre d'introduire plusieurs concepts incontournable du langage SQL. Nous proposons ici trois niveaux de mise en œuvre avec degré de complexité (et de souplesse) croissante.

Niveau 1 : « check like »

Supposons qu'on dispose d'une table permettant de stocker des véhicules et que cette table contienne un champ mémorisant l'immatriculation.

véhicule		
...	immat	...
(...)	(chaîne)	(...)

Une façon simple de conserver une certaine intégrité consiste à imposer que l'immatriculation contienne 2 tirets :

```
alter table vehicule
add constraint immat_valide
check (immat like '%-%-%')
```

Ceci mérite quelques explications :

- l'opérateur **like** compare deux chaînes de caractères et peut se traduire en « ressemble à » ou « comme »
- le caractère % a ici une signification spéciale et sera remplacé par n'importe quel ensemble de caractères (lettres, chiffres ou ponctuation) *y compris l'ensemble vide* (c'est-à-dire « aucun caractère »).

Finalement la contrainte d'intégrité énoncée ci-dessus en langage SQL peut se traduire en français en :

Vérifier que le champ (chaîne de caractère) immat est composé d'un ensemble de caractères suivi d'un tiret, suivi d'un ensemble de caractères, suivi d'un tiret, suivi d'un ensemble de caractères.

Cette contrainte permet donc de garantir une certaine cohérence dans les immatriculations mais est très limitée puisqu'elle « laisse passer » les fausses immatriculations de la forme :

- 123-ABCEF-FF
- -123-AAAA
- ---
- etc.

On pourrait améliorer ce filtrage en utilisant le caractère spécial '_' de l'opérateur **like** qui *s'apparie avec un caractère (lettre, chiffre, ponctuation) et un seul*. La contrainte aurait donc pu être posée comme ceci :

```
alter table vehicule
add constraint immat_valide
check (immat like '__-____-__')
```

Cette contrainte peut se traduire en français par :

Vérifier que le champ immat est composé
de 2 caractères suivis d'un tiret
suivi de 3 caractères, suivis d'un tiret
suivi de 2 caractères

Limitation de cette contrainte : le caractère _ ne permet pas de distinguer la famille de caractères, lettre ou chiffre en particulier. Ainsi, l'immatriculation 12-ABC-34 sera considérée comme une immatriculation alors que ça n'est pas le cas, il faut bien se rendre à l'évidence.



On retiendra que la langage SQL est doté d'un opérateur de comparaison de chaînes de caractères laissant la possibilité de détecter des motifs avec deux caractères spéciaux :

- % n'importe quelle chaîne y compris la chaîne vide
- _ n'importe quel caractère

Tous les autres caractères ont leur sens originel. Et les esprits tordus qui voudront créer un motif contenant le caractère % devront procéder à ce qu'on appelle dans le jargon des langages de programmation, l'échappement du caractère avec un \.

Niveau 2 : « check regexp »

Nous allons passer ici à un niveau supérieur de détection de motifs dans les chaînes de caractères puisque nous vous proposons d'utiliser les célèbres *expressions régulières* (traduction ²² littérale de

22. Une autre traduction est proposée par un groupe armé de libération des regexp (GALRE) : *expression rationnelle*. Une véritable guerre de religion s'est engagée entre le GALRE et le mouvement orthodoxe de soutien des regexp (MOSRE) qui lui, tente d'imposer la première traduction.

l'anglais *regular expression*). Cet outil bien connu dans le monde de la programmation est implanté, sous des formes légèrement différentes, dans des bibliothèques de plusieurs langages de programmation. Les expressions régulières se rapprochent de l'opérateur **like** vu plus haut au détail près qu'il n'y pas deux caractères spéciaux, mais beaucoup plus, avec la possibilité de constructions très complexes et donc très puissantes.

Nous proposons ici de vous montrer comment on peut filtrer à coup sûr la validité d'une immatriculation grâce à une « regexp ». Tout d'abord nous disons que l'immatriculation est constituée de 2 caractères :

[A-Z]{2}

le [A-Z] signifie *un* caractère parmi l'ensemble des 26 lettres de l'alphabet latin en majuscule. Le {2} qui suit s'applique à ce qui le précède (ici une lettre) et stipule qu'il en faut exactement 2. Nous ajoutons que ce qui suit doit être un tiret :

[A-Z]{2}-

Puis que la suite du motif doit être composée de 3 chiffres, pris cette fois parmi les chiffres arabes ([0-9] signifie *un* chiffre) :

[A-Z]{2}-[0-9]{3}

Un tiret en plus :

[A-Z]{2}-[0-9]{3}-

Puis les deux dernières lettres :

[A-Z]{2}-[0-9]{3}-[A-Z]{2}

Vous pensiez que cette expression était complète ainsi, et bien non. Il se trouve que dans l'état actuel des choses, elle est capable de répondre oui sur la chaîne suivante :

XYZ`AB-123-CD`XYZ

L'encadré ci-dessus veut illustrer le fait que le motif correspondant à l'expression régulière a été détecté « au milieu » de la chaîne fournie. Ce qui est pour le moins gênant dans notre cas, puisque la chaîne précédente n'est pas une immatriculation. Il faut donc spécifier dans notre expression, que la chaîne :

– *doit commencer par le motif défini* : caractère ^

- doit finir par le motif défini : caractère \$

Finalement, en langage SQL on définira la contrainte comme suit :

```
alter table vehicule
add constraint immat_valide
check (immat ~ '^[A-Z]{2}-[0-9]{3}-[A-Z]{2}$')
```

On notera l'utilisation de l'opérateur ~ en lieu et place de l'opérateur **like**, pour mettre en place la comparaison avec un motif de type expression régulière.

Niveau 3 : créer un domaine

Le dernier niveau d'implémentation consiste à faire un effort d'abstraction et imaginer que l'immatriculation est un peu plus qu'une chaîne de caractères : *c'est un domaine* à part entière. En d'autres termes, nous allons informer le SGBD qu'il existe un nouveau domaine (ou type de données) dont l'objet est de stocker des immatriculations.

```
create domain Timmat as text
constraint immat_valide
check (value ~ '^[A-Z]{2}-[0-9]{3}-[A-Z]{2}$')
```

Le nouveau domaine se nomme `Timmat`²³ et est basé sur le type chaîne de caractère de PostgreSQL (`text`). À la deuxième ligne ci-dessus, on précise le nom de la contrainte associée au domaine, ici `immat_valide` puis la contrainte elle-même. Finalement, au lieu de créer la table `vehicule` en deux temps :

1. créer l'attribut `immat` de type `text`
2. poser une des contraintes d'intégrité de niveau 1 ou 2 vues précédemment

on écrira directement :

```
create table vehicule( ..., immat Timmat, ...)
```

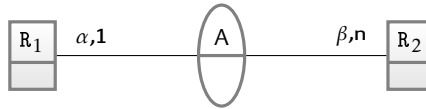
stipulant que la table `vehicule` contient un attribut de type `Timmat`.

23. Vous auriez pu l'appeler « Jean-Gontran » mais cela aurait rendu son utilisation peu claire.

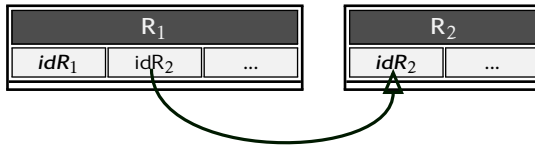
4.7 MCD \rightarrow MR suite : « plusieurs à plusieurs »

4.7.1 Un petit bilan sur « 1 à plusieurs » ...

Après toutes nos digressions sur les contraintes d'intégrité, il pourrait être nécessaire, de résumer la situation classique étudiée dans les paragraphes précédents à savoir une association de 1 à plusieurs :



Dans ce cas (et quelles que soient les valeurs des cardinalités α et β on construira les relations selon le principe qui suit :



On notera que :

- $R_1.idR_1$ est la clé primaire de la relation R_1
- $R_2.idR_2$ est la clé primaire de la relation R_2
- $R_1.idR_2$ est une clé étrangère faisant référence à $R_2.idR_2$
- si la cardinalité α vaut 1, il faudra imposer l'existence de la valeur pour l'attribut $R_1.idR_2$ grâce à une contrainte d'intégrité de type **not null** ◀

► § 4.6.1
p. 117



Dans ce type de configuration, on dit parfois que la clé de relation R_2 migre vers la relation R_1 , pour matérialiser l'association de 1 à plusieurs.



Enfin, on pourra discuter longuement sur le fait qu'en SQL, assurer la cardinalité $\beta = 1$ n'est pas simple à implémenter. Pour reprendre un des exemples illustrant l'association de 1 à plusieurs, il n'est en effet pas évident d'imposer qu'un pays ait nécessairement une personne : pour ce faire il faudrait interdire l'insertion d'un pays sans un individu originaire... Nous proposons au chapitre 6 une solution utilisant les contraintes dites « reportables. » ▶

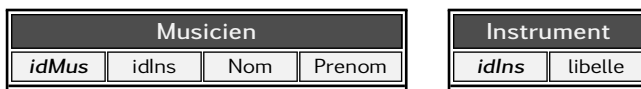
► § 6.7.6
p. 266

4.7.2 ... et on attaque « de plusieurs à plusieurs »

Reprenons pour cela, l'exemple du chapitre précédent :



On ne peut bien sûr pas envisager de contruire, comme dans le cas de 1 à plusieurs, la relation musicien en faisant migrer la clé de instrument vers la relation musicien :



Puisque cela imposerait qu'un musicien ne sait jouer que d'au plus un instrument. Certains parmi vous pourraient penser qu'une solution « à la tableur » résoudrait également le problème comme suit :



Même si elle peut paraître intéressante au premier abord, par le concepteur débutant, cette solution n'est pas une solution viable pour plusieurs raisons :

- la première, c'est que si un musicien sait jouer de 18 instruments ce modèle n'est pas adapté ;
- la deuxième, c'est que si au lieu d'un maximum de 4, nous avons spécifié 18, et que la majeure partie des musiciens ne maîtrise que deux ou trois instruments, alors le modèle serait « sur-dimensionné » et on peut imaginer que de l'espace serait utilisé inutilement dans la majeure partie des cas ;
- enfin la raison essentielle, c'est que le langage SQL ne permettra pas de manipuler²⁴ aisément l'ensemble des instruments. Cette solution est donc à proscrire impérativement.

Le raisonnement aboutissant à multiplier les clés étrangères sur la relation instrument dans la relation musicien pourrait être mené de la même manière en insérant plusieurs clés de musiciens dans la table instrument. Il conduirait bien évidemment aux mêmes conclusions.

24. Les jointures et les agrégations présentées au chapitre suivant, en particulier.

Finalement la solution adaptée consiste à créer *une troisième relation* pour matérialiser l'association sait jouer :

Musicien			saitjouer		Instrument	
<i>idMus</i>	Nom	...	<i>idMus</i>	<i>idIns</i>	<i>idIns</i>	libelle
4	Vander	...	4	12	10	Guitare
18	Zappa	...	4	11	11	Chant
			18	11	12	Batterie
			18	10		

Pour ce qui est des contraintes d'intégrité, on indiquera que les attributs `saitjouer.idMus` et `saitjouer.idIns` sont des clés étrangères pointant sur `musicien.idMus` et `instrument.idIns`, respectivement. Ceci garantira que la relation `saitjouer` ne fasse référence qu'à des musiciens et des instruments existants. Ces deux contraintes peuvent être écrites en SQL de la manière suivante :

4

```
alter table saitjouer
  add constraint fk_saitjouer_musicien
  foreign key(id_Mus)
  references musicien(idMus)
```

Et :

```
alter table saitjouer
  add constraint fk_saitjouer_instrument
  foreign key(id_Ins)
  references instrument(idIns)
```

Enfin pour garantir que l'on ne mémorise pas deux fois l'idée qu'un musicien sait jouer d'un instrument, on devra préciser quelle est la clé primaire de la table `saitjouer`. Les plus observateurs d'entre vous auront noté que :

- `saitjouer.idMus` n'est pas une clé candidate car un même musicien peut savoir jouer de deux instruments, dans ce cas son identifiant apparaîtra deux fois pour cet attribut ;
- de même, un instrument peut être maîtrisé par plusieurs musiciens, dans ce cas des valeurs de l'attribut `saitjouer.idIns` pourront être dupliquées.

Par conséquent la solution est de spécifier que *le couple* formé par les deux attributs est unique :

```
alter table saitjouer add constraint pk_sait
  primary key(idMus,idIns)
```

Voyons maintenant, pour finir l'étude du cas « plusieurs à plusieurs », si le modèle était le suivant :

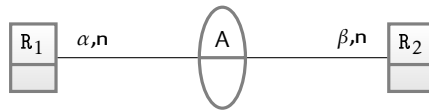


Dans lequel, l'association sait jouer possède maintenant un attribut niveau précisant quel est le niveau de maîtrise de l'instrument (par exemple un entier entre 1 et 5). Dans ce cas, la relation sait jouer devient simplement :

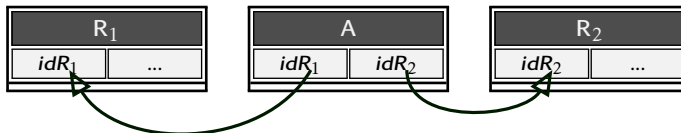
saitjouer		
idMus	idIns	niveau
4	11	1
18	11	3

4.7.3 Finalement, pour « plusieurs à plusieurs » ...

Dans le cas générique où on a une association du type



Dans ce cas (et quelles que soient les valeurs des cardinalités α et β on aura le schéma suivant pour les relations :



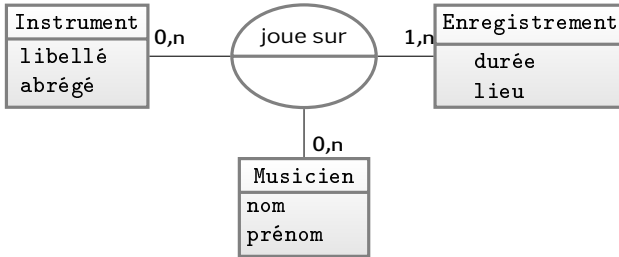
On notera que :

- $R_1 \cdot idR_1$ est la clé primaire de la relation R_1 ;
- $R_2 \cdot idR_2$ est la clé primaire de la relation R_2 ;
- le couple $(A \cdot idR_1, A \cdot idR_2)$ est la clé primaire de la relation A ;
- $A \cdot idR_1$ est une clé étrangère faisant référence à $R_1 \cdot idR_1$;
- $A \cdot idR_2$ est une clé étrangère faisant référence à $R_2 \cdot idR_2$;
- si l'association A possède des attributs, ils deviendront des attributs de la relation A

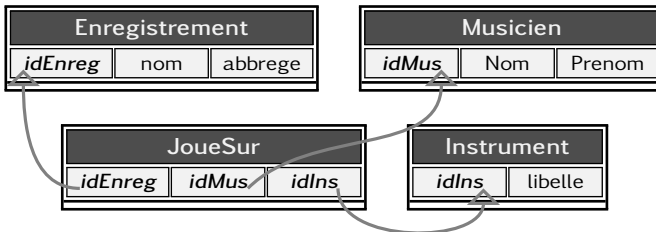
4.7.4 Autres cas de MCD → MR

Association ternaire

Nous avons rencontré au chapitre précédent (en particulier au paragraphe 3.5.6 page 89) une association ternaire :



Une telle association avec ces cardinalités se traduira en modèle relationnel par une table reliant les identifiants des trois relations correspondant aux trois entités (instrument, musicien, enregistrement). On aura donc une relation par entité :

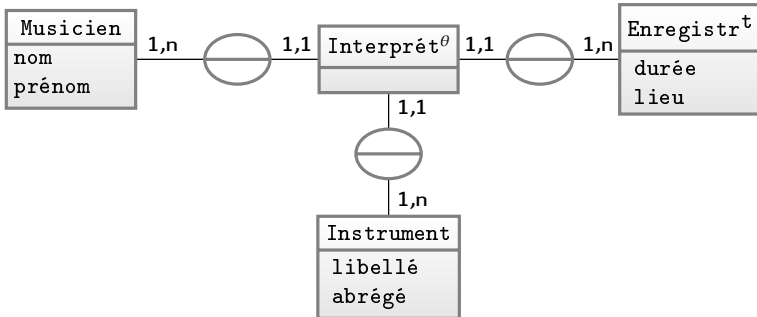


La clé primaire de la relation jouesur est une clé composite constituée des trois attributs. On pourra la poser comme suit :

```
alter table jouesur
add constraint pk_jouesur
primary key(idEnreg,idMus,idIns)
```

Autre approche de l'association ternaire

Comme indiqué dans le chapitre précédent (§ 3.3.3 page 77), l'association ternaire précédente pourrait tout à fait être modélisée légèrement différemment en considérant une entité « interprétation » :



Dans ce cas, le modèle relationnel n'est pas différent de ce qui est exposé plus haut. À savoir :



Dans le cas où une « interprétation » sera référencée via une clé étrangère depuis une autre relation, il faudra que la source de cette clé soit également composée de ces trois attributs. C'est la raison pour laquelle, pour simplifier notamment l'écriture des requêtes et en particulier les ►jointures, on pourra créer dans ce cas une clé de substitution pour la relation interpretation :



qui fera office de clé primaire et sera associée à une ►séquence. Pour conserver l'intégrité des données on pourra stipuler que le triplet est unique :

```

alter table interpretation
add constraint unique_interp
unique (idEnreg, idMus, idIns)
  
```

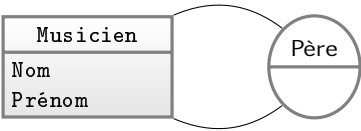
Il faudra également stipuler que chacun des trois attributs ne peut pas être vide, en écrivant pour idEnreg (à faire pour les deux autres) :

```

alter table interpretation
alter idEnreg set not null
  
```

Association réflexive

Une association réflexive vous a été présentée à la section 3.2.2 page 70. En voici une autre :



Notons que dans ce cas précis, chaque branche de l'association a une signification différente. La relation correspondante peut être conçue en ajoutant un attribut idPère :

```
alter table musicien
  add idPère references musicien(idMus)
```

Un extrait de la relation ainsi modifiée pourra ressembler à :

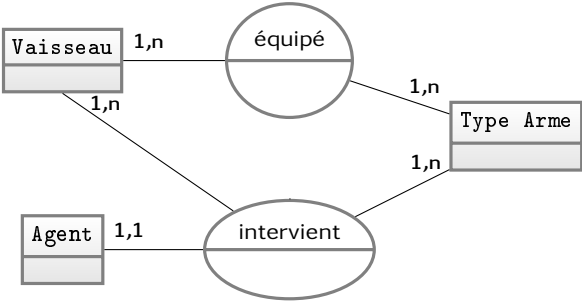
Musicien			
idMus	Nom	Prenom	idPère
3	Zappa	Frank	•
117	Zappa	Dweezil	3

L'attribut idPère se verra attribuer une contrainte de clé étrangère vers l'attribut idMus. Les tuples dont le père est inconnu auront la valeur null.

Le retour des vaisseaux



Au chapitre 3 traitant des MCD, vous était proposé un modèle conceptuel (§ 3.3.3 page 78) pour « l'intervention d'agent dans des vaisseaux équipés de types d'arme. » :

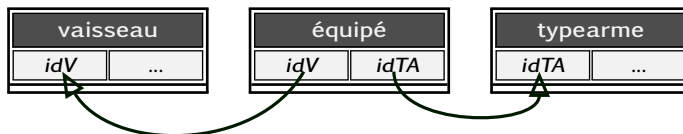


Nous tentons ici de montrer que le passage au modèle relationnel peut mener à deux solutions légèrement différentes du fait d'une précision absente dans le modèle conceptuel, à savoir, « *les agents interviennent sur les vaisseaux* » :

- avec un type d'arme équipant nécessairement le vaisseau sur lequel il intervient
- ou avec n'importe quel type d'arme. »

Notons, que c'est le formalisme utilisé pour le MCD (entité/association) qui ne nous permet pas de le préciser.

Dans un premier temps on peut imaginer que l'implémentation, dans le modèle relationnel, du fait que les vaisseaux sont équipés de types d'arme, nous donne :



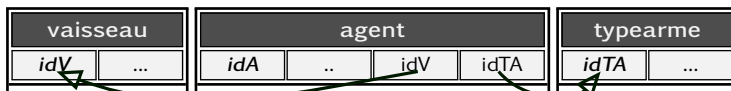
Avec bien sûr :

- vaisseau.idV et typearme.idTA, clés primaires ;
- le couple (équipé.idV, équipé.idTA) également ;
- équipé.idV et équipé.idTA clés étrangères pointant toutes les deux vers vaisseau et typearme, respectivement.

Et pour mémoriser en termes de relation, l'intervention d'un agent, on aura — selon les cardinalités de l'association intervient :



Sans y réfléchir plus avant, on pourrait naïvement proposer les clés étrangères suivantes :



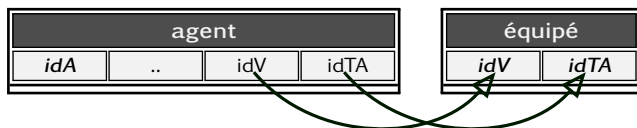
Cette configuration de clés étrangères assure effectivement que le vaisseau est valide, ainsi que le type d'arme. Par contre il ne garantit pas que le couple (idV, idTA) l'est. Pour s'assurer de cela, il faut que ce couple dans la table agent fasse référence à un des couples de la relation équipé. Pour y parvenir on écrira dans notre langage préféré :

```

alter table agent
add constraint equipt_valide
foreign key(idV,idTA)
references equipe(idV,idTA)

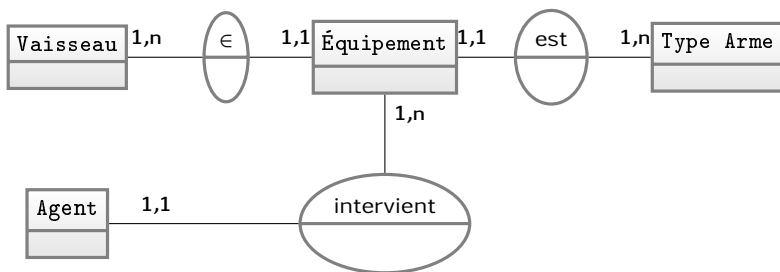
```

Ce qui schématiquement pourrait ²⁵ s'exprimer comme ceci :



Et pour terminer sur les vaisseaux on notera également que le modèle relationnel, intégrant cette dernière clé étrangère est une implémentation de la deuxième version du MCD proposé à la section 3.3.3 page 78, à savoir :

4



Dans lequel apparaissait implicitement, la contrainte « un agent intervient sur un équipement particulier du vaisseau ».

Ça passait, c'était beau... ²⁶

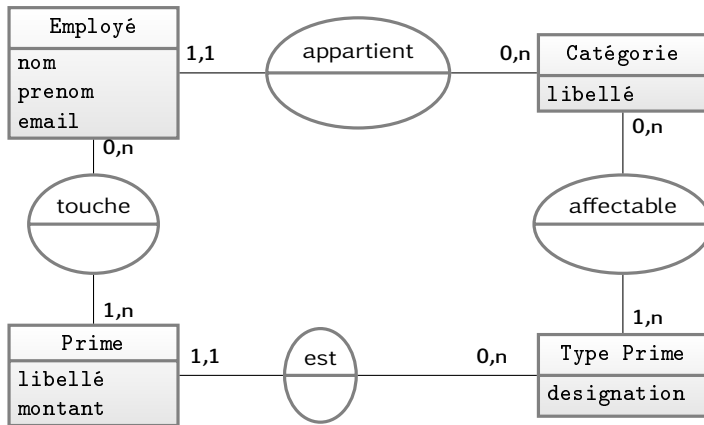


Pour rappel le modèle ci-dessous, étudié au paragraphe 3.4 page 80, répond à la question :

« On souhaite affecter des primes de types différents à des employés selon leur catégorie ».

25. Idéalement il ne faudrait qu'un seul arc, qui va d'un couple à un autre.

26. Joe Bar Team vol. 5 page 11.



En appliquant les différentes règles de conduite vues précédemment, sur le passage du MCD au modèle relationnel, on sera amené à construire les relations qui suivent (quelques tuples y sont ajoutés pour comprendre l'imbrication de chacune d'elles). Tout d'abord, la prime et son type, dans laquelle :

- prime.idP et typeprime.idTP sont les clés primaires
- prime.idTP une clé étrangère pointant sur typeprime.idTP

Prime			
idP	idTP	libelle	montant
1	FA	prime 1	10
2	SP	prime 2	30
3	SP	prime 3	15

Type Prime	
idTP	designation
FA	fraise d'Agatha
SP	salsepareille

Pour les employés et leur catégorie, on aura :

- employé.idE et catégorie.idC sont les clés primaires
- employé.idC pointe sur catégorie.idC

Employé		
idE	idC	nom
1	S	Farceur
2	B	Barbouille
3	B	Barbidur

Catégorie	
idC	libelle
S	Schtroumpf
B	Barbapapa

Et enfin, les tables suivantes mémorisent, à quel type d'employé on pourra affecter un certain type de prime (à gauche) et quelle(s) prime(s) on affecte aux employés (à droite) :

Affectable	
idC	idTP
S	SP
B	FA

Touche	
idE	idP
1	1
1	2
3	1

Dans ces deux tables (qui précisent que les schtroumpfs ont droit à la salspareille et les barbabapas aux fraises), on a, pour les clés étrangères :

- les attributs `affectable.idC` et `affectable.idTP` pointent sur `categorie.idC` et `typeprime.idP`, respectivement ;
- `touche.idE` et `touche.idP` font respectivement référence à `employé.idE` et `prime.idP`.



Les plus observateurs d'entre vous auront remarqué un défaut majeur de ce modèle relationnel : si l'on peut s'assurer — grâce aux clés étrangères de la table `touche` — que celle-ci contient effectivement un employé existant et une prime existante, il est impossible, dans l'état actuel des choses, de s'assurer que l'employé est bien autorisé à toucher cette prime. Dans nos données, l'employé 1 est un schtroumpf (c'est sa catégorie) et à ce titre il a le droit de toucher les primes de type SP, ce qui est le cas des primes 2 et 3, mais pas de la prime 1 qu'on lui a malgré tout affectée !

Une façon de résoudre ce problème serait d'ajouter dans la relation `touche` une contrainte d'intégrité référentielle qui exprimerait pour chaque tuple de cette table : : « la catégorie de l'individu est compatible avec le type de la prime ». Et ni la catégorie, ni le type, ne sont présents dans la relation `touche`. Nous pourrions donc ajouter dans celle-ci ces deux attributs :

Touche			
idE	idP	idC	idTP
1	1		
1	2		
3	1		

Le défaut de cette solution est qu'elle crée de la redondance dans nos données. Nous n'irons donc pas plus loin dans cette direction, mais nous proposons donc dans la section sur les triggers ◀ une solution technique permettant d'assurer cette intégrité référentielle.

4.8 Gestion des dates

Une date est un *type de données*, au même titre que les entiers, les nombres à virgule flottante, les chaînes de caractères, etc. À ce titre est associé à ce type, un ensemble d'*opérateurs* (comparaison, arithmétique en particulier).



L'erreur grossière des débutants ou des utilisateurs qui persisteraient à ignorer l'existence des types dates proposés par les Sgbd, consiste à s'escrimer à stocker ces dates sous forme de chaînes de caractères. Voici en vrac quelques faits qui devraient les en dissuader :

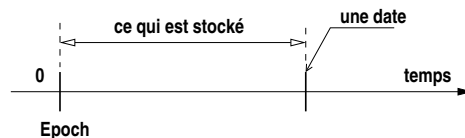
- "17/12/1980" serait classée avant "20/01/1924" lors d'un tri ;
- la durée entre les deux dates précédentes ne peut être calculée qu'au prix d'un effort important ;
- le calcul de la date "21/02/1974 + 8 jours" est tout aussi périlleux ;
- l'extraction, souvent nécessaire, du jour la semaine (lundi, mardi, etc.) correspondant au "27/10/1969" relève d'un exercice aussi difficile qu'inutile ;
- ...

4

4.8.1 Trois principes à distinguer

Pour comprendre le principe général des dates, on distinguera :

Le stockage interne : c'est la technique utilisée pour stocker l'information de type date en interne. Généralement, on mémorise un nombre de secondes écoulées depuis le début d'un calendrier :



Cette date se nomme généralement « epoch » et correspond au 1^{er} janvier 1970 à minuit sur un système Unix. Il est possible de voir ce nombre de secondes pour les valeureux utilisateurs du langage de commande Unix grâce à la commande `date` qui donne docilement la date et l'heure du jour :

```
mamachine:~$ date
Wed Feb 20 09:06:27 CET 2013
```

Alors que la commande :

```
mamachine:~$ date +%s
1361347588
```

L'affichage : c'est la manière de produire ou de formater pour l'utilisateur la représentation interne. Ainsi dans l'exemple ci-dessus, la *même date* est affichée de deux manières différentes. Les SGBD proposent tous des fonctions permettant de choisir de manière très souple un affichage pour une donnée de type date.

La saisie : on comprend bien qu'il n'est pas question de saisir un nombre de secondes (1361347751) pour indiquer qu'il s'agit du 20 février 2013 à 9h09 ! En revanche, il faut bien choisir un format de saisie ne serait-ce que pour lever les ambiguïtés : la date 10/04/2013 sera le 4 octobre pour un anglo-saxon et au milieu du mois d'avril pour un français.

4



Pour la saisie et l'affichage des dates, la technique utilisée consiste à passer de la représentation interne à une chaîne de caractères (pour l'affichage et l'inverse pour la saisie), en spécifiant un format à l'aide d'une syntaxe particulière que nous présentons plus loin.

4.8.2 Représentation interne

Les SGBD proposent des types divers et variés pour stocker les dates. Ces types portent parfois des noms différents d'un éditeur à un autre. Par contre tous disposent d'un type date dont la précision est la seconde, correspondant donc à l'organisation « classique » : nombre de secondes écoulées depuis le début d'un calendrier. Dans postgresql ce type se nomme `timestamp`.

Il existe également dans Postgresql un type dont la précision est le jour : `date`. Il pourrait être utilisé par exemple pour stocker les dates de naissance (et de mort le cas échéant) :

```
alter table personne add naissance date
alter table personne add deces date
```



Le Sgbd Oracle propose un type `date` dont la précision est la seconde et « l'unité implicite » est le jour. Ainsi, ajouter 0,5 à un attribut de ce type, lui ajoutera 12 heures.

4.8.3 Affichage : « to_char » est l'outil qu'il vous faut

Dans une requête `select` (voir le paragraphe 5.1 page 150), la fonction `to_char(...)` va nous permettre d'afficher une date avec une très grande souplesse.

Supposons qu'on ait la relation :

Personne				
id_personne	prenom	nom	naissance	deces
1	Zappa	Frank	1940-12-21	1993-12-04
2	Van Vliet	Don	1941-01-15	•
3	Ponty	Jean-Luc	1942-09-29	•

On pourra par exemple savoir quel jour de la semaine est né Frank ZAPPA²⁷.

```
select to_char(naissance, 'day') as jour
      from personne where nom='Zappa'
```

Donnera :

jour
saturday

De manière plus courante, il est fréquent d'avoir besoin de formater les dates avec des slashes comme ceci :

```
select
  nom,
  to_char(naissance, 'DD/MM/YY') as was_born
from personne
where ...
```

Ici aussi on utilise la fonction « à tout faire » `to_char` à laquelle on indique une chaîne de format (en guise de 2^e argument) pour indiquer la forme que prendra la date. La commande ci-dessus renverra :

nom	was_born
Zappa	21/12/40
Van Vliet	15/01/41
Ponty	29/09/42

27. Ce que vous vous demandiez déjà depuis un bon bout de temps, avouez-le...



Il existe bien entendu de nombreuses possibilités offertes par la fonction de formatage `to_char` et l'objet de ce manuel n'est pas de les documenter de manière exhaustive. On imagine cependant aisément que l'on peut faire apparaître au moment de l'affichage d'une date tous les champs nécessaires, de la micro-seconde au siècle, en passant par le numéro de la semaine, le mois en toutes lettres et autres joyeusetés...

4.8.4 Saisie : « `to_date` » est l'outil qu'il vous faut

Lors des opérations d'insertion ou de mise à jour (voir les paragraphes 5.2.1 page 171 et 5.2.3 page 174), il faudra avoir recours à une routine — en quelque sorte réciproque de la fonction `to_char` — permettant d'insérer une date à partir d'une chaîne de caractères. Ainsi on pourra simplement modifier la date de décès du Captain Beefheart²⁸ en indiquant :

```
update personne
set deces=to_date('17_12_2010','DD_MM/YYYY')
where ...
```

Ici on précise que la date saisie l'est sous la forme : jour (sur 2 chiffres) mois (sur 2 chiffres également) et année (sur 4 chiffres), les trois informations étant séparées pas des espaces.



Peut-être constaterez-vous un jour que Postgresql accepte docilement la requête suivante, qui ne fait pas appel à la routine `to_date` mais dans laquelle la date est fournie dans un ordre particulier :

```
update personne set deces='2010-12-17' where ...
```

Cet ordre (année, mois et jour séparés par des tirets) est normalisé par l'ISO (ISO 8601). Cette norme stipule également que si la date contient heure et minute, elle sera saisie « 2012-12-17 15:26:34 » pour le 17 décembre 2012 à 15 heures 26 minutes et 34 secondes.

4.8.5 Amusons-nous avec les fuseaux horaires

Imaginons que la finale du tournoi international de lancer de noyaux de cerises ait lieu le 25 octobre 2014 à 14h30 à New-York. Chacun souhaite mémoriser dans son agenda la date de cette rencontre. La relation utilisée pour cela est la suivante :

²⁸. Puisque depuis quelques lignes déjà, vous vous dites : « fichtre » Don Van Vliet nous a quitté en 2010, ça n'est pas ce qui est indiqué sur cette page 137...

```
create table rdv(  
    ...,  
    horaire timestamp with time zone,  
    objet text,  
    ...)
```

Vous aurez noté le type de l'attribut horaire indiquant la mémorisation des dates (précision en dessous de la seconde) *avec une indication de fuseau horaire* (c'est le sens du mot *time zone*). Avec une telle table il est possible d'insérer une date en indiquant le fuseau au format ISO :

```
insert into rdv(horaire, objet)  
values ('2014-10-25_14:30_America/New_York',  
        'Cerises')
```

En supposant que le système que vous utilisez ait comme fuseau horaire par défaut celui de Paris, ce dont vous pouvez vous assurer en tapotant la commande `show timezone` sur votre clavier²⁹, alors la commande suivante :

```
select horaire from rdv
```

devrait afficher un tuple contenant l'unique valeur :

```
2014-10-25 20:30:00+02
```

Que signifie donc tout ceci, allez-vous me dire ? L'explication est simple :

- nous avons saisi 14h30 à l'heure de New-York ;
- nous demandons l'affichage de la date, qui est effectué par défaut au format ISO et *dans le fuseau par défaut*, c'est-à-dire celui de Paris. Donc 20h30 ;
- en passant on notera le +02 après l'heure, qui est une forme normalisée pour indiquer le fuseau horaire, en l'occurrence « Zulu Time » plus 2 heures. « Zulu Time » étant une autre manière de dire GMT (*Greenwich Meridian Time*) ou UTC.

Un participant japonais pourrait avoir besoin de voir cette même date à l'heure de Tokyo :

```
select horaire at time zone 'Japan' from rdv
```

qui afficherait :

```
2014-10-26 03:30:00
```

29. Avec PostgreSQL, désolé pour les autres...

En d'autres termes, le Japonais désirant suivre la retransmission de la finale de cette rencontre devra allumer son poste de télévision le 26 octobre à 3h30 du matin.



Finalement, une date peut être produite dans n'importe quel fuseau à condition qu'on connaisse son fuseau initial. En d'autres termes, on ne peut pas savoir à quoi correspond 20h30 au Japon, si on ne précise pas dans quel fuseau « se situe » le 20h30 en question.

Zulu Time

Les esprits les plus alertes d'entre vous se seront sans doute posé la question pertinente suivante : « *on nous a bien précisé que la représentation interne des dates était le nombre de secondes écoulées depuis Epoch. Qu'en est-il des dates avec fuseau horaire ?* » En effet, si on y réfléchit, notre rencontre de lancer de noyaux de cerises est retransmise :

- le 25 octobre 2014 à 20h30 à Paris
- le 25 octobre 2014 à 14h30 à New-York
- le 26 octobre 2014 à 3h30 à Tokyo

et ces trois³⁰ dates représentent bien entendu le même instant³¹ ! Pour répondre à cette mystérieuse question, lançons la commande :

```
select date_part('epoch',horaire) from rdv;
```

qui renvoie 1414261800, c'est-à-dire la représentation interne de la date du 25 octobre 2014, 20h30 à Paris. Pour trouver à quel instant correspond ce nombre de secondes nous pouvons le demander à PostgreSQL en écrivant :

```
select '1970-01-01_00:00+00'::date
+ '1414261800_seconds'::interval
```

qui renvoie : 2014-10-25 18:30:00 qui est, je vous le donne en mille, la date de notre finale à l'heure Zulu, l'heure du méridien 0... On peut donc en conclure, et ceci est attesté dans les documentations de PostgreSQL, que la représentation interne d'une date avec fuseau horaire est le nombre de secondes écoulées depuis Epoch jusqu'à la date donnée en UTC.

30. Quitte à paraître un peu insistant, on pourrait ajouter autant de dates que l'on veut en choisissant des villes dans fuseaux horaires différents...

31. Même si au moment de la rencontre quelques Japonais se sont risqués à supplier les supporters américains et français de ne pas divulguer l'issue de la finale, étant donné qu'ils la verraient bien avant eux...

DayLight Saving

Supposons maintenant que l'on veuille connaître quelle sera la date dans chaque ville, 24 heures après le début de la finale. M'enfin³², la réponse est évidente... Pas forcément, comme vous allez le constater...

```
select (horaire + '24_hours')
       at time zone 'Japan'
from rdv
```

nous donne : 2014-10-27 03:30:00 ce qui semble logique,

```
select (horaire + '24_hours')
       at time zone 'America/New_York'
from rdv
```

nous donne : 2014-10-26 14:30:00, *so far so good*, par contre :

```
select (horaire + '24_hours')
       at time zone 'Europe/Paris'
from rdv
```

renvoie : 2014-10-26 19:30:00. Bref il est 19h30, 24 heures après le début de la rencontre à 20h30! Il y a quelque chose qui cloche. En fait rien « ne cloche » : le 25 octobre 2014 *est la nuit où les Français passent à l'heure d'hiver*. Ceci est lisible si on exécute la commande :

```
select horaire + '24_hours' from rdv
```

qui renvoie 2014-10-26 19:30:00+01. Le « +01 » indiquant un décalage d'une heure par rapport à GMT, au lieu de 2 la veille.

4.9 Tas d'octets

Si l'on fait le choix de stocker dans une base de données des *fichiers*, on aura recours à un type spécial appelé génériquement *Binary Large Object* ou *blob*. Ce type de données peut être vu comme un type dont le but est de stocker « un tas d'octets » quelle que soit l'information sous-jacente (musique, document texte, image, etc.) et sans que le SGBD ne dispose d'opérateur associé. En d'autres termes, s'il existe des opérateurs pour manipuler le type chaîne de caractères (text en PostgreSQL) — par exemple, trouver une sous-chaîne dans une autre — il n'existera pas d'opérateur pour

32. Je cite FRANQUIN de mémoire.

manipuler ces tas d'octets, dans la mesure où les SGBD ne sont pas conçus pour être dotés de fonctionnalités autour du format ogg, png et autres pdf.

Dans PostgreSQL ce type spécial « tas d'octets » est `bytea` et nous vous proposons un exemple d'utilisation d'un tel type : le stockage des enregistrements d'un morceau. À partir de la table suivante :

Enregistrement		
idEnreg (entier)	titre (chaîne)	musique (« tas d'octets »)
237	G-Spot Tornado	FD 48 58 0A B8...
44	Penguin in Bondage	AB 48 C8 34 FF...

créée par la requête suivante :

```
create table
  enregistrement(idEnreg int primary key,
                 titre text,
                 musique bytea)
```



Contrairement aux types natifs du Sgbd, le type `bytea` ne permet pas les requêtes classiques d'insertion, de modification ou de recherche de données — qui sont présentées au chapitre suivant. En tous les cas elles ne pourront être utilisées telles quelles. Si bien qu'on est contraint pour manipuler ce type de données de passer par une couche application, c'est-à-dire écrire un programme dans un langage de son choix pour prévoir l'insertion (généralement depuis un fichier) et l'extraction (généralement vers un fichier) de ce type de données. Cette technique est présentée au chapitre suivant à la section 5.6.2 page 202.

4.10 Épilogue : normalisation du modèle

En cherchant des informations sur « la meilleure façon » de concevoir le modèle relationnel pour votre système d'information, vous tomberez un jour ou l'autre, nez à nez avec ces bêtes étranges et inquiétantes que sont *les formes normales...*

Les formes normales sont des règles permettant de rendre robuste le modèle relationnel en garantissant avant tout **la cohérence des informations**. Chaque forme normale est « numérotée » et on dit qu'une relation « *est en n^e forme normale* » lorsque la règle *n* est respectée ainsi que les *n – 1* qui la précèdent.

Nous vous proposons ici de passer en revue quelques-unes de ces règles à partir d'exemples, puis de constater leur efficacité sur un modèle relationnel initialement très mal conçu.

4.10.1 Première forme normale (1FN)

La première forme normale stipule que :

- 1. chaque relation doit être dotée d'une clé ;
- 2. chaque attribut doit être atomique

Ces deux règles qui doivent maintenant vous sembler naturelles, ont déjà été énoncées au tout début du chapitre (§ 4.2 page 97) sans faire référence à la notion de forme normale.

4.10.2 Deuxième forme normale (2FN)

La deuxième forme normale concerne uniquement les relations dont les clés sont composées de plusieurs attributs. Cette 2^e forme normale stipule qu'en plus de respecter la 1FN, une relation sera en 2FN si aucun attribut *ne participant pas* à la clé *ne dépend que d'une partie* de la clé (ou en d'autres termes n'est déterminé que par une partie de la clé). Par exemple, si nous avons l'idée — saugrenue — de construire la relation *saitjouer* comme suit :

saitjouer		
idMus	idIns	nom
4	12	Vander
4	11	Vander
18	11	Zappa
18	10	Zappa

Nous pourrions dire que comme l'attribut (non clé) *nom dépend de*³³ ou *est déterminé par* l'attribut *idMus* (clé) — c'est-à-dire que la valeur 18 implique de manière univoque le nom "Zappa" — ou que comme *nom est fonction* de *idMus*, alors cette relation n'est pas en 2FN. Il faut donc modifier cette relation, en particulier il faudrait supprimer cet attribut pour le stocker dans une relation destinée à stocker les musiciens. Conclusion à laquelle nous étions arrivés via notre analyse conceptuelle sur les associations de ►plusieurs à plusieurs.

33. Il s'agit ici de *dépendance fonctionnelle*, notion que nous choisissons lâchement de ne pas développer.

4.10.3 Troisième forme normale

► § 4.3
p. 104

Une relation est dite en 3^e forme normale (3FN) si elle est en 2FN (et donc en 1FN) et lorsqu'aucun attribut non clé n'est déterminé par un attribut ne participant pas à la clé. Par exemple en partant de notre première analyse de l'association de 1 à plusieurs, nous avons proposé une relation comme celle-ci :

Musicien				
idMus	prenom	nom	code	pays
1	Zappa	Frank	USA	United States
2	Van Vliet	Don	USA	United States
3	Ponty	Jean-Luc	FR	France

Dans cette relation on peut constater que l'attribut pays (attribut non clé) est déterminé par (ou dépend de) l'attribut code (attribut non clé également). Par dépendance on entend par exemple que la valeur "FR" implique nécessairement la valeur "France" et elle seule. La relation n'est donc pas en 3FN et le remède est d'éclater (normaliser) cette relation en deux : une pour les musiciens et une pour les pays. Ceci vous a été « démontré » via l'approche conceptuelle dans le paragraphe mentionné ci-dessus.

4.10.4 Normalisons...

Un jour vous tomberez sur des données stockées dans des tableurs, qu'il faudra normaliser. En suivant les règles rencontrées jusqu'ici (passages du MCD au modèle relationnel, clé primaire pour chaque table, atomicité des attributs, etc.), vous arriverez naturellement vers un modèle relationnel cohérent. Un jour, disions-nous, vous tomberez donc sur ceci :

film		
titre	realisateur	acteurs
Husbands	Cassavetes John	John Marley, Fred Draper, Val Avery,...
Faces	Cassavetes John	Peter Falk, Ben Gazzara, John Cassavetes,...
Viridiana	Buñel Luis	Francesco Rabal, Fernando Rey, ...

Si, si je vous assure... Nous sommes convaincus, compte tenu des différentes analyses brillamment exposées dans ce chapitre et le précédent, que vous vous orienterez naturellement vers :

film			
idf	titre	nom	prenom
1	Husbands	Cassavetes	John
2	Faces	Cassavetes	John
3	Viridiana	Buñel	Luis

Ceci afin de respecter — sans que vous ne le sachiez — la première forme normale (clé primaire et attributs atomiques). Si on prenait ici la décision d’associer un identifiant à chaque réalisateur :

film				
idF	titre	idR	nom	prenom
1	Husbands	10	Cassavetes	John
2	Faces	10	Cassavetes	John
3	Viridiana	20	Buñel	Luis

On pourrait alors dire que nom et prenom (attributs non clé) sont fonction de l’attribut idR (attribut ne participant pas à la clé). La relation n’est donc pas en 3^e forme normale. Scindons-la donc :

film		
idF	titre	idR
1	Husbands	10
2	Faces	10
3	Viridiana	20

realisateur		
idR	nom	prenom
10	Cassavetes	John
20	Buñel	Luis

Concernant les acteurs, le seul moyen d’établir une relation entre les films et les acteurs n’est certainement pas de mémoriser une liste de noms et de prénoms dans un attribut (pas 1FN). Pour se mettre sur la voie :

film					
idF	titre	idR	idA	nom_act	prenom_act
2	Faces	10	210	Rowland	Gena
2	Faces	10	211	Cassel	Seymour
2	Faces	10	213	Marley	John
3	Viridiana	20	101	Pinal	Silvia
3	Viridiana	20	102	Rey	Fernando

La clé de cette relation dans son état actuel, est le couple (idF, idA). Cette relation n'est donc pas en 2FN car les attributs nom_act et prenom_act de l'acteur ne sont déterminés que par une partie de la clé (en l'occurrence idA). Scindons-la donc :

film				acteur		
idF	titre	idR	idA	idA	nom_act	prenom_act
2	Faces	10	210	210	Rowland	Gena
2	Faces	10	211	211	Cassel	Seymour
2	Faces	10	213	213	Marley	John
3	Viridiana	20	101	101	Pinal	Silvia
3	Viridiana	20	102	102	Rey	Fernandon

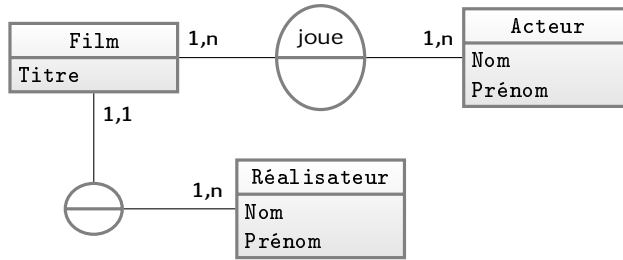
4

En gardant à l'esprit que la clé de la relation film est le couple (idF, idA), on peut ici aussi dire qu'elle n'est toujours pas en 2FN car l'attribut titre (non clé) est déterminé par une partie de la clé (en l'occurrence idF). Qu'à cela ne tienne, scindons-la :

film			joue		acteur		
idF	titre	idR	idF	idA	idA	nom	prenom
2	Faces	10	2	210	210	Rowland	Gena
			2	211	211	Cassel	Seymour
			2	213	213	Marley	John
3	Viridiana	20	3	101	101	Pinal	Silvia
			3	102	102	Rey	Fernandon



Nous avons donc montré ici que les formes normales nous ont permis de bâtir un modèle relationnel cohérent et sans redondance à partir d'un modèle initial en dehors des réalités des contraintes d'un Sgbd. Notons bien qu'avec l'approche conceptuelle présentée au chapitre 3 et les règles de passage d'un MCD au modèle relationnel exposées dans le présent chapitre, nous serions arrivés aux mêmes résultats :



Et maintenant ?

L'objet de ce chapitre a été d'exposer les principaux outils permettant d'agir sur la structure des données, outils rangés dans la boîte « langage de description de données » (*data description language*, DDL). Nous allons maintenant découvrir les outils de la boîte *data manipulation language* ou DML — en français **LMD**, pour langage de manipulation de données — **outils agissant sur les données elles-mêmes...**



- 5.1 Demander simplement
- 5.2 Modifier les données
- 5.3 Assembler : les jointures
- 5.4 Agréger
- 5.5 Combiner des requêtes
- 5.6 Utiliser des données externes

Manipuler les données

*Contrary to general belief,
an artist is never ahead of his time
but most people are far behind theirs.*

Edgard VARÈSE.

CE CHAPITRE aborde les outils du modèle relationnel et de SQL pour manipuler les données bâties conformément à ce qui a été exposé au ►chapitre précédent. Le volet de SQL présenté dans ce chapitre se nomme *langage de manipulation de données* (en anglais *Data Manipulation Language*). On verra qu'il se limite à quatre primitives suffisantes pour réaliser toutes les manipulations souhaitées sur les relations :

Chap. 4 ◀
p. 95

1. chercher avec **select**;
2. ajouter avec **insert into**;
3. modifier avec **update**;
4. effacer avec **delete from**.

Ce chapitre est l'occasion de présenter ces requêtes, puis d'exposer les deux grands classiques autour de la première (**select**), à savoir les *jointures* et les *agrégations*. Il est clos par quelques considérations sur la manière d'*exporter* ou d'*importer* des données depuis un fichier, puis par un exposé sur le moyen de d'interagir avec les données stockées sous forme de ►tas d'octets dans une table.

§ 4.9 ◀
p. 141

5.1 Demander simplement

La requête la plus célèbre du langage SQL est la requête **select** permettant d'interroger la base — *sans la modifier* — pour en extraire une partie des informations selon des critères définis par l'utilisateur. Elle a la forme suivante :

```
select <att1>,<att2>,...
from <table>
where <expression booléenne>
```

Cette requête renvoie, pour tous les tuples de la relation <table> respectant l'<expression booléenne>, les attributs stipulés dans la liste <att₁>,<att₂>,... Ainsi avec la table suivante :

Personne		
nom	prenom	origine
Zappa	Franck	US
Hendrix	Jimi	US
Wyatt	Robert	GB
Vander	Christian	FR

la requête :

```
select prenom, nom from individu
where not origine='US'
```

renverra la relation suivante :

nom	prenom
Robert	Wyatt
Christian	Vander

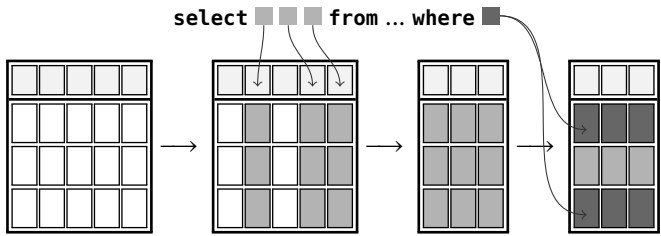


Si vous débarquiez ici sans avoir lu le chapitre précédent, sachez que l'interpréteur SQL ne fera pas de distinction entre minuscules et majuscules, ni pour les mots réservés du langage (par exemple **select**), ni pour le nom des objets que vous créerez pas plus que pour leurs attributs. Vous pourriez donc écrire :

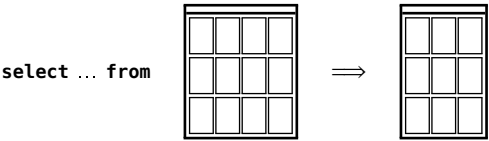
```
SELECT prenom, NOM From Individu
WHERE Not ORIGine='US'
```

Cette remarque ne s'applique bien sûr pas à la chaîne 'US'.

Finalement on peut représenter le rôle des clauses du **select** et du **where** comme suit :



❗ Contrairement aux requêtes rencontrées jusqu'ici, **select** renvoie une relation. Il s'agit d'un opérateur agissant sur une (ou plusieurs, on le verra plus tard) relation dont le résultat est une relation. En d'autres termes, au même titre qu'un opérateur arithmétique agit sur des opérandes qui sont des nombres et renvoie comme résultat un nombre, on peut voir la requête **select** comme un opérateur agissant sur des opérandes étant des relations et renvoyant un résultat dont la nature est une relation :



Dans le doute, et quitte à paraître quelque peu insistant : cette requête ne modifie pas les données de la base, en particulier, même si on dit ici qu'elle renvoie une relation, elle ne crée pas de relation.

5.1.1 Attributs sélectionnés

Dans la liste des attributs de la requête **select** on spécifie les attributs qui apparaîtront dans le résultat. Ainsi :

select prenom **from** personne

renvoie :

prenom
Franck
Jimi
Robert
Christian

Chaque attribut de la relation résultat peut être rebaptisé au bon vouloir de celui qui énonce la « phrase » Sql :

```
select prenom as petit_nom, origine as pays
from personne
```

Cette requête renverra :

petit_nom	pays
Franck	US
Jimi	US
Robert	GB
Christian	FR

En utilisant le caractère ***** dans la liste des attributs, on spécifie implicitement que l'on souhaite l'ensemble des attributs de la relation apparaissant après la clause **from**. Par conséquent :

```
select * from personne
```

renverra tous les attributs de la table *personne*. Enfin, comme il en a été question au chapitre précédent, on verra qu'il est parfois nécessaire (par exemple lorsqu'on utilise les jointures◀), de préfixer les attributs par le nom de la relation :

```
select personne.nom, personne.prenom
from personne
```

5

► § 5.3
p. 177

5.1.2 Filtrer

La clause **where** permet de *filtrer* les tuples extraits de la relation stipulée dans la clause **from**, en utilisant une expression booléenne :

```
select ... from ... where <expr. bool.>
```

où *<expr. bool.>* est constituée d'expressions logiques pouvant être connectées par :

- **and** pour le ET logique ;
- **or** pour le OU logique ;
- **not** pour la négation logique.



Même si cela paraît évident, on peut tout de même souligner que les attributs de la clause **where** ne sont pas nécessairement ceux de la clause **select**. Par exemple dans la requête :


```
select prenom from personne
      where origine='US'
```

qui signifie « donnez-moi le prénom des personnes dont l'origine est les USA », le filtre se fait sur le pays et seul l'attribut prénom est renvoyé...

5.1.3 Opérateurs et fonctions

Comme présenté au paragraphe 4.6.4 page 120, l'opérateur **like** permet de réaliser des filtrages basés sur des motifs simples et les caractères spéciaux :

- **_** qui s'apparie avec un caractère et un seul
- **%** qui s'apparie avec une chaîne *même vide*

Par exemple :

```
select nom from personne where nom like '%a%'
```

renvoie le nom de toutes les personnes dont le nom contient un 'a', y compris ceux commençant ou finissant par cette lettre. Dans la table ci-dessous :

Individu			
idP (entier)	nom (chaîne)	prenom (chaîne)	taille (entier)
1	Dupond	Jean-Luc	180
2	Dupont	Jean-Michel	164
3	Dupond	Jean	192
4	Petit	Jean-Pierre	179

la requête :

```
select prenom, nom from individu
      where prenom like '%-%'
      and nom      like 'Dupon_'
```

renvoie les noms et les prénoms des personnes ayant un prénom composé (contenant un tiret, c'est le sens du motif %-%) et se nommant 'Dupont', 'Dupond' ou tout autre nom constitué de 'Dupon' suivi d'un autre caractère (c'est le sens du motif Dupon_) :

prenom	nom
Jean-Luc	Dupond
Jean-Michel	Dupont




Tout SGBD est doté d'un certain nombre de fonctions agissant sur des nombres, des chaînes de caractères, des dates, et tout autre type prédéfini dans le système. Il faut lire la documentation du logiciel pour avoir une liste exhaustive de ces fonctions qui se comptent généralement par plusieurs dizaines pour chaque catégorie. Nous vous montrons ici, et tout au long de ce chapitre, comment peuvent être utilisées ces fonctions dans le cadre d'une requête SQL.

Si l'on souhaitait extraire les noms contenant la lettre « P » indépendamment de sa casse¹, il faudrait écrire une requête comme celle-ci :

```
select nom from personne
      where nom like '%P%' or nom like '%p%'
```

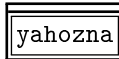
La fonction **upper** peut venir à notre secours :

```
select upper('yaHoZNa')
```

renvoie : . La requête :

```
select lower('yaHoZNa')
```

aurait bien entendu renvoyé :



Armés de ces outils nous pouvons réécrire la requête cherchant les noms contenant un « P » comme ceci :

```
select nom from personne
      where lower(nom) like '%p%'
```



Les vaillants utilisateurs de PostgreSQL auront quant à eux le plaisir de pouvoir faire appel à l'opérateur **ilike** fonctionnant comme l'opérateur **like** mais ne tenant pas compte de la casse.

La fonction **upper** pourrait également être utilisée. Voici un autre exemple d'appels de fonction, attention tenez-vous bien :

```
select upper(split_part(prenom, '-', 1)) as partie1,
      upper(split_part(prenom, '-', 2)) as partie2
from   personne
where  prenom like '%-i%'
```

1. Majuscule ou minuscule

Figurez-vous que ceci donne :

partie1	partie2
JEAN	MICHEL
JEAN	PIERRE

En effet la fonction **split_part** de PostgreSQL permet d'extraire des sous-chaînes en spécifiant un délimiteur (pour nous le tiret) et numérotant les sous-chaînes extraites à partir de 1 :

```
split_part(⟨chaîne à découper⟩,⟨délimiteur⟩,⟨n° de ss-chaîne⟩)
```

Un dernier pour la route :

```
select idp,
       cos(taille/100)*length(prenom) as vlunch
from personne
```

renvoyant bien évidemment :

idp	vlunch
1	4.32241844694512
2	5.94332536454954
3	2.16120922347256
4	5.94332536454954

Pour ceux qui l'ignoreraient, le coefficient de *vlunch* est un indicateur dans le domaine de partipltomanie donnant une idée assez bonne du pourcentage de flastipum dans l'organisme. Il est calculé en multipliant le nombre de lettres du prénom par le cosinus de sa taille. Mais revenons à la fonction elle-même :

- elle utilise la fonction **cos** qui attend un angle en radian ;
- ainsi que **length**, fonction renvoyant la longueur d'une chaîne, c'est-à-dire le nombre de caractères qui la composent ;
- les opérateurs ***** et **/** correspondent respectivement à la multiplication et à la division.

Terminons ce paragraphe par le célèbre opérateur de *concaténation*. Pour ceux qui auraient osé ouvrir ce merveilleux manuel sans savoir la signification de ce mot, nous prendrons la peine de préciser que l'opération de concaténation consiste à « coller » deux chaînes de caractères pour n'en faire qu'une seule. On peut utiliser cet opérateur — que l'on doit écrire **||** en SQL — pour créer les adresses électroniques de nos amis les Duponds et leurs potes :

```
select idp,nom,prenom,
       lower(prenom||'.'||nom)|| '@truc.org' as email
from personne
```

qui renverra :

idp	nom	prenom	email
1	Dupond	Jean-Luc	jean-luc.dupond@truc.org
2	Dupont	Jean-Michel	jean-michel.dupont@truc.org
...

Nous verrons un peu plus loin qu'il est possible d'utiliser une telle construction pour stocker véritablement l'email. Ici il est uniquement renvoyé au client par le SGBD.

5.1.4 Priorités des opérateurs booléens

Si l'on vous demandait : « parmi les musiciens américains ou français, quels sont ceux dont le prénom contient un "i" ? » Vous pourriez répondre naïvement :

```
select nom, prenom, origine from musicien
where origine='US'
      or origine='FR'
and prenom like '%i%'
```

ce à quoi le SGBD vous répondrait sans sourciller :

nom	prenom	origine
Zappa	Franck	US
Vander	Christian	FR
Hendrix	Jimi	US
Bozzio	Terry	US

Vous constaterez alors que les prénoms « Franck » et « Terry » ne contiennent pas franchement de « i » sauf un (grec) que votre SGBD préféré n'aura sans doute pas confondu avec un romain. L'explication est simple : l'interpréteur SQL ne combine pas les opérateurs **and** et **or** comme vous vous y attendriez. Votre requête est comprise de la façon suivante :

Donne-moi les nom, prénom et origine des musiciens qui sont :

- soit américain (whatever their first name)
- soit français avec un prénom contenant un "i"

En d'autres termes, c'est d'abord l'expression :

```
origine='FR' and prenom like '%i%'
```

qui est évaluée, puis l'expresssion **origine='US'**. Tout est donc parfaitement cohérent dans le résultat précédent. Quant au résultat que nous attendions, la requête :

```
select nom, prenom, origine from musicien
where ( origine='US' or origine='FR')
and prenom like '%i%'
```

permet de l'obtenir.



Dans le jargon des langages de programmation on dit que l'opérateur booléen **and** est prioritaire sur l'opérateur **or** (au même titre que l'opérateur arithmétique \times est prioritaire sur l'opérateur $+$).

5.1.5 Avec ou sans relation, avec ou sans attribut

Dans sa forme habituelle, la requête **select** attend une relation de laquelle seront extraites les informations. Il est cependant possible d'écrire :

```
select 4+3 as resultat
```

qui renverra un singleton en guise de tuple :

resultat
7



Le SGBD Oracle ne permet pas de faire un **select** sans ajouter une clause **from**. Pour arriver à effectuer la requête ci-dessous, on aura recours à une table spéciale appelée **dual**, dont la seule raison d'être est d'assurer une certaine cohérence dans la syntaxe de la requête :

```
select 4+3 as resultat from dual
```

Notons aussi la construction suivante incluant dans la clause des attributs, une constante, et non une expression dépendant d'un ou des attributs :

```
select 'M.' as civilite ,nom, prenom
from personne
```

qui renverra, pour notre table de musiciens :

civilite	nom	prenom
M.	Zappa	Frank
M.	Hendrix	Jimi
...

5.1.6 Trier (« ceci n’est pas une table », le retour)



Nous rappelons ici une notion importante déjà introduite au chapitre précédent, à savoir qu’une relation (une table) est un ensemble, c’est-à-dire un « tas » d’éléments non ordonnés, à ce titre :

select peut renvoyer les données dans n’importe quel ordre

En d’autres termes, si ça n’était pas clair :

select peut renvoyer les données dans un ordre différent
d’une fois sur l’autre

En faisant l’analogie « ensemble = sac contenant des éléments », on peut imaginer que **select** consiste à vider le sac, les éléments en sortiront dans un ordre quelconque.

C’est la raison pour laquelle le langage SQL a introduit une clause **order by** à la requête **select** permettant de trier la relation renvoyée selon un ou plusieurs attributs. Par exemple :

select nom, prenom **from** copain
order by nom, prenom

sur la table de gauche ci-dessous, renverra la relation de droite :

Copain		
idc	nom	prenom
1	Dupond	Zoé
2	Dupont	Jean
3	Dupond	André

nom	prenom
Dupond	André
Dupond	Zoé
Dupont	Jean

Il est possible d’imposer un sens au tri — qui est par défaut l’ordre croissant — avec le mot clé **desc** (pour décroissant).

Dans la requête suivante, on impose l'ordre décroissant pour les noms :

```
select nom, prenom
from copain
order by nom desc, prenom
```

Requête qui renverra donc :

nom	prenom
Dupont	Jean
Dupond	André
Dupond	Zoé

5.1.7 Chercher et manipuler des dates²

Nous proposons ici quelques exemples de manipulation de relation contenant des attributs de type date. Nous considérons la relation suivante :

Musicien				
nom	prenom	origine	naissance	deces
Zappa	Franck	US	1940-12-21	1993-12-04
Van Vliet	Don	US	1941-01-15	2010-12-17
Underwood	Ruth	GB	1946-05-23	
Bozzio	Terry	FR	1950-12-27	

Nous affichons ici la date au format ISO (voir le stockage des dates au chapitre précédent, paragraphe 4.8 page 135).



On notera également que les exemples proposés dans cette section reposent sur l'utilisation de PostgreSQL et en particulier du type `date` dont la précision est le jour et quelques fonctions qui n'ont pas forcément leur équivalent dans d'autres Sgbd³.

Combien de temps ont vécu les musiciens disparus ?

Voilà une question qu'elle est bonne. Pour y répondre :

```
select nom, ... from musicien
where deces is not null
```

2. Ne cherchez pas il n'y a pas de contrepèterie...

3. « Standard » qu'ils disaient...

nous permettra de ne sélectionner que les musiciens disparus, car le champ `deces` est renseigné. Puis, intuitivement nous pourrions écrire :

```
select nom, deces-naissance as temps
from musicien
where deces is not null
```

qui renverra la relation :

nom	temps
Zappa	19341
Van Vliet	25538

L'attribut `temps` contient bien sûr le nombre de jours écoulés entre la date de naissance et la date de décès pour chacun des musiciens. Il est tout à fait envisageable de diviser ce nombre de jours pour obtenir une durée en année comme suit :

```
select nom, (deces-naissance)/365 as annees
from musicien
where deces is not null
```

Qui donnera :

nom	annees
Zappa	52
Van Vliet	69

Les opérandes de la division étant tout deux des entiers le résultat est donné en entier. On peut éviter de réaliser une division entière en forçant l'un des opérandes en flottant, en écrivant :

- soit `(deces-naissance)::float/365`
- soit `(deces-naissance)/365.0`

qui donneront :

nom	temps
Zappa	52.9890410958904
Van Vliet	69.9671232876712

Cela n'est pas proprement lié à la manipulation des dates, mais la fonction, déjà rencontrée, `to_char` vous permettra de produire un affichage de ce nombre à virgule en imposant le nombre de chiffres avant et après la virgule :


```

select
    nom,
    to_char((deces-naissance)/365.24,'99D99')
from musicien
where deces is not null

```

Le motif 99D99 signifiant deux chiffres (deux 9) avant la virgule (D) et deux chiffres après. En passant on divise le nombre de jours par 365.24 puisque c'est le nombre de jours d'une année et on obtient donc :

nom	temps
Zappa	52,99
Van Vliet	69,97

À titre d'information, sachez qu'il existe une fonction *age* permettant de renvoyer la *différence* entre deux dates sous la forme de ce que PostgreSQL appelle un *intervalle* :

```

select nom, age(deces,naissance) as temps
from musicien
where deces is not null

```

qui renverra la relation⁴ :

nom	temps
Zappa	52 years 11 mons 14 days
Van Vliet	69 years 11 mons 2 days

Âge des musiciens vivants ?

Pour calculer l'âge actuel des musiciens il faut comparer leur date de naissance avec la date courante. Cette dernière peut être obtenue de différentes manières sous PostgreSQL :

- la fonction `now()` renvoie la date courante en seconde ;
- l'expression `current_date` renvoie la date courante en jour ;

Il est donc possible d'écrire quelque chose du genre :

```

select  nom,
        (current_date-naissance)/365.24 as age
from    musicien where deces is null

```

4. L'affichage par défaut sera en anglais, vous pouvez consulter le § 7.4 page 306 pour avoir une traduction en français.

À l'heure où cette partie du document a été écrite ⁵, cette requête a produit :

nom	temps
Bozzio	63.7262149212867899
Underwood	68.3230663928815880

Même remarque que précédemment concernant la fonction `age` : cette dernière, appelée avec un seul argument, vous renvoie le temps écoulé depuis celui-ci. Ainsi :

```
select nom, age(naissance) from musicien
where deces is null
```

qui donne :

nom	temps
Bozzio	63 years 8 mons 22 days
Underwood	68 years 3 mons 26 days

5

Âge des musiciens vivants ou non ?

Enfin, on peut se poser la question « *quel est l'âge des musiciens vivants, ainsi que celui des musiciens disparus au moment de leur décès ?* ». La fonction **coalesce** ⁶ de SQL va nous permettre de répondre à cette question. En effet, dans l'expression :

```
... coalesce(deces, current_date) ...
```


la fonction **coalesce** renverra le premier de ses arguments différent de **NULL**, donc dans ce cas précis la valeur de l'attribut `deces` si ça n'est pas le vide et **current_date** sinon. Finalement la requête ci-dessous répond à la question initialement posée :

```
select
    nom,
    age(coalesce(deces, current_date), naissance)
from musicien
```

5. 18 septembre 2014 à 16:58...

6. Cette fonction est également présentée dans le paragraphe sur les agrégations, page 189.

Qui a connu Claude, Igor et Edgard ?

 Si vous saturez sur les requêtes sur les dates, vous pouvez — en première lecture — ignorer cette section et vous rendre directement à la section 5.1.8 page 169.

Supposons que notre relation musicien contienne les entrées suivantes :

Personne			
nom	prenom	naissance	deces
Zappa	Franck	1940-12-21	1993-12-04
Van Vliet	Don	1941-01-15	2010-12-17
Underwood	Ruth	1946-05-23	
Bozzio	Terry	1950-12-27	
Parker	Charlie	1920-08-29	1955-03-12
Debussy	Claude	1862-08-22	1918-03-25
Varèse	Edgard	1883-12-22	1965-11-06
Stravinsky	Igor	1882-06-17	1971-04-06

5

Nous souhaitons savoir quels sont les musiciens susceptibles d’avoir rencontré de leur vivant ceux nés avant 1900. On suppose, pour l’exercice, que la rencontre est possible si : « *elle a lieu 5 ans avant le décès des personnes concernées et si celles-ci sont âgées d’au moins 16 ans*⁷. » Voyons étape par étape comment répondre à cette question délicate...

Tout d’abord, il faut être conscient que la question implique nécessairement un ►produit cartésien de la relation avec elle-même. § 5.3.1 ◀
C’est ce qui va nous permettre d’envisager toutes les rencontres p. 178
possibles. On peut donc d’ores et déjà écrire :

```
select m1.nom as nom1, m2.nom as nom2
      from musicien m1 cross join musicien m2
```

Si l’on exécute cette requête on aura 64 tuples (8 × 8) contenant toutes les combinaisons possibles de couples de musiciens, dont voici un extrait :

7. La seule raison d’être de cette contrainte est de vous montrer qu’on peut soustraire un intervalle à une date.

nom1	nom2
Underwood	Varèse
Varèse	Bozzio
Varèse	Van Vliet
Varèse	Zappa
Varèse	Underwood
Varèse	Varèse
Varèse	Stravinsky

On constatera avec effarement que VARÈSE peut se rencontrer lui-même et que sa rencontre potentielle avec Terry Bozzio apparaît deux fois. Pour se tirer de ce mauvais pas, il suffit d'ajouter un filtre sur le produit :

```
select m1.nom as nom1, m2.nom as nom2
  from musicien m1 cross join musicien m2
 where m1.idmus!=m2.idmus
       and m1.naissance<='1900-01-01'
       and m2.naissance>'1900-01-01'
```

5

Quelques remarques sur le filtre :

- **m1.idmus!=m2.idmus** évite qu'un musicien ne se rencontre lui-même ;
- le test **m1.naissance<='1900-01-01'** assure que le premier nom de la relation résultat ne concerne que les musiciens nés au XIX^e siècle ;
- le test **m1.naissance>'1900-01-01'** assure que le deuxième nom de la relation résultat ne concerne que les musiciens nés après le 1^{er} janvier 1900. *On règle ainsi également les doublons (X rencontre Y et Y rencontre X).*

Il reste maintenant à s'assurer que ces rencontres sont possibles, c'est-à-dire que les musiciens étaient effectivement vivants ! On utilisera pour cela l'opérateur **overlaps** attendant quatre arguments de type date. Un appel à :

```
overlaps(⟨début1⟩, ⟨fin1⟩, ⟨début2⟩, ⟨fin2⟩)
```

renverra la valeur booléenne « vrai » si les deux intervalles de temps définis par [⟨début₁⟩, ⟨fin₁⟩] d'une part, et [⟨début₂⟩, ⟨fin₂⟩] d'autre part, *se chevauchent*. On pourra donc écrire :

```

select
  m1.nom as nom1,
  m2.nom as nom2
from musicien m1 cross join musicien m2
where m1.idmus!=m2.idmus
  and m1.naissance<='1900-01-01'
  and m2.naissance>'1900-01-01'
  and overlaps (m1.naissance,m1.deces,
                 m2.naissance,m2.deces)

```

qui renverra une relation dont voici un extrait :

nom1	nom2
Varèse	Bozzio
Varèse	Van Vliet
...	...
Stravinsky	Bozzio
Stravinsky	Van Vliet
...	...

On imagine bien que ces rencontres potentielles seront fructueuses uniquement si les deux protagonistes sont suffisamment âgés⁸. C'est pourquoi on restreindra le chevauchement des intervalles comme ceci :

```

select
  m1.nom as nom1, m2.nom as nom2
from musicien m1 cross join musicien m2
where m1.idmus!=m2.idmus
  and m1.naissance<='1900-01-01'
  and m2.naissance>'1900-01-01'
  and
  overlaps (m1.naissance+'16_years'::interval,
             m1.deces-'5_years'::interval,
             m2.naissance+'16_years'::interval,
             m2.deces-'5_years'::interval)

```

Comme convenu, on impose ici que les personnes doivent au moins être âgées de 16 ans et que la rencontre ait lieu 5 ans avant le décès de celle-ci.

8. Même de la bouche d'un Frank ZAPPA un « dididididid » babillé par un enfant de 11 mois n'aurait pas intéressé Edgard VARÈSE



On notera donc qu'il est possible d'ajouter ou de soustraire une valeur à une date. PostgreSQL exige que le type de cette valeur soit un intervalle. On a ici recours au transtypage de la chaîne de caractère '16 years' en un intervalle en y ajoutant le nom du type (interval) précédé des deux caractères ::.

On obtient finalement :

nom1	nom2
Varèse	Van Vliet
Varèse	Zappa
Varèse	Parker
Stravinsky	Van Vliet
Stravinsky	Zappa
Stravinsky	Underwood
Stravinsky	Parker

On y trouvera confirmation du fait qu'Igor STRAVINSKY a entendu la musique de Charlie PARKER et que VARÈSE et ZAPPA se sont entretenus au téléphone.

5

Rien à voir : parcours dans un CV



Si vous saturez sur les requêtes sur les dates, vous pouvez — en première lecture — ignorer cette section et vous rendre directement à la section 5.1.8 page 169.

Supposons que l'on dispose d'une première relation indiquant pour chaque individu, identifié par l'attribut id, sa présence en tant qu'employé dans un établissement, entre une date de début et une date de fin (l'absence de valeur pour cet attribut indiquant qu'il s'y trouve encore) :

etablissement_pers			
id	deb	fin	designation
7	2012-04-24	•	BA217
23	2013-03-08	2015-10-13	BA115
7	2008-01-03	2012-04-23	BA107
23	2015-10-14	•	BA107
...

Un autre table, structurée de manière identique, permet de mémoriser pour chaque individu, l'évolution de sa carrière en termes de grade :

grade_pers			
id	deb	fin	designation
12	2011-12-01	●	Sergent chef
7	2008-01-03	2010-07-19	Lieutenant
7	2010-07-20	●	Capitaine
...



Si vous avez bien tout suivi jusqu'ici vous devriez vous offusquer que ces tables ne soient pas correctement normalisées. Cela ne dénature pas notre propos et nous imaginerons ici que ces relations sont créées à partir de ►vues.

§ 6.1 ◀
p. 208

L'objectif serait donc ici de constituer pour chaque individu une liste qui indiquerait pour chaque période, l'établissement et le grade. On aurait donc pour l'individu 7, quelque chose comme :

- 2008-01-03 à 2010-07-20 : *lieutenant* à la BA107 ;
- 2010-07-20 à 2012-04-23 : devient *capitaine* à la BA107 ;
- 2012-04-24 à **maintenant** : *mute* à la base BA217.

De manière générale, pour chaque individu, on a des changements d'établissement et de grade, complètement découplés :

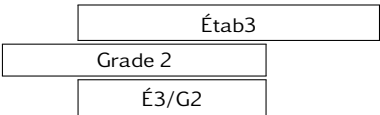
Étab1	Étab2	Étab3	Étab4
Grade1	Grade2	Grade3	

Dans le CV, on devrait faire apparaître un bloc avec date de début et de fin, à chaque fois qu'un des deux éléments (établissement ou grade) change :

É1/G1	É2/G1	É2/G2	É3/G2	É3/G3	É4/G3
-------	-------	-------	-------	-------	-------

La difficulté est donc de construire les blocs Ei/Gi ci-dessus, pour chaque personne. La construction des blocs ci-dessus peut être obtenue en remarquant que chacun d'eux :

- résulte d'un *chevauchement* entre un bloc Ei et un bloc Gj :



- commence à la date la plus *récente* des deux *débuts* de chaque bloc;
- finit à la date la plus *ancienne* des deux *fins* de chaque bloc.

Pour récupérer tous les chevauchements d'établissement et grade, on écrira :

```
select
    ...
from
    etab_pers e inner join grade_pers g
        on overlaps(e.deb,e.fin,g.deb,g.fin)
        and e.id=g.id
```

cette requête renvoie une ligne par chevauchement de bloc établissement et grade, pour une même personne (e.id=g.id). Il faut maintenant conserver de ce chevauchement :

- le plus grand de e.deb et g.deb
- le plus petit de e.fin et g.fin

► § 5.4
p. 186

Pour cela, on fera appel aux fonctions **greatest** et **least**⁹ (et non pas **max** et **min** qui sont des fonctions d'agrégations◀) :

```
select e.id,
    -- le plus récent des débuts :
    greatest(e.deb,g.deb) as deb,
    -- la plus vieille des fins :
    least(e.fin,g.fin)    as fin,
    e.designation as etablissement,
    g.designation as grade
from
    etab_pers e inner join grade_pers g
        on overlaps(e.deb,e.fin,g.deb,g.fin)
        and e.id=g.id
```



Il reste bien entendu un peu de travail pour présenter toutes ces informations à une application. On pourra par exemple faire appel à la routine **coalesce** pour traiter les cas des valeurs de fin à NULL, par exemple comme ceci :

```
least(e.fin,coalesce(f.fin,now())) as fin
```

Il faudra également utiliser la fonction **to_char** pour ne présenter que le mois et l'année, par exemple.

9. Ne faisant pas partie du standard SQL, mais assez répandues malgré tout.

5.1.8 Recherche sur les chaînes de caractères

Vous aurez peut-être un jour besoin de comparer des chaînes de caractères de manière « approchée ». Sachez qu'il existe plusieurs algorithmes pour comparer des chaînes, la plupart d'entre eux étant souvent basés sur l'une ou l'autre des méthodes :

- calcul d'une distance : ainsi, si la distance entre deux textes est faible alors on considérera qu'ils sont proches ;
- calcul d'un motif « phonétique » d'une chaîne à l'aide d'une fonction. Ainsi, si deux chaînes ont le même motif au travers de cette fonction, on peut dire qu'elles se ressemblent.



À condition d'installer le module `fuzzystrmatch` de votre PostgreSQL préféré, vous serez en mesure d'utiliser trois algorithmes de comparaison « floue » (c'est le sens de fuzzy match) : `soundex`, distance de Levenshtein et `metaphone`. Pour l'installation :

```
create extension fuzzystrmatch
```

doit suffire pour PostgreSQL à partir de la version 9.

Voici quelques exemples d'utilisation de telles fonctions. Pour chercher les personnes dont le nom ressemble à « trucmuche » on pourrait lancer la requête suivante :

```
select *  
from personne  
where  
    levenshtein(nom, 'trucmuche')<=1
```

Ici on cherche les personnes pour lesquelles la distance de Levenshtein entre leur nom et « trucmuche » est faible. Avec les algorithmes basés sur les motifs on écrirait :

```
select *  
from personne  
where  
    soundex(nom)=soundex('trucmuche')
```



Les algorithmes mentionnés ici sont basés sur une ressemblance de prononciation. Il faudra donc veiller à ce que les algorithmes utilisés soient bien adaptés au français et non uniquement à l'anglais.

5.1.9 Outils divers

Dédoublonner

Il est parfois nécessaire de « dédoublonner » les tuples résultant d'une requête, c'est-à-dire fusionner dans le résultat, les tuples ayant mêmes valeurs d'attributs. Si on veut par exemple avoir la liste des prénoms des musiciens, on pourra écrire :

```
select distinct prenom from musicien
```

Sans le mot clé **distinct**, cette requête renverra autant de Barnabé et de Gontran que de musiciens se prénommant ainsi. Avec ce mot clé chaque prénom n'apparaîtra qu'une fois dans le résultat.



Si en plus vous souhaitez faire des opérations spéciales sur chaque ensemble de tuples fusionnés (agrégés devrais-je dire), nous ne pouvons que vous inviter à lire le lumineux paragraphe sur les agrégations...

► § 5.4
p. 186

Restreindre les tuples

PostgreSQL permet de limiter le nombre de tuples renvoyés par un **select** :

```
select ... from ... limit 10
```

limitera à 10 le résultat de la requête. De plus, une construction du type :

```
select ... from ... limit 10 offset 5
```

renverra 10 tuples à partir du 6^e (décalage de 5 tuples).



Les mots clé **limit** et **offset** sont des particularités de PostgreSQL. Le Sgbd Oracle propose de son côté la fonctionnalité suivante :

```
select ... where rownum>5 and rownum<=7
```

permettant de ne conserver que les tuples dont le « numéro de ligne¹⁰ » (row number) est compris entre 5 et 7. NB : pour restreindre un **select** contenant un **order by**, il faudra imbriquer deux requêtes :

```
select ... from (
  select ... from ... order by ...
)
where rownum ...
```

10. Les **rownum** d'Oracle commencent à 1.

5.2 Modifier les données

Pour reprendre l'analogie entre une relation en tant qu'ensemble, avec un sac de billes, le **select** permet de vider le sac de tout ou partie des éléments. Nous examinons maintenant les trois primitives SQL prévues pour modifier les données, à savoir :

1. **update** pour mettre à jour un ou plusieurs tuples — modifier chacun des éléments ;
2. **delete from** pour effacer un ou plusieurs tuples d'une relation — retirer des éléments du sac ;
3. **insert into** pour ajouter un ou plusieurs tuples à une relation — ajouter des éléments dans le sac ;

On peut d'ores et déjà noter que les deux premières requêtes peuvent toutes les deux être agrémentées d'une clause **where** pour préciser quel(s) tuple(s) sont affectés. Cette clause suit la même syntaxe que celle de la requête **select** présentée plus haut :

update ou **delete** ... **where** *<expression booléenne>*

dans laquelle l'expression booléenne suit les mêmes règles que celle de **select**

5

5.2.1 Mettre à jour

Pour *mettre à jour*, c'est-à-dire modifier un ou plusieurs tuples on écrira :

```
update <T>
  set <att1>=<val1>, <att2>=<val2>, ... , <attn>=<valn>
  where <expression booléenne>
```

pour dire :

- qu'on a l'intention de modifier la table <T>;
- que l'attribut <att_i> prendra comme nouvelle valeur <val_i>;
- et ceci pour chaque tuple renvoyant VRAI pour l'<expression booléenne>.

Nous tentons ici de vous fournir quelques exemples pour insister sur des aspects notables de cette construction.

On s'appuie ici sur la relation suivante :

personne		
idp	nom	prenom
1	Thempion	Tar
...

Tout d'abord une forme simple où un seul tuple est précisé :

```
update personne set nom='Thempion' where idp=1
```

On remarquera que si `idp` est une clé primaire, on est sûr de n'agir que sur un seul tuple, puisque par définition il n'y a qu'un tuple pour lequel l'attribut `idp` a la valeur 1.



Notez bien que si « par malheur » dans un moment d'égarement vous oubliez la clause **where** :

```
update personne set nom='Thempion'
```

l'attribut se verra affecter la valeur « Thempion » pour tous les tuples de la relation ! Ceci n'est pas nécessairement ce que vous vouliez au départ, mais peut s'avérer utile dans certaines situations, par exemple :

```
update personne set nom=upper(nom)
```

met tous les noms en majuscules.

Voici maintenant une particularité de **update** qui va à l'encontre de ce qu'un programmeur pourrait penser. Soit la table suivante :

T	
X	Y
1	3
2	4

Alors, la requête :

```
update T set x=y, y=x
```

inverse effectivement les valeurs des deux colonnes (table de droite ci-dessous). Si vous ne voyez pas pourquoi il en serait autrement, sachez que dans un programme qui exécute des instructions séquentiellement, on aurait d'abord affecté les valeurs de l'attribut `y` à la

colonne x en écrasant l'existant, puis affecté les valeurs de l'attribut x (contenant les valeurs initiales de y) à y, pour se retrouver finalement avec la relation de gauche ci-dessous :

T	
X	Y
3	3
4	4

T	
X	Y
3	1
4	2

Nous vous proposons ici un petit exemple d'utilisation de la requête **update** faisant appel à une des relations données précédemment en exemple :

Individu			
idP (entier)	nom (chaîne)	prenom (chaîne)	email (chaîne)
1	Dupond	Jean-Luc	
2	Dupont	Jean-Michel	
3	Dupond	Jean	
4	Petit	Jean-Pierre	

On peut utiliser la requête **update** pour définir l'email de chaque individu, en le constituant comme expliqué au paragraphe 5.1.3 page 153 :

- le prénom suivi d'un point suivi du nom, le tout en minuscule
- le tout collé à la chaîne @jazz.org

En deux coups de cuillère à pot :

```
update individu
set email=lower(prenom||'.'||nom)||'@jazz.org')
```

5.2.2 Effacer

Pour effacer définitivement des tuples d'une relation on utilisera :

```
delete from <T> where <expression booléenne>
```

Par exemple la requête :

```
delete from musicien where prenom='John'
```

détruit les tuples pour lesquels le prénom est « John ».

Attention, sans la clause **where**, cette requête efface *tous les tuples* :

```
delete from musicien
```

5.2.3 Ajouter

Pour ajouter des tuples dans une relation $\langle T \rangle$ on pourra :

1. soit ajouter des valeurs :

```
insert into  $\langle T \rangle$  (...) values (...)
```

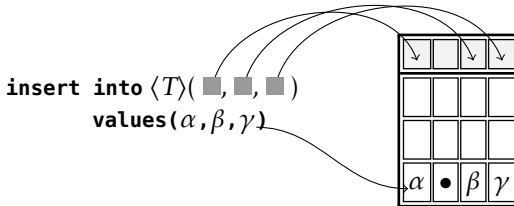
2. soit ajouter des tuples provenant d'une autre relation :

```
insert into  $\langle T \rangle$  (...) select ...
```

Dans les deux formes (mot-clé **values** ou **select**), **insert into** attend, dans les parenthèses qui suivent le nom de la table, la liste des attributs qui recevront effectivement une valeur :

```
insert into  $\langle T \rangle$  ( $\langle attr_1 \rangle$ ,  $\langle attr_2 \rangle$ , ...) ...
```

On peut aussi lire une requête d'insertion comme ci-dessous :



On notera donc qu'il doit y avoir une correspondance en nombre et en type entre les attributs spécifiés après la table $\langle T \rangle$ et ceux de la relation ajoutée, que ce soit par un **values** ou par un **select**.

Voici quelques exemples se basant sur la relation *copain* vue précédemment et rappelée à la figure 5.1 page suivante. Pour ajouter un nouveau copain, on écrira :

```
insert into copain(idc,      nom,  prenom)
values      ( 4, 'Durand', 'Luc')
```

Il est bien sûr possible de ne spécifier qu'une partie des attributs (à condition bien sûr de respecter les contraintes d'intégrité) :

```
insert into copain(id,nom) values(5,'Dupont')
```

Copain		
idc	nom	prenom
1	Dupond	Zoé
2	Dupont	Jean
3	Dupond	André

(a) État initial

Copain		
idc	nom	prenom
1	Dupond	Zoé
2	Dupont	Jean
3	Dupond	André
4	Durand	Luc
5	Dupont	•

(b) Après *insert* forme 1

Copain		
idc	nom	prenom
1	Dupond	Zoé
2	Dupont	Jean
3	Dupond	André
4	Durand	Luc
5	Dupont	•
21	Hendrix	•
23	Mitchell	•

(c) Après *insert* forme 2

FIGURE 5.1 – Évolution d’une relation après des opérations d’insertion

dans ce cas l’attribut prenom recevra la valeur NULL si aucune valeur par défaut n’a été définie pour celui-ci. Pour finir avec la première forme du **insert into**, sachez qu’on peut multiplier le nombre de tuples insérés en répétant ceux-ci entre parenthèses :

```
insert into copain(id,nom)
values(8,'Tar') (9,'Tampion') (7,'Hop')
```

Pour la deuxième forme, il faut comprendre qu’on peut insérer dans une relation les tuples renvoyés par un **select**. Comme indiqué plus haut, ici aussi, les tuples insérés doivent avoir le même nombre d’attributs et le type de ceux-ci doit être compatible avec la relation dans laquelle on insère. On pourra par exemple écrire ceci :

```
insert into copain(idc,nom)
select idp,nom
from musicien where idp>10
```

Cette requête insère dans la relation *copain* les tuples de la relation *musicien* pour lesquels l'attribut *idp* est supérieur à 10. Le résultat pourrait être celui montré à la figure 5.1c page précédente

5.2.4 Introduction à l'atomicité

Une caractéristique des SGBD, nommée *l'atomicité* des requêtes, peut s'énoncer simplement en disant qu'ils font **tout** ce qu'on leur demande **ou rien**. Nous retrouverons au chapitre 6.7 page 258 cette notion qui est une des composantes importantes des accès concurrents. Pour l'instant nous allons examiner quelques cas de figure illustrant ce mécanisme qui fait la force des SGBD. Imaginons dans un premier temps que nous travaillons sur la relation ci-dessous

T	
id	lib
1	ABC
2	DEF
3	abc

Faisons l'hypothèse que le champ *T.lib* soit doté d'une contrainte d'unicité, en d'autres termes que le concepteur de cette relation aurait lâchement écrit :

```
alter table T add constraint libelle_unique
unique(lib)
```

Puis que la requête suivante soit envoyée au système :

```
update T set lib=lower(lib)
```

dont le but serait de mettre tous les « libellés » en minuscules. Bien évidemment cette requête enfreint la contrainte posée sur l'attribut, le SGBD refuse donc de mettre à jour les tuples concernés (qui sont au nombre de trois). Même si les deux derniers tuples pourraient être mis à jour, puisqu'aucune contrainte n'est enfreinte pour eux, le SGBD **refuse de faire la mise à jour car au moins un tuple viole une contrainte**.


Autre exemple, avec les deux tables liées par une clé étrangère de *personne.origine* vers *pays.code*, vues au chapitre précédent (cf. § 4.3 page 104) :

Personne				Pays	
id_personne	nom	...	origine	code	nom
1	Zappa	...	US	US	United States
2	Van V.	...	US	GB	United Kingdom
...

Supposons que l'on veuille supprimer tous les pays :

```
delete from pays
```


ce qui peut paraître une idée saugrenue, mais telle n'est pas la question. Supposons également que la clé étrangère de la table personne ait été posée en mode « no action » : c'est-à-dire que le SGBD interdira la suppression d'un tuple de pays si un enregistrement de la table source y fait référence. Dans notre cas on ne pourra supprimer les États Unis, par exemple. Il est donc important de noter que la requête demandant la suppression de tous les pays échouera dans la mesure où *au moins* une opération n'est pas réalisable. Par conséquent, l'atomicité implique ici qu'*aucun pays ne sera supprimé*, même si la majeure partie d'entre eux aurait pu l'être.

 Finalement lorsque vous demandez aux Sgbd d'exécuter une requête qui modifie les données, ce dernier s'exécutera poliment sauf si au moins un tuple viole une contrainte. Ceci est à opposer au comportement que vous pouvez rencontrer avec les couches supérieures d'un système d'exploitation : lorsque vous demandez la copie de 100 fichiers de votre disque vers votre clé USB, une erreur (d'accès, de place,...) n'annule pas toute la copie si elle s'arrête au 37^e fichier, mais laissera les 36 fichiers copiés sans erreur, sur le support cible.

5.3 Assembler : les jointures

Jusqu'à présent nous avons vu comment faire un **select** sur une seule table. Nous allons voir dans cette section que le langage SQL propose plusieurs formes de requête permettant l'extraction de données, en *travaillant sur plusieurs relations*. Notons que la nécessité de relier deux tables avait émergé lors de la présentation des ►clés étrangères.

§ 4.3.1 ◀
p. 106

 Personne ne niera le fait que le gestionnaire de la base doit être capable de présenter à l'utilisateur deux relations reliées par une clé comme un seul bloc d'information. Par exemple, l'utilisateur de la

base traitant de la discographie de Franck Zappa n'a pas à savoir que les musiciens sont stockés dans une table et les pays dans une autre : il souhaite avoir sous les yeux les musiciens et leur origine sur une même « ligne ».

5.3.1 Produit cartésien

Nous allons découvrir progressivement que pour relier les tables entre elles, il faut s'appuyer sur un opérateur nommé *jointure* (symbolisé \bowtie ci-dessous), reposant sur deux idées :

1. il attend deux opérands : une table située à sa *gauche* (R_g) et une table située à sa *droite* (R_d) :

`select ... from $\boxed{R_g}$ \bowtie $\boxed{R_d}$ on ...`

2. il s'appuie sur la notion de *produit cartésien*.

Le produit cartésien de deux ensembles A et B est un nouvel ensemble P contenant toutes les combinaisons possibles d'éléments constitués d'un élément de A et d'un élément de B . Les mathématiciens notent souvent le produit :

$$P = A \times B$$

Souvenons-nous que les relations peuvent être perçues comme des ensembles et à ce titre SQL offre la possibilité d'exprimer un produit cartésien. Imaginons les deux relations suivantes :

A	B
N	M
α	x
β	y
γ	

Alors la requête suivante :

`select * from A cross join B`

renverra précisément le produit cartésien, appelé parfois *produit en croix* :

N	M
α	x
α	y
β	x
β	y
γ	x
γ	y

Quelques remarques :

- l’opérateur est ici **cross join**, la table de gauche est *A* et celle de droite *B* ;
- dans le cas du produit cartésien, l’ordre des opérandes n’importe pas, on aurait pu écrire, avec le même résultat :

```
select * from B cross join A
```

- le même résultat peut être obtenu avec la requête :

```
select * from A , B
```

5.3.2 Jointure interne

Fort de la connaissance du produit cartésien, nous abordons ici le premier opérateur de jointure : la *jointure interne* ou *inner join*. La motivation première de la jointure est de pouvoir « relier » une table avec une autre. En règle générale — *mais ça n’est pas une nécessité* — la jointure se fait via un attribut de la première table sur lequel il existe une clé étrangère pointant sur un attribut de la deuxième. Supposons l’existence des relations :

Personne				Pays	
id_personne	nom	...	origine	code	nom
1	Zappa	...	US	US	United States
2	Van V.	...	US	GB	United Kingdom
3	Ayler	...		FR	France

Dans ces deux tables on suppose que le champ `personne.origine` est doté d’une clé étrangère pointant sur `pays.code`. Si l’on veut une relation contenant à la fois les attributs de la table des personnes, et ceux de la table des pays (en particulier le libellé du pays), on aura recours à une jointure :

```
select * from personne
        inner join pays on origine=code
```

donnera :

id_personne	prenom	nom	origine	code	nom
1	Zappa	Frank	US	US	United States
2	Van Vliet	Don	US	US	United States

- On notera que :
- l’opérateur **inner join** attend deux opérandes de type relation ;
 - la jointure se fait ici « selon » (**on**) l’égalité des champs `code` et `origine` ;
 - en d’autres termes, le SGBD va envisager toutes les combinaisons possibles (en faisant le produit cartésien), et il ne conservera que les tuples satisfaisant la clause **on**.

Ainsi on peut imaginer que parmi toutes les solutions suivantes :

5

id_personne	prenom	nom	origine	code	nom
1	Zappa	Frank	US	US	United States
1	Zappa	Frank	US	GB	United Kingdom
1	Zappa	Frank	US	FR	France
2	Van V.	Don	US	US	United States
2	Van V.	Don	US	GB	United Kingdom
2	Van V.	Don	US	FR	France
3	Ayler	Albert	•	US	United States
3	Ayler	Albert	•	GB	United Kingdom
3	Ayler	Albert	•	FR	France

le système ne va conserver que les tuples qui répondent « vrai » à la question :

```
personne.origine=pays.code
```

Conformément à ce qui est expliqué à la section 4.4 page 109, tous les tuples pour lesquels le champ `personne.origine` est vide (**NULL**) seront occultés de la jointure.

5.3.3 Lever les ambiguïtés sur les noms des attributs

Il est important de noter que si les attributs avaient été définies avec le même nom (par exemple `code_pays` pour les champs `origine` et `code`), il aurait fallu lever l'ambiguïté en écrivant :

```
select * from personne
  inner join pays
    on personne.code_pays=pays.code_pays
```

au risque de recevoir une remontrance du SGBD. En effet en écrivant :

```
select * from personne
  inner join on code_pays=code_pays
```

l'interpréteur SQL ne saurait décider de quel « `code_pays` » il s'agit. De même s'il fallait dans le résultat de la jointure, intégrer le nom de la personne, le Sgbd ne saurait de lui-même comprendre la requête :

```
select nom from personne inner join pays on ...
```

car le champ `nom` apparaît dans les deux relations opérandes de la jointure. Il faudra donc écrire :

```
select personne.nom from personne
  inner join pays on personne.origine=pays.code
```



Étant donné qu'il peut être fastidieux de préfixer les noms d'attributs par le nom de la relation, SQL offre la possibilité de raccourcir les noms des tables grâce à la syntaxe suivante :

```
select pe.nom, ps.nom from personne pe
  inner join pays ps on pe.origine=ps.code
```

Dans la requête ci-dessus, l'alias « `pe` » peut être utilisé en lieu et place du nom de la relation `personne`.

5.3.4 Jointures externes

Reprenons la situation précédente :

Personne				Pays	
id_personne	prenom	nom	origine	code	nom
1	Zappa	Frank	US	US	United States
2	Van Vliet	Don	US	GB	United Kingdom
3	Ayler	Albert	•	FR	France

Nous avons vu que la jointure interne n'incluait pas dans la relation résultat les tuples n'ayant pas de correspondance pour la clause `on`. En particulier le tuple « Albert Ayler », pour lequel l'origine est inconnue (valeur NULL), n'apparaît pas dans la jointure interne avec la table des pays.

On appelle alors jointure « externe gauche » la requête :

```
select ...
  from <table_gauche>
 left outer join <table_droite> on ...
```

renvoyant :

- la jointure interne entre *<table_gauche>* et *<table_droite>* ;
- à laquelle on ajoute les tuples de la table de gauche n'ayant pas de correspondance dans la jointure via la clause `on`.

Ainsi avec notre exemple, on pourra écrire :

```
select
  id_personne, pe.nom, ps.nom as origine
from  personne pe
left outer join pays ps
  on pe.origine=ps.code
```

id_personne	nom	origine
1	Zappa	United States
2	Van Vliet	United States
3	Ayler	•



La jointure externe gauche (**left outer join**) renvoie donc ici, « tous les musiciens et leur origine, si celle-ci est connue ». Alors que la jointure interne (**inner join**) renvoyait « les musiciens d'origine connue ». On pourra également noter que la plupart des Sgbd ne feront pas la fine bouche si vous omettez le mot clé **outer** car **left** implique nécessairement **outer**.

Oui, vous l'aviez deviné, s'il existe une jointure gauche, il doit bien y en avoir une droite. La jointure droite s'écrit :

```
select ...
  from <table_gauche>
 right outer join <table_droite> on ...
```

renvoyant :

- la jointure interne entre $\langle table_{gauche} \rangle$ et $\langle table_{droite} \rangle$
- à laquelle on ajoute les *tuples de la table de droite n'ayant pas de correspondance dans la jointure via la clause on*.

Ainsi :

```
select id_personne, pe.nom, ps.nom as origine
  from personne pe
 right outer join pays ps on pe.origine=ps.code
```

renverra :

id_personne	nom	origine
1	Zappa	United States
2	Van Vliet	United States
•	•	France



Comme précédemment le mot clé **outer** pourrait être omis car une jointure droite est nécessairement externe (il n'existe pas de jointure interne droite ou gauche). Et de même que précédemment, la jointure droite de cet exemple renvoie « tous les pays et les musiciens originaires de ces pays, y compris les pays dont personne n'est originaire ».

Non, vous n'y pensez pas, si ? Mais oui, après la jointure gauche, après la jointure droite, merci d'accueillir : *la jointure complète* ! Vous avez deviné, si la jointure gauche (respectivement droite) ajoute à la jointure interne les tuples de la table de gauche (respectivement de droite) sans correspondance, la jointure externe complète (**full outer join**) ajoute les deux, par exemple :

```
select id_personne, pe.nom, ps.nom as origine
  from personne pe
 full outer join pays ps on pe.origine=ps.code
```

renverra la relation :

id_personne	nom	origine
1	Zappa	United States
2	Van Vliet	United States
•	•	France
3	Ayler	•

5.3.5 Autres jointures

Plus de deux tables

Les opérateurs de jointures agissent sur deux tables. Nous avons même vu qu’elles avaient un rôle particulier en fonction de leur position (gauche ou droite) par rapport à l’opérateur. Pour réaliser une jointure faisant intervenir plus de deux tables on utilisera le fait qu’une jointure renvoie une relation, celle-ci peut donc être jointe avec une troisième. La forme de la requête est donc la suivante :

```
select ... from
  <table1>
  ... join <table2> on ...
  ... join <table3> on ...
```

Si on reprend l’exemple donné au paragraphe 4.7 page 124 sur les associations de plusieurs à plusieurs :

Musicien		
idMus	Nom	Prenom
4	Vander	Ch.
18	Zappa	F.k

saitjouer	
idMus	idIns
4	12
4	11
18	11
18	10

Instrument	
idIns	libelle
10	Guitare
11	Chant
12	Batterie

Il est possible avec la requête suivante :

```
select m.idMus, m.nom, i.idIns, i.libelle
  from musicien m
  inner join saitjouer sj on m.idMus = sj.idMus
  inner join instrumemt i on sj.idIns = i.idIns
```

de produire la relation suivante :

idMus	nom	idIns	libelle
4	Vander	12	Batterie
4	Vander	11	Chant
18	Zappa	11	Chant
18	Zappa	10	Guitare

Jointure d'une table avec elle-même

Mais, pourquoi diable faudrait-il joindre une table avec elle-même ? Par exemple pour envisager tout ou partie des couples formés de deux éléments de la table en question. Par exemple, vous avez un ensemble de participants :

participant	
id	nom
1	Tartempion
2	Machinchouette
3	Biduletruc

et vous voulez organiser un tournoi de ping-pong. Qu'à cela ne tienne vous pouvez élaborer l'ensemble des rencontres comme ceci :

```
select * from participant p1
      cross join participant p2
```

qui renverra le produit cartésien de la table participant avec elle-même, c'est-à-dire :

id	nom	id	nom
1	Tartempion	1	Tartempion
2	Machinchouette	1	Tartempion
3	Biduletruc	1	Tartempion
1	Tartempion	2	Machinchouette
2	Machinchouette	2	Machinchouette
3	Biduletruc	2	Machinchouette
1	Tartempion	3	Biduletruc
2	Machinchouette	3	Biduletruc
3	Biduletruc	3	Biduletruc

Les plus attentifs d'entre vous auront remarqué que M. Tartempion va tenter une rencontre pongiste mémorable avec lui-même. Ceci pourrait être évité en transformant le produit cartésien ci-dessus en auto-jointure interne :

```
select * from participant p1
      inner join participant p2 on p1.id!=p2.id
```

Si l'on voulait éviter les matches retour, on pourrait également se fendre d'un :

```
select * from participant p1
      inner join participant p2 on p1.id>p2.id
```

qui renverrait la relation :

id	nom	id	nom
2	Machinchouette	1	Tartempion
3	Biduletruc	1	Tartempion
3	Biduletruc	2	Machinchouette

La condition de jointure $p1.id < p2.id$ réglerait également la suppression des matches retour.

5.4 Agréger

5

Le langage SQL propose des formes particulières du select dans lesquelles :

1. on va **fusionner** des tuples ayant des valeurs communes pour un ou plusieurs attributs dans le but de fabriquer des sous-ensembles appelés *agrégats* ;
2. puis **appliquer une fonction** à chacun de ces sous-ensembles, par exemple : compter, calculer la moyenne, le max, etc. La fonction en question se nomme *fonction d'agrégation*.

On pourra par exemple répondre à des questions comme :

- quel est le *nombre* de musiciens pour chaque album de Frank ZAPPA ;
- quelle est la durée moyenne des morceaux pour chaque année entre 1973 et 1978 ;
- de manière générale, faire toutes sortes de statistiques ;
- ...



Nous pouvons noter, avant d'entrer dans le vif du sujet, que les agrégations pourront être combinées avec les jointures que nous venons d'aborder. D'autre part, ces agrégations font partie des requêtes dont la mise au point n'est pas forcément immédiate, c'est pourquoi nous prenons le temps dans ce qui suit d'en détailler le processus de construction.

5.4.1 Principe général

Nous nous appuierons sur la relation suivante :

R			
α (entier)	β (entier)	γ (entier)	δ (chaîne)
1	50	3	A
2	•	4	C
1	•	3	E
3	30	6	G
2	20	4	I
2	10	4	K

Quel(s) attribut(s) on agrège

C'est la clause **group by** qui permet de le spécifier. On écrira par exemple :

```
select ... from R group by  $\alpha,\gamma$ 
```

La clause **group by** indique que l'on regroupe *les tuples* en créant des agrégats pour tous ceux ayant des valeurs identiques pour les attributs α et γ .

R			
α	β	γ	δ
1	50	3	A
2	•	4	C
1	•	3	E
3	30	6	G
2	20	4	I
2	10	4	K

Dans l'exemple ci-dessus, les tuples en gris contiennent tous les valeurs (2,4) pour ces attributs, les tuples en gris foncé (3,6) et les tuples en gris clair (1,3).

Ce qu'on fait de l'ensemble des tuples agrégés

Une fois les agrégats créés, il faut indiquer ce que l'on fait de chacun des attributs non concernés par le regroupement — dans notre exemple, les attributs β et δ . C'est ici que l'on fait appel à une *fonction d'agrégation* dont chaque SGBD dispose. On pourra par exemple dire : pour chaque agrégat de tuples :

- je veux calculer la moyenne de l'attribut β
- je veux connaître le nombre de valeurs de l'attribut β
- je veux obtenir le max de l'attribut δ

Ceci peut être obtenu en implorant votre SGBD comme suit :

```
select  $\alpha$  , avg( $\beta$ ) , count( $\beta$ ) ,  $\gamma$  , max( $\delta$ )
  from R
 group by  $\alpha$  ,  $\gamma$ 
```

α	moyenne	nombre	γ	max
1	50	1	3	E
3	30	1	6	G
2	15	2	4	K

5

Les nouvelles fonctions que nous rencontrons ici sont les *fonctions d'agrégation* :

- **avg** fait la *moyenne* des valeurs de l'attribut indiqué, pour chaque agrégat, en traitant correctement l'absence de valeur (le vide n'entre pas en compte dans le calcul, voir aussi § 5.4.3 page 192);
- **count** *compte* le nombre de valeurs de l'attribut pour chaque agrégat;
- **max** calcule la valeur max des valeurs de l'attribut pour chacun des agrégats.



Une règle à mémoriser pour bien comprendre la construction d'une requête avec agrégation est la suivante, dans :

```
select <liste_attr> from ... group by <liste_groupe>
```

Un attribut de la clause *<liste_attr>* doit :

- soit apparaître dans la clause *<liste_groupe>*;
- soit apparaître dans une fonction d'agrégation.

Dans le cas contraire le SGBD vous enverra gentiment sur les roses.

Par exemple, dans :

```
select nom, prenom, max(taille)
  from personne group by nom
```

l'attribut `prenom` n'est pas dans la clause **group by**, et se trouve dans la liste des attributs sans faire appel à une fonction d'agrégation. L'interpréteur SQL vous reprocherait ici de lui demander d'agréger par nom et ne pas préciser ce que vous voulez faire des prénoms. Ici, il faudrait donc, soit supprimer l'attribut `prenom` de la liste des attributs, soit l'ajouter dans la clause **group by**, soit lui appliquer une fonction d'agrégation.

5.4.2 Un petit exemple

Supposons que l'on dispose de la relation suivante dans laquelle on a posé une clé primaire sur l'attribut `id` :

individu		
id	nom	prenom
10	X	Luc
11	Y	Jean
12	Z	Jean
13	Z	Luc
14	Z	Jean
15	T	•

et que l'on veuille compter les occurrences de chacun des prénoms de ces individus. Pour cela on écrira :

```
select prenom, count(id) as nb from individu
  group by prenom
```

Ceci retournera la relation :

prenom	nb
Luc	2
Jean	3
•	1

Ici, la fonction **count** compte le nombre de clés `id` pour chaque agrégat créé en regroupant les tuples ayant le même prénom.

Une erreur fréquemment commise est la suivante :

```
select
  prenom, count(prenom) as nb
from individu group by prenom
```

qui renverrait la relation :

prenom	nb
Luc	2
Jean	3
•	0

Puisque dans ce cas, on compte le nombre de prénoms pour chaque agrégat. Le bon choix pour l'attribut à compter est donc bien la clé primaire puisque celle-ci garantit l'existence d'une valeur pour chaque attribut (cf. § 4.2.3 page 100).

Au passage, voici une autre application de la fonction **coalesce** permettant ici de présenter les résultats :

```
select
  coalesce(prenom, '[?inconnu?]' ) as prenom,
  count(id) as nb
from individu group by prenom
```

qui renverra :

prenom	nb
Luc	2
Jean	3
[?inconnu?]	1



La fonction **coalesce** renvoie le premier de ses arguments non vides. En d'autres termes si on l'appelle avec :

```
... coalesce(<arg1>, <arg2>, ... <argn>)
```

elle retournera le premier des $\langle arg_i \rangle$ n'étant pas **NULL**. Dans notre exemple elle renverra donc la valeur de l'attribut **prenom** s'il n'est pas **NULL** et la chaîne '[?inconnu?]' dans les autres cas puisque cette valeur ([?inconnu?]) n'est pas **NULL**.

Autre exemple : les relations à partir desquelles on souhaiterait connaître pour chaque pays, le nombre de personnes :

Personne			Pays	
id	nom	origine	code	libelle
1	Zappa	US	US	United States
2	Van Vliet	US	GB	United Kingdom
3	Ayler	•	FR	France

On supposera que `pays.code` est la clé primaire. Il faudra d’abord choisir la bonne jointure (**left** ou **right**), de manière à faire apparaître *tous les pays* et le nombre les personnes correspondantes, puis procéder à l’agrégation. Voyons d’abord la jointure :

```
select *
  from pays
 left join personne per
    on per.origine=pays.code
```

qui donne :

code	libelle	id	nom	origine
US	United States	1	Zappa	US
US	United States	2	Van Vliet	US
GB	United Kingdom	•	•	•
FR	France	•	•	•

On va donc agréger les tuples ayant même valeur pour les attributs `code` et `libelle`, puis compter. L’attribut candidat pour ce comptage est `id` (clé primaire, donc valeur présente nécessairement), ou `nom` si une contrainte de type **not null** a été posée. On écrira donc :

```
select
  code, libelle, count(id) as nb
  from pays
 left join personne per
    on per.origine=pays.code
 group by code, libelle
```

qui renverra :

code	libelle	nb
US	United States	2
GB	United Kingdom	0
FR	France	0

5.4.3 Fonctions d'agrégation et absence de valeur

Nous avons déjà remarqué que les fonctions d'agrégation traitaient l'absence de valeur (**NULL**) de la « bonne manière » au paragraphe 5.4.1 page 187. Nous avons vu, qu'appliquée aux valeurs :

A
10
•
30

la fonction **avg(A)** renverra la valeur 20, c'est-à-dire que le calcul de la moyenne ne sera fait qu'en tenant compte des valeurs présentes. Il en est de même pour :

- **sum(A)** : qui donnera 40
- **count(A)** : qui donnera 2
- **min(A)** : qui donnera 10
- **max(A)** : qui donnera 30

5

5.4.4 Filtrer avant ou après l'agrégation

Intuitivement on pourrait vouloir écrire

```
select prenom, count(prenom) as nb
  from individu
 group by prenom
 where count(prenom)>4
```

pour exprimer le fait qu'on ne veut conserver que les prénoms pour lesquels le nombre d'occurrences est supérieur à 4. Cependant, l'interpréteur SQL renverra alors une erreur car pour lui, la clause **where** n'est pas à sa place. Il faut en effet comprendre que cette dernière a pour but de filtrer les tuples *avant l'agrégation*, par exemple :

```
select prenom, count(prenom) as nb
  from individu
 where prenom not like '%e%'
 group by prenom
```

qui signifie : merci de compter le nombre de chacun des prénoms ne contenant pas la lettre 'e'. SQL dispose d'une clause spéciale pour le filtrage *après agrégation* : la clause **having**.

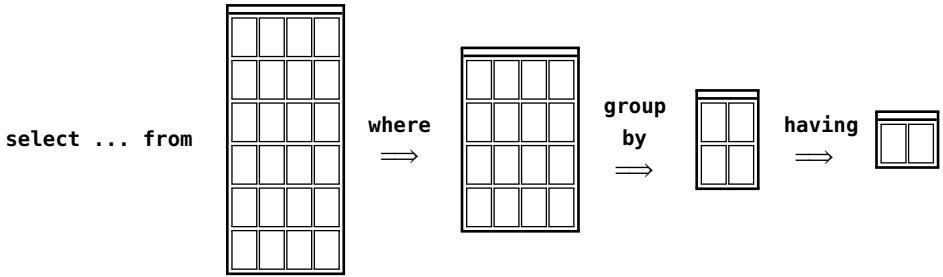


FIGURE 5.2 – Filtrer avant ou après l'agrégation : la figure illustre les différentes étapes autour de l'agrégation : le filtrage avant (*where*), l'agrégation elle-même (*group by*) et enfin le filtrage après l'agrégation (*having*).

Celle-ci s'écrit là où vous souhaitez l'écrire, c'est-à-dire :

```
select prenom, count(prenom) as nb
  from individu
 group by prenom
 having count(prenom)>4
```

qui effectue le filtrage souhaité (ne conserver que les prénoms apparaissant plus de 4 fois). On peut bien sûr combiner les deux clauses (*where* et *having*) :

```
select prenom, count(prenom) as nb
  from individu
 where nom like '%a%'
 group by prenom
 having count(prenom)>4
```

On fait le même type de statistique (nombre de prénoms apparaissant plus de 4 fois), mais ici uniquement sur les personnes dont le nom contient un 'a'. La figure 5.2 illustre le positionnement des différentes étapes autour des opérations d'agrégation.

5.4.5 Agréger des chaînes

À partir de la version 9, PostgreSQL propose à ses valeureux utilisateurs une fonction pour agréger les chaînes de caractères :

```
select origine, string_agg(nom, '/') as noms
  from musicien group by origine
```

Notez l'utilisation d'un caractère séparateur. Appliquée à la relation de gauche ci-dessous, cette requête renverra la relation de droite :

Personne		
nom	prenom	origine
Zappa	Franck	US
Hendrix	Jimi	US
Wyatt	Robert	GB
Vander	Christian	FR

origine	noms
US	Zappa/Hendrix
GB	Wyatt
FR	Vander

Avec une clause **order by** nous pouvons même nous offrir le luxe de classer les noms :

```
select
  origine ,
  string_agg(nom,'=>' order by nom) as noms
from musicien group by origine
```

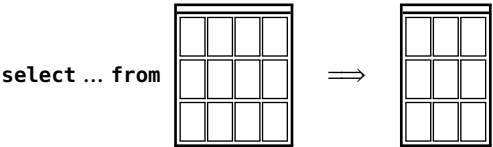
qui classera donc *dans l'attribut* noms, les différents éléments séparés par la chaîne => :

origine	noms
US	Hendrix=>Zappa
GB	Wyatt
FR	Vander

5

5.5 Combiner des requêtes

Nous vous proposons ici quelques fonctionnalités autour de la requête **select** exploitant le fait que cet opérateur renvoie une relation et donc un ensemble.



5.5.1 Mise en bouche : l'opérateur **in**

Passé sous silence jusqu'ici, l'opérateur **in** du langage SQL permet de poser la question de savoir si une valeur appartient (en anglais *in* veut dire « dans ») à un ensemble. Par exemple lorsqu'on écrit :

```
select  nom from musicien
        where prenom in ( 'Jean', 'Jacques', 'Bob' )
```

on demande quels sont les noms des musiciens dont le prénom correspond à l'une des valeurs indiquées dans la liste construite entre parenthèses, à droite de l'opérateur **in**.

Il est tout à fait envisageable de construire cette liste dynamiquement à l'aide d'un **select** (qui devra toutefois ne renvoyer que des tuples constitués d'un seul attribut) :

```
select ... from <table1>
        where <attribut>
        in ( select ... from <table2> where ... )
```

Si vous souhaitiez connaître le nom des personnes dont le nom est celui d'une commune de la région Rhône-Alpes, vous pourriez écrire quelque chose du genre :

```
select nom, prenom from personne
        where nom
        in ( select lib from commune where idReg=4 )
```

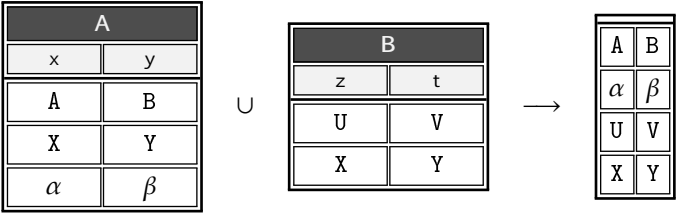
En supposant que la table des communes contienne un attribut **lib** pour chacun des noms et que cette table soit dotée d'un attribut **idReg** pour lequel la région Rhône-Alpes ait la valeur 4.

5.5.2 Opérations ensemblistes

« Vous avez dit ensemble ? » donc vous accepterez qu'on puisse utiliser les opérateurs classiques tels que *intersection*, *union* et *différence* sur nos chères relations. Ainsi, la requête suivante :

```
select x, y from A union select z, t from B
```

effectue l'*union* des deux ensembles, c'est-à-dire renvoie un ensemble contenant les tuples de chacun des deux ensembles opérands. On notera le traitement particulier réservé au tuple (X, Y) :



Comme indiqué précédemment, il est tout à fait possible de baptiser les attributs de la relation résultat (par exemple lors de la création d'une vue) en écrivant : **as**

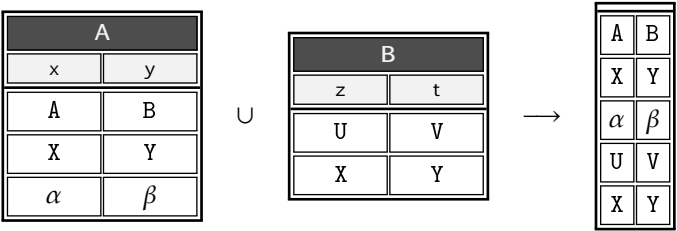
► § 6.1
p. 208

```
select x, y from A
union
select z as x, t as y from B
```

On notera que :

```
select x, y from A
union all
select z, t from B
```

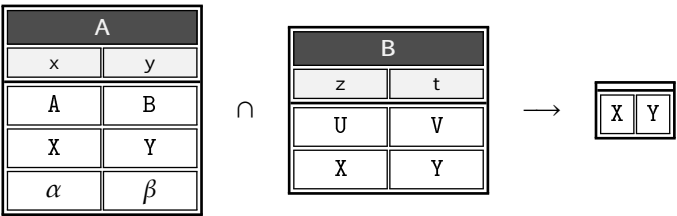
ne « dédoublonne » pas les tuples dans la relation résultat, on retrouvera donc ici :



L'intersection de 2 ensembles peut être exprimée comme suit :

```
select x, y from A intersect select z, t from B
```

et peut être illustrée par les relations suivantes :



Les plus attentifs d'entre vous rétorqueront qu'une intersection peut tout à fait être obtenue grâce à une jointure finement travaillée :

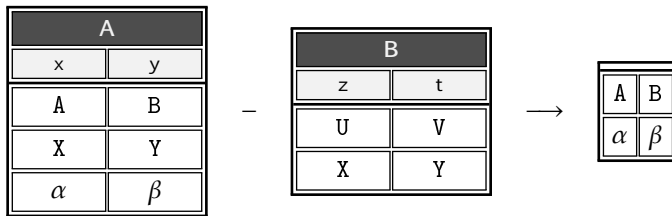
```
select x, y from A inner join B on x=z and y=t
```

Et nous ne pourrions pas leur donner tort.

Pour terminer sur les opérations ensemblistes, voici la *différence* de deux ensembles :

```
select x, y from A except select z, t from B
```

et l'illustration qui va avec :



5.5.3 Select de Select de Select de ...

Vous vous souvenez que l'opérateur **select** travaille sur une relation et renvoie une relation. Ceci implique qu'il est possible d'écrire une requête du type :

```
select ... from (select ... from ...)
```

Le SGBD PostgreSQL impose qu'on baptise la sous-requête entre parenthèses, en stipulant un nom après celles-ci. Par exemple :

```
select ... from (select ... from ...) stat
```

Cette construction permet donc de nommer *stat* la sous-requête entre parenthèses. Ce nom pourra être utilisé pour faire référence aux attributs renvoyés par celle-ci, par exemple :

```
select max(stat.nb) from (
    select ...nb...
) stat
```

Mais voyons sans plus attendre quelques exemples de requêtes imbriquées.

Nombre max de ...

► § 5.4
p. 186

Une question récurrente liée aux agrégations ◀ est une question du type :

« *quel est le maximum de personnes originaires du même pays ?* »

Nous allons voir que la réponse à cette question n'est pas si triviale à formuler en SQL. Un début de réponse est donné par une requête incluant jointure et agrégation :

```
select p.libelle, count(m.idmus) as nb
  from musicien m
    inner join pays p on m.origine=p.code
 group by p.libelle
```

produisant les statistiques des origines :

libelle	nb
France	12
United Kingdom	10
Spain	4
United States	30
...	...

Si on nomme temporairement cette relation stat, alors la requête :

```
select max(nb) from stat
```

renvoie le résultat escompté, à savoir un tuple doté d'un attribut contenant la valeur max de toutes les valeurs que peut prendre nb. Étant donné que stat n'existe pas en tant que relation dans la base, mais n'est que le résultat d'un select, il suffit d'imbriquer cette requête dans la requête précédente :

```
select max(nb) from
( select p.libelle, count(m.idmus) as nb
  from musicien m inner join pays p
    on m.origine=p.code
 group by p.libelle ) stat
```

Le nombre de pays dont 3 personnes ...

Un autre exemple : « *Quel est le nombre de pays dont 3 personnes au moins sont originaires ?* ». Dans le même ordre d'idée, on pourra écrire :

```
select count(*) from
( select p.libelle, count(m.idmus) as nb
  from musicien m inner join pays p
    on m.origine=p.code
 group by p.libelle
 having count(m.idmus)>=3 ) stat
```



Contrairement à ce qu'on pourrait penser intuitivement, le raccourci `nb` donnant un nom au `count`, ne peut pas être utilisé dans la clause `having` :

```
having nb>=3 -- pas possible ici
```

Le pays avec le plus de personnes

S'il fallait maintenant afficher *le libellé du pays pour lequel il y a le plus grand nombre de personnes*, on pourrait utiliser le système de restriction de tuples (cf. § 5.1.9 page 170). En écrivant par exemple :

```
select libelle from
( select p.libelle, count(m.idmus) as nb
  from musicien m inner join pays p
    on m.origine=p.code
 group by p.libelle ) stat
order by nb desc limit 1
```

Le `order by` de cette requête permet d'ordonner la relation `stat` générée par la sous-requête, par ordre décroissant de la valeur `nb` et de ne garder qu'une seule ligne de ce tri (clause `limit`), par conséquent de produire la plus grande valeur.

5.5.4 S'il existe...

Il existe dans SQL un test particulier nous permettant de répondre à une question du type : « *quels sont les musiciens (nom et prénom) ne sachant jouer d'aucun instrument ?* ». On rappelle le modèle relationnel sur lequel on s'appuie :

Musicien			saitjouer		Instrument	
idMus	Nom	Prenom	idMus	idIns	idIns	libelle

Une façon de formuler la question en SQL est d'écrire ceci :

```
select nom, prenom from musicien m
  where not exists (
    select 1 from saitjouer j
      where j.idMus=m.idMus )
```

La construction **exists (...)** renvoie la valeur booléenne vrai si la requête passée entre parenthèse renvoie au moins un tuple. Dans la mesure où ce qui importe est de savoir si un tuple est renvoyé ou non, les attributs sélectionnés n'ont pas d'influence. C'est la raison pour laquelle on a écrit ici **select 1 from ...**.



Notons qu'il est tout à fait possible de construire une jointure externe répandant à la même question :

```
select m.nom, m.prenom
  from musicien m
 left join saitjouer j on j.idmus=m.idmus
 where j.idmus is NULL
```

5

On pourra également répondre à la même question en prenant la négation de l'énoncé : « *quels sont les musiciens sachant jouer d'un instrument ?* ». On écrira alors simplement :

```
select nom, prenom from musicien m
  where exists (
    select 1 from saitjouer j
      where j.idMus=m.idMus )
```



Ici on pourra traduire cette dernière requête par une jointure :

```
select distinct m.nom, m.prenom
  from musicien m
 left join saitjouer j on j.idmus=m.idmus
 where j.idmus is not NULL
```

On notera cependant dans ce cas qu'il est nécessaire de « dédoubler » les résultats avec le mot clé **distinct**. En effet, un même musicien peut savoir jouer plus d'un instrument, il apparaîtra alors plusieurs fois dans le résultat de la jointure.

5.6 Utiliser des données externes

Dans ce paragraphe, nous entendons par *manipulation de données externes*, les deux situations bien distinctes :

1. alimenter une table depuis un fichier texte ou créer un fichier texte à partir du contenu d'une table ;
2. ajouter ou extraire des données correspondant à des attributs de type « tas d'octets » présentés au chapitre précédent.

5.6.1 Remplir les relations à partir de fichiers

Vous aurez parfois besoin de récupérer des données depuis un fichier et de les insérer dans une des relations de votre base. Le format adapté pour ce type de situation est le format appelé « csv » pour *comma separated values* : valeurs séparées par des virgules. Ce type de fichier est un fichier texte, dont l'organisation est la suivante :

```

- <val. attr.1> , <val. attr.2> , <val. attr.3> , ... \n
- <val. attr.1> , <val. attr.2> , <val. attr.3> , ... \n
- <val. attr.1> , <val. attr.2> , <val. attr.3> , ... \n
- ...

```

où :

- <val. attr._i> est la valeur du i^e attribut ;
- la virgule fait office de caractère séparateur des attributs ;
- \n représente un saut de ligne ;



Dans un fichier csv, le séparateur peut être n'importe quel caractère à condition qu'il n'apparaisse jamais dans la valeur des attributs. Les pratiques courantes sont les espaces, les tabulations, la virgule, le point-virgule, etc.

Voyons maintenant un exemple : l'ISO a normalisé pour chaque pays du monde deux codes, un à deux caractères et l'autre à trois caractères. Le listing ci-dessous montre un extrait de l'ensemble des informations fournis par l'ISO-3166 :

```

alpha_2_code,alpha_3_code,numeric_code,short_name_en,short_name[...]
"BR","BRA","076","Brazil","BRAZIL","the Federative Republic of Brazil"
"CH","CHE","756","Switzerland","SWITZERLAND","the Swiss Confederation"
"NZ","NZL","554","New Zealand","NEW ZEALAND",""

```

Créons une relation pour stocker ces pays :

```
create table pays(code char(2) primary key ,
                libelle text ,
                libelle_long text )
```

Nous préparons ensuite un fichier csv ne contenant que les champs qui nous intéressent :

```
"BR","Brazil","BRAZIL","the Federative Republic of Brazil"
"CH","Switzerland","SWITZERLAND","the Swiss Confederation"
```

Il est alors aisé d'insérer le contenu de ce fichier (que, dans un moment de lyrisme absolu, nous nommerons pays.csv) dans la table pays :

```
\copy pays from 'pays.csv' delimiter ',' quote '"' csv
```

Cette commande charge le fichier en supposant qu'il est dans le répertoire où vous avez lancé la commande `psql`. Elle stipule que le délimiteur est la virgule et que tous les champs sont entourés de guillemets (c'est le sens de quote `'"`). Notez également l'utilisation du mot clé `csv`.

L'opération inverse — exporter les données d'une relation vers un fichier — est bien sûr possible et c'est la même commande qui permet de le réaliser. On pourra par exemple écrire :

```
\copy musicien to 'zikos.csv' delimiter ',' csv
```

qui créera le fichier csv `zikos.csv` avec la virgule comme séparateur. Un tel fichier pourra ensuite être traité par un tableur.



Attention, la commande `copy` et toutes les méta-commandes de `psql` — celles commençant par `\` — ne peuvent être coupées et doivent par conséquent être écrites sur une seule ligne.

5.6.2 Stocker des fichiers

Nous avons vu au paragraphe 4.9 page 141 qu'il était possible via le type `bytea`, de stocker des fichiers comme attribut d'une relation en tant que séquence d'octets. La table présentée est la suivante :

Enregistrement		
idEnreg (entier)	titre (chaîne)	son («tas d'octets»)
237	G-Spot Tornado	FD 48 58 0A B8...
44	Penguin in Bondage	AB 48 C8 34 FF...

créée par cette requête :

create table

```
enregistrement(idEnreg int primary key,
               titre text,
               son bytea)
```



Contrairement aux types natifs du Sgbd, le type `bytea` ne permet pas les requêtes classiques du type :

```
insert into enregistrement(titre,son)
values('Oh_no', ????)
```

Pas plus qu'une requête du type :

```
select * from enregistrement
where son ????
```

car le Sgbd ne dispose pas d'opérateur pour chacun des formats de fichier que pourrait fournir l'utilisateur. Nous présentons donc ici une technique faisant appel à un langage d'application pour insérer et extraire un fichier stocké dans le champ `son` de la table `enregistrement`. Le langage choisi est Python.

Insérer un document



Les deux paragraphes qui suivent (« Insérer un document » et « Extraire un document ») exigent des connaissances en programmation et plus précisément en Python. Vous pouvez les ignorer si vous ne vous sentez pas d'attaque...

En supposant qu'on dispose :

- d'un fichier `penguin.ogg`
- du titre correspondant à cet enregistrement : « Penguin in Bondage »

alors, pour insérer le contenu d'un fichier fourni par l'utilisateur, on procédera comme suit :

1. ouvrir le fichier et le charger en mémoire du programme

2. se connecter à la base de données
3. convertir le contenu en mémoire pour l'insérer via un **insert** dans la relation enregistrement.

Les instructions Python pour ouvrir et lire le contenu d'un fichier sont les suivantes :

```
1 # ouverture en lecture , octet par octet
2 fichier=open( 'penguin.ogg' , 'rb' )
3 # lecture du contenu
4 son=fichier.read()
```

difficile de faire plus simple... Après ces deux instructions, la variable `son` contient la séquence d'octets composant le fichier dont le nom est `penguin.ogg`. La suite consiste à ouvrir une connexion avec la base de données :

```
5 # module Python pour connexion Postgresql
6 import psycopg2
7 # connexion au sgbd
8 conn = psycopg2.connect( """dbname='yahozna '
9                          user='lozano ' """ )
```

5

Une fois la connexion établie, on va exécuter la requête d'insertion :

```
10 titre="Penguin in Bondage"
11 cur=conn.cursor()
12 cur.execute( """ insert into
13                enregistrement( titre , son)
14                values(%s,%s) """ ,
15                ( titre , psycopg2.Binary( son ) ) )
16 conn.commit()
17 cur.close()
```

Nous porterons ici notre attention sur l'appel à la méthode `Binary` du module `psycopg2`, module capable d'interfacer Python et PostgreSQL. Cette routine a pour but de transcrire la séquence d'octets stockée dans la variable `son` en quelque chose d'acceptable par PostgreSQL. Si vous aviez à le faire à la main — ce qui est possible si vous avez de la patience — vous auriez à tapoter :

```
insert into enregistrement(titre,son)
values('Test','\x_4F_68_68_53_00_02...')
```

En d'autres termes vous auriez à saisir le « tas d'octets » du fichier `penguin.ogg`, octet par octet, en prenant soin d'entrer chacun d'eux

avec deux chiffres hexadécimaux — les espaces entre les couples d'octets étant facultatifs. Et pour parfaire cet exercice de saisie passionnant vous penseriez à saisir la séquence de contrôle \x annonçant dans les règles de la politesse à PostgreSQL que ce qui suit est de l'hexadécimal.

Extraire un document

Pour extraire un document déjà stocké dans l'attribut son de la relation enregistrement, on se munira :

- de l'identifiant de l'enregistrement à extraire, par ex. 237;
- d'un nom de fichier dans lequel on stockera cet enregistrement, gspot.ogg

La démarche générale est ensuite la suivante :

1. se connecter à la base;
2. exécuter une requête select pour récupérer la valeur de l'attribut son du tuple identifié par la clé 237;
3. stocker cette valeur dans une variable;
4. ouvrir un fichier en écriture et y stocker le contenu de cette variable.

Voici le squelette de base en Python pour réaliser tout ceci :

```

1 import psycopg2
2 # connexion à la base
3 conn = psycopg2.connect( """dbname='yahozna'
4                             user='lozano'""" )
5 # requête select pour récupérer la musique
6 cur = conn.cursor()
7 cur.execute( """select son
8                  from enregistrement
9                  where ide=237""" )
10 # stockage dans une variable Python
11 son=cur.fetchone()
12 # écriture dans un fichier
13 f=open('gspot.ogg', 'wb')
14 f.write(son[0])
15 # c'est fini, on ferme
16 cur.close()
```

Après exécution de ce code Python, vous aurez en votre possession un joli fichier nommé gspot.ogg contenant la musique stockée dans l'attribut son de la table enregistrement.

Et après, on fait quoi ?

Vous étiez déjà des experts *bâtisseurs de données* (suite à votre lecture attentive du chapitre 4), vous êtes désormais des spécialistes de la *manipulation de données*. Félicitations ! Il vous reste maintenant à découvrir, en plus des données, comment on peut **stocker et exécuter des traitements** à l'aide d'une base de données.

- 6.1 Vues
- 6.2 Avant-propos sur les procédures stockées
- 6.3 Un nouveau langage
- 6.4 Procédure
- 6.5 Trigger
- 6.6 Utiliser un autre langage
- 6.7 Accès concurrents
- 6.8 Interaction avec un programme : PHP

Stocker des traitements

*I can't understand why people are frightened of new ideas.
I'm frightened of the old ones.*

John CAGE.

CE CHAPITRE aborde les outils de SQL permettant de stocker des *traitements* dans une base de données. Pour commencer en douceur, les vues sont d'abord présentées. Sont ensuite exposés les grands principes du volet procédural de SQL qui en font un véritable langage de programmation. Cet « outillage » est indispensable pour aborder ensuite deux types de routines (ou procédures) :

1. celles qui s'appellent automatiquement ou implicitement : les *triggers* ;
2. celles que le concepteur de la base ou d'une application devra explicitement appeler : les *procédures stockées*.

Le chapitre continue par un aspect incontournable de l'étude des bases de données : la *gestion des accès concurrents* ou comment réagit un SGBD lorsque plusieurs processus accèdent simultanément aux données. Enfin, ceux qui seraient intéressés par les détails de l'interfaçage d'une application (écrite en Php pour l'exemple) avec un SGBD trouveront quelques informations utiles en toute fin de chapitre.

6.1 Vues

Le premier type de traitement que nous envisagerons dans ce chapitre se nomme *vue*. Les vues du langage Sql sont des objets particuliers permettant de stocker des requêtes de type **select**. Leur but est double :

1. d'abord *simplifier le modèle perçu par l'utilisateur* : les vues se présentent en effet comme des relations, il est donc possible de faire des **select** sur celles-ci comme s'il s'agissait de table.
2. *simplifier l'écriture des requêtes* : on a souvent besoin, pour extraire des informations de la base, d'écrire des jointures complexes mettant en jeu plusieurs tables. De manière à éviter d'avoir à réécrire de multiples fois ces requêtes, on a recours aux vues.

6.1.1 Création des vues

Pour créer une vue on écrit tout simplement :

```
create view <mavue> as select ...
```

L'objet <mavue> peut alors être utilisé¹ comme s'il s'agissait d'une table. Par exemple :

```
create view v_musicien_instrument
as
select m.idMus, m.nom, i.idIns, i.libelle
  from musicien m
  inner join saitjouer sj on m.idMus=sj.idMus
  inner join instrument i on sj.idIns= i.idIns
```

On pourra alors écrire :

```
select nom, libelle from v_musicien_instrument
  where libelle like '%sax%'
```



Une vue stocke un traitement (la requête sous-jacente) dont l'exécution (grâce à **select**) renvoie une relation. Les données quant à elles sont stockées dans les tables. Aucune donnée n'est stockée dans une vue. En revanche, grâce au mécanisme d'exécution, la relation renvoyée est toujours à jour, même si on modifie les données des tables sous-jacentes. Tout ceci est illustré à la figure 6.1 page suivante.

1. Avec **select**, pour le reste, voir plus loin.

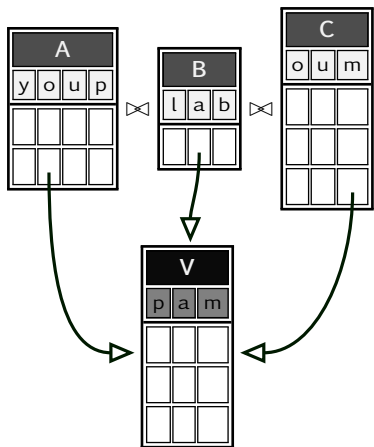


FIGURE 6.1 – Représentation schématique d’une vue : la vue V est créée à partir d’une requête mettant en jeu les relations A, B et C. Cette vue peut alors être manipulée comme une relation (ce qu’elle n’est pas puisqu’elle ne contient aucune donnée) en particulier avec **select**. Cette dernière opération renverra toujours des données fraîches, même si les données des relations A, B ou C sont modifiées.



Il est possible d’obtenir la liste des vues de la base grâce à la commande : `\dv`, ainsi que le détail d’une vue grâce à :

```
\dv+ v_musicien_instrument
```

View "public.v_musicien_instrument"				
Column	Type	Modifiers	Storage	Description
idmus	integer		plain	
nom	text		extended	
idins	integer		plain	
libelle	text		extended	
View definition:				
SELECT m.idmus,				
m.nom,				
i.idins,				
i.libelle				
FROM musicien m				
JOIN saitjouer sj ON m.idmus = sj.idmus				

6.1.2 Règles d'utilisation

Comme dit dans le paragraphe précédent, une fois une vue créée, elle peut être manipulée comme une relation au travers d'une requête **select** (jointures et agrégations comprises). Il se trouve que certains Sgbd comme PostgreSQL pour ne pas le nommer, proposent un mécanisme permettant également de faire des **update**, **delete** et autres **insert** sur des vues. Ce mécanisme est connu sous le nom de **rule** (ou *règles* dans la langue de Jean-Baptiste²). Voyons leur utilisation avec un exemple :

```
create view v_musicien as
select m.id,m.nom,m.prenom,p.code,p.libelle
  from musicien m
 inner join pays p on m.origine=p.code
```

Cette vue renverra une relation :

v_musicien				
id	prenom	nom	code	libelle
1	Zappa	Frank	US	United States
2	Van Vliet	Don	US	United States
...

Si vous jamais vous tentiez un³ :

```
delete from v_musicien where id=1
```

ce brave PostgreSQL vous répondrait vertement :

ERROR: cannot delete from a view

HINT: You need an unconditional ON DELETE DO INSTEAD rule.

Message on ne peut plus clair : « *vous n'avez pas le droit de faire un **delete** sur une vue* ». En revanche une lueur d'espoir apparaît : il semblerait que ça soit quand même possible si on avait une règle...

Ce que nous suggère le SGBD signifie qu'il est possible pour lui de faire un *traitement particulier* (à définir par nous) si l'on réclame un **delete** sur la vue. La création d'une *règle* (*rule*) consiste précisément à expliquer comment doit réagir le SGBD dans ce cas.

2. POQUELIN, faut suivre les gars...

3. Votre serveur souhaite malgré tout que jamais ne vous passe par la tête l'idée de supprimer ce grand iconoclaste devant l'éternel qu'a été Frank ZAPPA...

Voici un exemple de règle, d'abord en français : « *si quelqu'un a l'outrecuidance de tenter un **delete** sur la vue `v_musicien`, j'effacerai docilement le musicien correspondant au tuple de la vue concernée, dans la table sous-jacente.* » Ce qui peut se traduire en SQL par :

```
create rule del_v_musicien as
  on delete to v_musicien
  do instead
    delete from musicien where id=OLD.id
```

Vous pourrez alors allègrement demander :

```
delete from v_musicien where ...
```

Les tuples correspondants dans la table `musicien` seront supprimés.

Voici une autre règle : « *si jamais une demande aussi farfelue que l'insertion d'un tuple dans la vue `v_musicien` me parvient, alors je prendrai gentiment le temps d'insérer les informations dans les deux tables sous-jacentes (`pays` et `musicien`).* » Qui se traduit en SQL par :

```
create or replace rule ins_v_musicien as
  on insert to v_musicien
  do instead
    insert into musicien(id,nom,prenom,origine)
    values (NEW.id,NEW.nom,NEW.prenom,NEW.code);
```

Il vous sera alors possible d'émettre le souhait suivant :

```
insert into v_musicien
  values (1,'Zappa','Franck','US')
```



Un mécanisme tout à fait analogue aux « *rules* » de PostgreSQL peut être mis en place dans le Sgbd Oracle en passant par des ►triggers sur les vues.

§ 6.5 ◀
p. 241



Oracle acceptera docilement d'opérer (**delete**, **update**) sur une vue à condition que la table visée par la requête soit régie par une clé faisant également office de clé dans la vue. Par exemple, dans la définition de `v_musicien` de notre exemple :

```
create view v_musicien as
  select m.id,m.nom,m.prenom,p.code,p.libelle
  from musicien m
  inner join pays p on m.origine=p.code
```

l'attribut `m.id` (clé primaire de la table `musicien`) fait également office de clé pour la vue. Dans ce cas, les deux requêtes :

```
delete from v_musicien where idm=1;
update v_musicien
    set prenom='truc' where idm=3;
```

seront acceptées par Oracle. Dans les autres cas, vous serez gratifiés d'un : « ORA-01779: cannot modify a column which maps to a non key-preserved table ».

6.1.3 Vues matérialisées



Un inconvénient des vues est que par définition, elles ne stockent que le traitement à effectuer. Par conséquent à chaque opération **select**, le Sgbd rassemblera toutes ses forces pour produire comme résultat une relation — production pouvant être complexe si le traitement fait appel à de nombreuses jointures — sur des relations comportant de nombreux tuples. C'est la raison pour laquelle pour les vues très exigeantes en ressources, on fera appel aux vues dites matérialisées, pour lesquelles les données résultant de l'exécution seront stockées.

► § 5.3
p. 177

► § 6.1
p. 208

On peut créer une vue matérialisée comme une vue normale, en ajoutant le mot **materialized** :

```
create materialized view mv_ma_vue_a_moi
as
select ...
```



Il est bien entendu possible de demander à PostgreSQL la liste des vues matérialisées grâce à la commande `\dm` :

```
yahozna=>\dm
Schema |      Name      |      Type      | Owner
-----+-----+-----+-----
public | mv_ma_vue_a_moi | materialized view | lozano
```

Et la commande `\dm mv_ma_vue_a_moi` donnera les détails associés à la vue, en particulier la requête exécutée pour construire les données.



Au moment où ce merveilleux manuel est écrit, PostgreSQL (version 10.x) n'est pas en mesure de mettre à jour automatiquement une vue matérialisée, contrairement à Oracle. Ce dernier propose en effet deux politiques de mise à jour des données :

- mise à jour en temps réel des données lorsque les relations sous-jacentes ont été modifiées;
- mise à jour planifiée (toutes les nuits par exemple) à définir par le concepteur en fonction de l'application.

Nous montrons ici comment il est envisageable dans PostgreSQL de mettre en œuvre la première politique de mise à jour via un ►trigger. Supposons que la vue `mv_mus_ins` soit une vue matérialisée construite comme celle définie au paragraphe 6.1.1 page 208 :

§ 6.5 ◀
p. 241

```
create materialized view mv_mus_ins
as
select m.idMus, m.nom, i.idIns, i.libelle
from musicien m
    inner join saitjouer sj on m.idMus=sj.idMus
    inner join instrument i on sj.idIns= i.idIns
```

Pour des raisons de simplicité de notre propos, supposons que la politique de mise à jour consiste à rafraîchir cette vue matérialisée lorsque la relation `saitjouer` est modifiée avec l'une des requêtes des modifications. On va tout d'abord créer une routine :

```
create or replace function refresh_mv_mus_ins()
returns trigger as $$
begin
    refresh materialized view mv_mus_ins;
    return NEW;
end; $$ language plpgsql;
```

puis en créant le trigger niveau table à proprement parler :

```
create trigger trig_upd_sait_jouer
after insert or update or delete
on saitjouer
execute procedure refresh_mv_mus_instr();
```

Et le tour est joué! Pour chaque **insert**, **update** ou **delete** sur la relation `saitjouer` les données de la vue matérialisée seront régénérées.



Pour s'amuser un peu, notons ici que si vous demandiez : « bon c'est bien gentil, mais en définitive, si on stocke des données dans une vue matérialisée, quelle est donc alors la différence avec une table ??? ». Et bien sachez que ces vues matérialisées apparaîtront sans aucune vergogne, entre autres endroits, dans la liste des tables de votre base de données ;-)

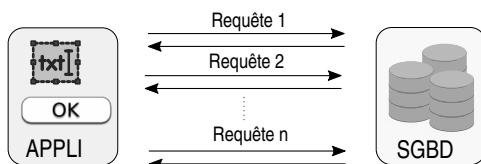
6.2 Avant-propos sur les procédures stockées

6.2.1 Pourquoi stocker des traitements ?

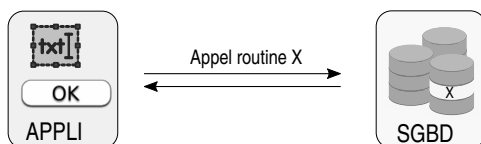
Les deux principales raisons de stocker des traitements dans une base de données sont :

- la simplification de l'écriture des requêtes ;
- la simplification de l'écriture des applications.

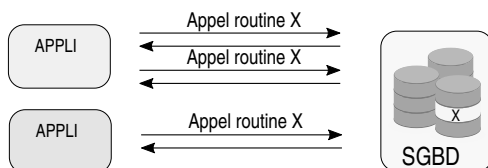
Le concepteur d'une application sera souvent amené à déclencher plusieurs requêtes SQL après que l'utilisateur de cette application a rempli un ou plusieurs champs puis cliqué sur un « bouton OK » :



Dans ce cas, on prendra la sage décision de stocker l'ensemble des requêtes dans une procédure mémorisée dans la base de données — nommons-la X. L'application se contentera alors d'un seul appel à la routine X en question :



L'écriture des applications est simplifiée, puisqu'au sein d'une même application, il suffira d'appeler autant de fois la routine. Et d'autres applications, écrites éventuellement dans un autre langage, pourront également y faire appel :



Un autre intérêt de stocker des requêtes dans une routine réside dans le fait qu'on en fait un bloc atomique, qui sera par conséquent complètement exécuté ou pas du tout. Ceci pourrait ne pas être le cas dans le cas où chacune des requêtes est exécutée depuis une application.

6.2.2 Mise au point dans PostgreSQL

En n'utilisant que la console `psql`, la mise au point consistera à :

1. éditer un fichier source dans votre éditeur de texte préféré, fichier qui contiendra quelque chose du genre :

```
create or replace function X() ...
```

2. exécuter ce fichier grâce à la commande `psql` suivante :

```
yahozna=> \i le_nom_du_fichier
```

la commande `\i` demande simplement à envoyer le contenu du fichier au Sgbd plutôt que de le saisir dans l'interpréteur.

3. exécuter votre nouvelle fonction `X` (elle s'appelle comme ça, regarder ci-dessus) :

```
select X (...);
```

4. reprendre en 1 si nécessaire...

6.3 Un nouveau langage

Le langage `Sql` que nous avons utilisé jusqu'ici est *impératif* : on passe un ordre, le Sgbd s'exécute. Nous allons découvrir dans ce chapitre qu'il existe aussi un volet *procédural* à notre langage préféré, qui fait de lui un « vrai langage » de programmation au sens où on l'entend habituellement (*avec des variables et des boucles et tout et tout*). On gardera à l'esprit que le propos de ce volet de `SQL` est de *définir des traitements*. Chaque traitement à définir constituera ce qu'on appelle une *procédure* ou une *fonction*, et à ce titre:

- aura un *nom* qui lui permettra d'être *appelée* (exécutée);
- pourra accepter des *arguments* ou *paramètres*;
- pourra utiliser des *variables* (zone de stockage temporaire) d'un type donné;
- contiendra les *instructions* définissant le traitement.



Selon les langages de programmation, de tels objets portent des noms différents : fonctions, procédures, routines, sous-programmes, etc. Certains de ces langages, comme le Pascal, font la distinction entre procédures (qui ne renverraient pas de valeur) et fonctions (qui en renverraient une). Dans ce qui suit, et à l'instar du langage C, nous ne ferons pas de distinction et utiliserons l'un des termes ci-dessus, sans préférence.



Le volet procédural de PostgreSQL se nomme aussi `pgplsql` (pour PostgreSQL procedural language SQL). Il est malheureusement propre à ce Sgbd et n'est donc pas standardisé. De manière générale ce volet de SQL fait l'objet de variations selon les Sgbd utilisés. Ce que vous apprendrez ici sera (légèrement) différent dans Oracle ou Microsoft SQL Server.

On peut distinguer deux types de traitements dans une base de données (les deux sont présentés plus loin dans ce chapitre) :

1. les traitements appelés *explicitement* : pour ceux-ci — c'est le cas des procédures stockées — on doit expressément faire un appel pour en demander l'exécution ;
2. les traitements appelés *implicitement* comme les triggers. Pour ceux-ci, c'est un événement particulier (par un exemple, un **delete** sur une relation) qui va en déclencher l'exécution.

Les instructions de la procédure sont écrites à l'intérieur d'un bloc :

```
begin
end
```

On peut y manipuler des variables comme dans n'importe quel langage de programmation. Celles-ci doivent être déclarées en dehors du bloc :

```
declare
  i integer; -- declaration de la variable
begin
  i:=4; -- affectation
end
```

On notera que dans le volet SQL procédural, c'est l'opérateur `:=` qui réalise l'affectation. Le point virgule quant à lui sert de terminateur d'instructions, il faudra donc penser à le mettre aux endroits nécessaires. Par ailleurs, il est fréquent d'avoir à affecter le résultat d'une requête dans une variable grâce à une forme spéciale du **select** :

```
declare
  patronyme text;
begin
  select nom into patronyme
    from musicien where id=28;
  raise NOTICE 'Le_nom_est_:%',patronyme;
end
```




La dernière instruction permet de lever une exception de type « avertissement » avec un message formaté à la `printf` du langage C. Une exception de niveau `NOTICE` n'interrompt pas le déroulement normal des instructions mais envoie le message indiqué dans la console. C'est un moyen simple de déboguer les fonctions que vous mettez au point.

Le **`select into`** permet également de stocker un tuple complet ou partiel. En déclarant une variable comme étant du type d'un tuple de la relation `instrument` :

```
declare
  i instrument%ROWTYPE;
```

on peut stocker dans la variable `i` un tuple de la table `instrument` :

```
begin
  select * into i from instrument where ...
  raise NOTICE 'Instrument_:_%', i.libelle
end
```

Dans l'expression `i.libelle` on notera l'utilisation du point pour accéder aux attributs du tuple `i`. Si vous avez une bonne raison de ne stocker qu'une partie des attributs d'une relation, il faudra passer par :

```
declare
  i record;
```

Vous serez alors libre d'y stocker ce que bon vous semble :

```
select c.bidule, y.truc, y.machin into i
from chose c inner join youpla y on ...
```

Le tuple `i` contiendra alors les attributs `bidule`, `truc` et `machin` et leurs valeurs correspondantes, bien entendu.

Finalement, vous aurez sans doute besoin un jour où l'autre de mettre en œuvre une structure de contrôle conditionnelle, par exemple :

```
select * into i from instrument where ...
if i.libelle like '%saxophone%' then
  -- traitement(s) pour les sax
else
  -- traitement(s) pour les autres
end if;
```

Donc rien à signaler de particulier si ce n'est le point virgule à la fin du **`end if`**. La clause **`else`** est bien sûr facultative.

6.4 Procédure

Forts de la lecture du paragraphe précédent, nous allons examiner un exemple de création d'une routine. Le propos de celle-ci serait d'ajouter dans notre base un saxophoniste. Pour ce faire on devra procéder à l'insertion du nouveau musicien et la mémorisation du fait qu'il sait jouer de tous les saxophones. On supposera que la relation contenant les musiciens a la structure suivante :

Musicien		
<i>idMus</i>	Nom	Prenom

► § 4.5.2
p. 113

L'attribut `musicien.idMus` fait office de clé primaire et on supposera que ses valeurs sont générées automatiquement par une séquence dont le nom est `seq_musicien`. On supposera également l'existence d'une table contenant les instruments dont — vous l'avez deviné — les saxophones :

Instrument	
<i>idIns</i>	libelle
1	Clarinette
2	Clarinette basse
10	Saxophone soprano
12	Saxophone ténor
17	Saxophone baryton
...	...

Enfin, la relation `saitjouer` est construite comme expliqué au paragraphe 4.7.2 page 125. À savoir :

saitjouer	
<i>idMus</i>	<i>idIns</i>

Pour revenir à nos moutons, nous devons écrire une fonction pour ajouter un nouveau saxophoniste :

```
create or replace function ajouter_saxophoniste (
    nouveau_nom    in text ,
    nouveau_prenom in text
) returns void
```

```
as
$$
begin
    null; -- rien pour l'instant
end;
$$
language plpgsql;
```

6.4.1 Appel d'une fonction

Cette fonction pourra être appelée dans une application en écrivant par exemple :

```
select ajouter_saxophoniste('Underwood','Ian');
```

Ou dans une autre procédure :

```
begin
...
    perform ajouter_saxophoniste('Underwood','Ian');
...
end;
```

6.4.2 Arguments d'une fonction

Les deux arguments nom et prenom de la routine sont passés en mode « in », c'est-à-dire qu'il s'agit de données transmises en « entrée » du traitement effectué par la routine. En d'autres termes, la fonction reçoit ces deux informations, les utilise pour son traitement, mais *n'a pas le droit de les modifier*. D'ailleurs si la routine contenait une instruction :

```
nouveau_nom:='bidule';
```

Le SGBD vous gratifierait d'un :

```
ERROR: "$1" is declared CONSTANT
```

au moment de la création de la routine. Ce message vous indique que le premier argument (\$1 et donc nouveau_nom) est supposé ne pas être modifié.

6.4.3 Corps d'une fonction : les saxophonistes !

Notre fonction dont le but est d'ajouter un saxophoniste doit finalement :

1. ajouter un nouveau musicien ;
2. ajouter le fait que ce musicien sait jouer de tous les saxophones.

Pour la première partie, fastoche :

```
create or replace function ajouter_saxophoniste
(
    nouveau_nom      in text ,
    nouveau_prenom in text
) returns void
as $$
begin
    -- insertion du nouveau musicien
    insert into muscien(nom,prenom)
        values (nouveau_nom,nouveau_prenom);
    -- puis dire qu'il joue de tous les saxs
end;
$$ language plpgsql;
```

Pour la partie « dire que ce nouveau joue de tous les sax » il va falloir faire usage des quelques petites choses apprises lors du chapitre précédent traitant du volet LMD de SQL. En particulier, nous avons vu au paragraphe 5.2.3 page 174 que la commande **insert** pouvait insérer dans une relation le résultat d'un **select** avec la forme :

```
insert into ma_table_a_moi_que_j_ai select ...
```

D'autre part nous avons vu (§ 5.1.5 page 157) qu'un **select** pouvait contenir des attributs constants. Ainsi par exemple, la requête :

```
select 18,idi from instrument
      where libelle ilike '%saxophone%'
```

renverra une relation composée de deux attributs, l'un valant toujours 18 et l'autre la valeur des identifiants des instruments saxophones :

18	10
18	12
18	17

Enfin pour répondre à la question : *spécifier que le nouveau musicien sait jouer de tous les saxs*, il faudrait arriver à exprimer qu'en lieu et place de la valeur 18 ci-dessus, on spécifie le numéro du dernier musicien qu'on vient d'insérer dans la table `musicien`. Dans la mesure où l'insertion d'un nouveau tuple dans cette table provoque l'incrémentation de la séquence `seq_musicien`, on peut obtenir la valeur du numéro du dernier musicien inséré en écrivant :

```
currval('seq_musicien')
```

Finalement, la routine complète⁴ peut s'écrire :

```
begin
  -- insertion du nouveau musicien
  insert into musicien(nom,prenom)
    values(nouveau_nom,nouveau_prenom);
  -- le nouveau joue de tous les saxs
  insert into saitjouer(idm,idi)
    select currval('seq_musicien'), idi
    from musicien
    where libelle ilike '%saxophone%';
end;
```

6.4.4 Valeur renvoyée par une fonction

Indépendamment des arguments transmis, une fonction pourra renvoyer ou non une valeur dont le *type* sera précisé grâce au mot clé **returns** dans l'entête :

```
create function hop(...) returns int as ...
```

Une telle fonction pourra être appelée interactivement :

```
yahozna=> select hop(...);
   hop
-----
     6
(1 row)
```

le résultat étant donc une relation constituée d'un seul tuple et un seul attribut, ou dans le contexte d'une fonction :

4. Uniquement son corps.

```

declare
    montentier int;
begin
    monentier:=hop(...)+7;
end

```

Quant à la fonction `hop` elle-même, son corps devrait contenir au moins un appel au mot clé **return** pour faire le retour effectif de la valeur :

```

create function hop (...) returns int
...
begin
    ...
    return 7*j; -- valeur de retour
end

```



Dans la mesure où le mot clé **return** est obligatoire dans la déclaration d'une fonction, on utilisera le type **void** (le vide en anglais) pour indiquer qu'aucune valeur ne sera retournée.



Pour aller plus loin autour du concept de valeur renvoyée, les paragraphes 6.4.9 page 231 et 6.4.10 page 233 présentent quelques techniques permettant d'écrire des fonctions renvoyant plus d'une valeur et des tuples.

6

6.4.5 Autre exemple pour se dégourdir le clavier

Dans le paragraphe 7.3 page 298 consacré aux index est présentée une relation contenant plusieurs millions de tuples générés aléatoirement. Les données ressemblent à ce qui suit :

id	libelle
6391103	YITSBJKHWW
8904368	ZYMXRTLPEF
1701551	LIWZMOAPFY
4038765	VGUMOPZTOZ

Nous expliquons ici comment il est possible d'y parvenir grâce au langage SQL. Vos serveurs ont d'abord créé une fonction permettant de générer un caractère aléatoirement :

```

create or replace function rand_char
(
    c_min in int,
    c_max in int
) returns char as
$$
begin
    return chr(c_min
               + floor(100*random())::int
               % (c_max-c_min));
end;
$$
language plpgsql;

```

Fichtre, me direz-vous! Qu'est ce que ce charabia derrière le **return**? Cette fonction s'appuie sur deux idées :

1. on peut générer un nombre aléatoire grâce à la fonction pré-définie `random`, qui, nous le voyons ci-dessous, renvoie un nombre à virgule tiré au hasard entre 0 et 1, ainsi :

```

yahozna=> select random();
           random
-----
0.424025531392545

```

2. la fonction `chr` renvoie le caractère qui correspond au code Ascii qu'on lui transmet, ainsi :

```

yahozna=> select chr(65);
           chr
-----
A

```

La fonction ainsi définie attend deux arguments qui correspondent aux codes Ascii minimum et maximum des caractères qui seront produits. On peut ainsi l'appeler comme ceci :

```

yahozna=> select rand_char(88,120);
           rand_char
-----
m

```

Sur la génération du caractère lui-même le principe est le suivant :

- on génère un nombre aléatoire (entre 0 et 1) avec `random()` ;
- en multipliant par 100 on obtient un nombre entre 0 et 100 ;
- l'expression `floor(100*random())` tronque ce nombre à virgule vers l'entier immédiatement inférieur ;
- l'expression `floor(...)::int % (X)` renvoie le reste de la division entière du nombre tiré au hasard par X. Donc un nombre pris au hasard dans l'intervalle 0 à X-1 ;
- finalement, l'expression :

```
c_min
+ floor(100*random())::int % (c_max-c_min)
```

renvoie un nombre entre :

- `c_min + 0`
 - `c_min + (c_max - c_min - 1)`
- et donc entre `c_min` et `c_max-1`.

Pour poursuivre la construction des chaînes aléatoires, nous proposons une fonction `rand_str` assemblant un nombre défini de caractères. Une boucle pour comme celle qui suit :

```
str:=''
for i in 1 .. len loop
    str:=str||rand_char(65,91);
end loop;
```

permet de concaténer plusieurs caractères (`len` pour être précis) dans la variable `str`. Cette variable devra bien entendu être déclarée au début de la routine :

```
declare
    str text;
```

L'entête de la routine sera :

```
create function rand_str(len in integer)
returns text
```

Nous laissons au lecteur le soin de rassembler les morceaux pour que cette fonction puisse s'exécuter comme suit :

```
yahozna=> select rand_str(20);
           rand_str
-----
FSZEVXJBVBVUSMRCZHMK
(1 row)
```


Finalement, afin de générer autant de tuples qu'on le souhaite pour notre relation `truc`⁵ grâce à laquelle nous testerons l'utilisation des index :

```
create table truc(  
    id serial primary key,  
    libelle varchar(10)  
);
```

il suffira de créer une fonction paramétrée par le nombre de tuples à créer. Le corps d'une telle fonction sera tout simplement :

```
for i in 1..nbtuples loop  
    insert into truc(libelle)  
        values(rand_str(10));  
end loop;
```

et son entête :

```
create or replace function gen_rand_data(  
    nbtuples in integer  
)
```

Et l'appel :

```
yahozna=> select gen_rand_data(100);
```

ajoutera 100 chaînes de caractères de longueur 20, générées aléatoirement, à la relation `truc`.

6.4.6 Surcharge des procédures/fonctions

Lorsque vous créez une procédure :

```
create or replace function f(x in int) ...
```

Alors un appel à :

```
create or replace function f(x in float) ...
```

créera une nouvelle fonction `f`.

5. Le type `serial` permet de créer une séquence et de l'associer automatiquement à la table, en vous épargnant les opérations présentées à la section 4.5.2 page 113



Ceci correspond au mécanisme de la surcharge des fonctions (function overloading) et est disponible dans certains langages de programmation dont celui de PostgreSQL. Celui-ci vous autorise donc à créer des fonctions portant le même nom, mais avec des arguments en nombre et type différents. Ce n'est pas le cas d'Oracle.

Pour vous en convaincre, vous verrez apparaître ces deux fonctions dans la console :

```
yahozna=> \df f
```

List of functions				
Schema	Name	Res data type	Arg data types	Type
public	f	integer	x float	normal
public	f	integer	x integer	normal

C'est pourquoi, pour en supprimer une, il ne suffira pas de préciser son nom, mais aussi le type des arguments :

```
drop function f(float)
```

6.4.7 Curseurs

6

Il est parfois nécessaire dans une procédure d'effectuer une boucle sur tout ou partie des tuples d'une relation. Dans ce cas l'outil idoine est le *curseur*. PostgreSQL offre une version simple de ces objets permettant de réaliser des traitements spécifiques à chaque itération.



Ceux qui auront suivi jusqu'ici pourraient souligner le fait qu'une mise à jour d'une partie des tuples peut tout à fait être réalisée grâce à une requête du genre :

```
update ma_table where ...
```

Il existe cependant bien des situations où cette forme n'est pas suffisante pour exprimer ce que l'on souhaite. Par exemple, dans le cas où il est nécessaire d'exécuter une ou plusieurs procédures pour chaque tuple à mettre à jour ou lorsque le traitement varie selon les données rencontrées à chaque tuple.

Le squelette de base pour itérer sur un sous-ensemble de tuples consiste d'abord à déclarer une variable qui sera le réceptacle de chaque tuple :

```
declare
  R record;
```

Puis d'écrire la boucle à proprement parler :

```
for R in select ... loop
  -- do what the f... you want with R
end loop;
```

Par exemple, pour parcourir les musiciens sachant jouer du saxophone, nous pourrions écrire quelque chose du genre :

```
declare
  M record;  -- un musicien
begin
  for M in select id_musicien,nom,prenom
            from v_musicien_instrument
            where libelle like '%sax%'

  loop
    raise NOTICE '%_sait_jouer_du_sax_', M.nom;
  end loop;
end;
```

Pour l'instant vous avez une jolie boucle qui affiche le nom de chaque musicien sachant souffler dans un saxophone. J'imagine que vous êtes ravi de savoir mettre en œuvre une telle prouesse. L'étape suivante est d'insérer pour chaque itération le traitement de vos rêves. Par exemple :

```
for M in select id_musicien,nom,prenom
            from v_musicien_instrument
            where libelle like '%sax%'

loop
  raise NOTICE 'Sait_jouer_du_sax_%s_',M.nom;
  insert into saxplayer(ids,nom,prenom)
    values (M.id_musicien,M.nom,M.prenom);
end loop;
```

Ici on insère tous les saxophonistes dans la table (à créer) saxplayer. Comme indiqué plus haut, nous aurions pu réaliser cette opération en une requête :

```
insert into saxplayer(ids,nom,prenom)
  select id_musicien,nom,prenom
    from v_musicien_instrument
    where libelle like '%sax%'
```

Mais encore une fois, le curseur sera inévitable pour toutes les situations où les traitements à réaliser à chaque itération ne sont pas réduits à un simple **insert**.

6.4.8 Exceptions

Une *exception* est un mécanisme — présent dans la plupart des langages de programmation — dont l'objet est de laisser la possibilité à un programme d'*intercepter une erreur* (dans notre cas une violation de contrainte par exemple) et de réagir à cette erreur (par un traitement spécifique) plutôt que de faire échouer le programme, ce qui est le comportement par défaut.

À titre d'exemple introductif, on intercepte une erreur de type violation d'une contrainte d'unicité, comme ceci :

```
begin
  -- vos traitements ici
  -- ...
exception
  when unique_violation then
    -- ici ce que vous voulez faire lorsque
    -- une des requêtes génère une violation
    -- de contrainte d'unicité
end ;
```

6

Exemple : insérer ou modifier

Un exemple typique est l'écriture d'une requête dont l'objet est de réaliser le traitement suivant :

- insérer un nouvel individu s'il n'existe pas
- le modifier s'il existe

Le traitement en question doit être implémenté grâce à une fonction que l'on déclarera comme suit :

```
create or replace function maj_individu (
  inom      in text ,
  iprenom  in text ,
  iemail    in text
) returns void
```

La procédure en question attend donc trois arguments dont l'email pour lequel on fera l'hypothèse importante qu'il existe une contrainte d'unicité. On écrira donc :

```

begin
    -- on tente une insertion
    insert into personne(nom, prenom, email)
        values (inom, iprenom, iemail);
exception
    when unique_violation then
        -- si la personne existe ,
        -- on modifie son nom et son prénom
        update personne
            set nom=inom, prenom=iprenom
            where email=iemail;
end;

```

Ainsi avec une telle routine, la requête :

```
select maj_individu('Loz', 'Vince', 'vl@truc.org')
```

Effectuera l'un des traitements suivants :

- soit l'individu dont l'email indiqué existe, dans ce cas le nom et le prénom seront positionnés selon les arguments passés;
- soit il n'existe pas d'individu avec cet email, dans ce cas il sera ajouté dans la table.



On notera qu'il est possible d'intercepter plusieurs exceptions dans un même bloc :

```

begin
    exception
    when unique_violation then
    when foreign_key_violation then ...

```



Nous vous invitons à jeter un œil sur la documentation de PostgreSQL pour voir la liste exhaustive des différentes exceptions qu'il est possible d'intercepter.

Gérer les insert into

Lorsque dans une procédure vous souhaitez stocker dans une variable un des attributs du tuple résultat, vous écrirez :

```

declare
    truc int;
begin
    select machin into truc from select ...
...

```

Deux erreurs classiques générées par ce type de requête peuvent être interceptées :

- le fait que le **select** *ne renvoie aucune valeur* ;
- le fait que le **select** *renvoie plus d'une valeur*.

Le code SQL ci-dessous réagit à ces deux situations :

```
begin
  select machin into truc from select ...
exception
  when no_data_found then ...
  when too_many_rows then ...
end ;
```

Exceptions & curseurs

Lors d'une boucle d'instructions associée à un curseur, on distinguera :

```
begin
  for T in (select ...) loop
    traitement1 (...);
    traitement2 (...);
  end loop
exception
  when erreur_impardonnable then ...
end ;
```

qui quittera définitivement le parcours du curseur dès que l'erreur impardonnable surviendra, et :

```
begin
  for T in (select ...) loop
    begin
      traitement1 (...);
      traitement2 (...);
    exception
      when erreur_impardonnable then ...
    end loop
end ;
```

qui passera gentiment au traitement suivant dans le curseur, lorsque l'erreur impardonnable pointera le bout de son nez.

Gérer ses propres exceptions

Enfin, il est important de savoir qu'on peut créer ses propres exceptions : pour ce faire il faut d'abord « lever » soi-même l'exception dans la routine :

```
begin
    ...
    if ... then
        raise exception using
            errcode='MYERR',
            message='Non_mais_ca_va_pas_la_tete',
            hint='Essaie_encore...';
```

Cette exception pourra être interceptée par une autre fonction de la même manière que précédemment, en précisant le code spécifique :

```
begin
    ...
    exception
        when 'MYERR' then
            ...
end;
```



PostgreSQL définit une liste d'erreurs composées chacune d'un code de 5 caractères et d'un libellé en toutes lettres. Ces erreurs sont regroupées selon des classes, par exemple : erreurs autour des contraintes d'intégrité, erreurs de syntaxe, erreurs par manque de ressources suffisantes, etc. Votre code d'erreur devra contenir 5 caractères et ne pas faire double emploi avec les codes existants.

6.4.9 Renvoyer plusieurs valeurs : out argument



Nous exposons ici le mécanisme des arguments « out » de PostgreSQL qui sont régis par une syntaxe identique à celle d'Oracle, mais avec une utilisation « quelque peu singulière ». Oracle quant à lui, suit une logique classique pour la manipulation de ce type d'argument.

Il n'est pas rare que les procédures stockées élaborent des résultats et les renvoient à d'autres procédures. Imaginons par exemple que vous ayez besoin de calculer la somme et la différence de deux valeurs. Pour y parvenir vous pourriez écrire une procédure comme celle qui suit :

```

create or replace function somme_diff
(
  a in int, b in int,
  s out int, d out int
)
as
$$
begin
  s:=a+b; -- calculer la somme
  d:=a-b; -- calculer la différence
end;
$$
language plpgsql;

```

Dans le plus fidèle respect du dogme de l'église de la programmation procédurale du 16^e jour, un appel à une telle procédure devrait s'écrire comme ceci :

```

declare
  som int; dif int;
begin
  somme_diff(2,3,som,dif);
end;

```

6

Après l'appel à la routine `somme_diff`, les variables `som` et `dif` contiennent respectivement les valeurs 5 et -1.



Dans PostgreSQL, et contrairement à Oracle qui a adopté une syntaxe plus « orthodoxe », on appelle la fonction :

```

yahoyna=> select * from somme_diff(2,4);
 s | d
---+---
 6 | -2
(1 row)

```

Dans le contexte d'exécution d'une autre routine, celle-ci pourrait être appelée comme suit (`dif` et `som` ayant été déclarée au préalable) :

```

begin
  select * into som,dif from somme_diff(2,3);
end;

```


6.4.10 Renvoyer des tuples



Dans certaines situations, l'appel à une vue ne suffit pas à exprimer ce que l'on souhaite avec suffisamment de souplesse :

```
select * from la_super_vue_que_j_ai_creee
where ...
```

Dans ce cas il est envisageable de passer par une fonction renvoyant des tuples. Cette fonction pourra attendre des paramètres et être appelée dans le cadre d'une requête **select**.

Nous nous appuyerons ici sur la vue étudiée au chapitre 5, paragraphe 5.1.7 page 163, dont l'objet est de savoir quels sont les musiciens qui pouvaient potentiellement rencontrer ceux du XIX^e siècle. Nous étions arrivés à la définition suivante :

```
select m1.nom as nom1, m2.nom as nom2
from musicien m1 cross join musicien m2
where m1.idmus!=m2.idmus
and m1.naissance<='1900-01-01'
and m2.naissance>'1900-01-01'
and
overlaps(m1.naissance+'16_years'::interval,
          m1.deces-'5_years'::interval,
          m2.naissance+'16_years'::interval,
          m2.deces-'5_years'::interval)
```

Supposons maintenant que nous souhaitions *rendre variable* les différentes valeurs entrant en jeu dans la requête :

- l'âge minimum pour que la rencontre soit acceptable ;
- le nombre d'années minimum exigé avant le décès pour que la rencontre soit considérée comme possible ;
- la date charnière partageant l'ensemble des musiciens en deux ensembles. Ici il s'agit du 1^{er} janvier 1900 permettant de ranger d'un côté DEBUSSY, STRAVINSKY et VARÈSE, et de l'autre le reste de la troupe.

Si nous sommes capables de créer une routine retournant des tuples avec les paramètres ci-dessus, une requête y faisant appel ressemblera à :

```
select * from
rencontre('1900-01-01','16_years','5_years') r
inner join ...
where ...
```

Reste bien sûr à écrire une telle routine... On commencera par en définir l'entête :

```
create or replace function rencontre
(
    charniere in date ,
    delta_naissance in interval ,
    delta_deces in interval
)
returns setof t_rencontre
as $$ ... .. $$ language plpgsql;
```

Dans cette définition on spécifie :

- le type des arguments : la date charnière et les deux décalages, naissance et décès;
- et le type de la valeur renvoyée qui est donc ici : *ensemble de t_rencontre*.

On définit le type t_rencontre comme suit :

```
create table t_rencontre(nom1 text , nom2 text)
```

Le corps de la routine elle-même va être construit en utilisant un curseur parcourant tous les tuples de la requête **select** dont on a donné la définition au début de cette section :

6

```
declare
    renc t_rencontre%ROWTYPE;
    R record;
begin
    for R in <select qui tue> loop
        renc.nom1:=R.nom1;
        renc.nom2:=R.nom2;
        return next renc;
    end loop
end;
```

Il faut comprendre dans cette construction que pour chaque tuple R (contenant le nom des deux personnes qui peuvent se rencontrer) :

- on stocke les deux attributs *nom1* et *nom2*, dans le tuple *renc*
- l'instruction **return next renc** ajoute celui-ci au résultat déjà constitué (je vous rappelle pour ceux qui se sont endormis, que l'on renvoie un ensemble de tuples).

Dans le <select qui tue> on prendra soin de remplacer les expressions '16 years' et '5 years' par les paramètres que nous avons

déclarés dans l'entête de la routine. Finalement, le *⟨select qui tue⟩* et sa boucle s'écriront donc :

```
begin
  for R in
    select m1.nom as nom1, m2.nom as nom2
    from musicien m1 cross join musicien m2
    where m1.idmus!=m2.idmus
    and m1.naissance<=charniere
    and m2.naissance> charniere
    and
    overlaps (m1.naissance+delta_naissance,
              m1.deces      -delta_deces,
              m2.naissance+delta_naissance,
              m2.deces      -delta_deces)

  loop
    ...
  end loop
```

Une fois cette routine créée dans la base, chacun pourra se fendre de :

```
select * from rencontre('1950-12-28',
                        '144_months',
                        '5_minutes')
```

ou tout autre requête satisfaisant sa curiosité.

6.4.11 Calculer avec des nombres à virgule



Cette section prend le prétexte d'étudier les nombres à virgule de notre Sgbd (PostgreSQL) pour créer quelques procédures et ainsi vous proposer d'autres exemples d'utilisation de SQL en tant que langage procédural.



Vous ne comprendrez pas grand-chose aux subtilités exposées ici si vous n'avez pas lu et digéré le chapitre exposant les principes du stockage des ►nombres à virgule dans un système informatique.

§ 2.3 ◀
p. 28

Nous avons vu que les flottants ont comme caractéristiques :

1. de ne pas permettre une représentation exacte des nombres (y compris ceux aussi simple que 0,3 — cf. § 2.3.4 page 31) car dotés d'une *précision* ;

2. en contrepartie leur représentation est relativement *économe en mémoire* et l'arithmétique sur de tels nombres est directement opérée par les micro-processeurs, garantissant *vitesse d'exécution* pour les calculs.

À l'inverse, les nombres décimaux codés binaire permettent une représentation exacte moyennant un stockage plutôt gourmand ainsi qu'une relative lenteur des calculs. Dans ce qui suit, les types auxquels on fera appel seront :

1. pour les nombres à virgule flottante : `double precision`
2. pour les décimaux codés binaires : `numeric`

Précision des calculs

Pour illustrer la différence entre les deux représentations (flottant et exacte), nous proposons de cumuler une petite quantité Δ à une valeur de départ et ce, un grand nombre de fois. Par exemple :

« ajouter 10000 fois 0,00001 à la valeur 0 »

Si on accepte le fait que la valeur 0,00001 n'est pas représentée exactement, nous devrions faire apparaître une erreur de calcul. Le code ci-dessous réalise ce cumul grâce à une boucle **for** :

```
begin
    val:=0
    for i in 1..iter loop
        val:=val+delta;
    end loop;
end;
```

à condition bien sûr d'imaginer que :

- iter est le nombre de fois où l'on veut faire ce cumul;
- delta correspond à la petite valeur Δ .

Pour encapsuler tout ceci dans une fonction, on pourra écrire :

```
create or replace function cumul_flottant (
    init in double precision,
    delta in double precision,
    iter in int ) returns double precision
as $$ ... $$ language plpgsql;
```

L'idée étant qu'un appel :

```
select cumul_flottant(0, 0.0001, 10000)
```

demandera à ajouter 10000 fois la valeur 0,0001 à la valeur initiale 0 et renverra le résultat. Pour y parvenir, le corps de la fonction (ce qui est entre la paire de \$\$) devra contenir :

```
declare
    val double precision;
begin
    val:=init;
    for i in 1..iter loop
        val:=val+delta;
    end loop;
    return val;
end;
```

Et comme prévu :

```
yahozna=> select cumul_flottant(0, 0.001, 1000);
           cumul_flottant
-----
                        1
```

Mais :

```
yahozna=> select cumul_flottant(0, 0.00001, 100000);
           cumul_flottant
-----
0.9999999999998084
```

La même routine adaptée avec un type exact fait le boulot correctement (nous ne donnons ici que l'entête de la fonction puisque le corps est le même, sauf pour le type de la variable val) :

```
create or replace function cumul_numeric (
    init  in numeric ,
    delta in numeric ,
    iter  in int
) returns numeric
```

c'est-à-dire :

```
yahozna=> select cumul_numeric(0, 0.00001, 100000);
           cumul_numeric
-----
1.00000000
```

Performance des calculs

► § 7.3
p. 298

La différence en termes de temps de calcul peut-être illustrée en demandant au planificateur de requête de chronométrer le temps de calcul :

```
yahozna=> explain analyze
yahozna-> select cumul_numeric(0, 0.00000001, 100000000);
               QUERY PLAN
-----
Result  ...
Planning time: 0.021 ms
Execution time: 36394.350 ms
```

Soit un peu plus de 36 secondes pour 100000000 itérations. Et :

```
yahozna=> explain analyze
yahozna-> select cumul_flottant(0, 0.00000001, 100000000);
               QUERY PLAN
-----
Result  ...
Planning time: 0.025 ms
Execution time: 24565.706 ms
```

6

Soit environ 24 secondes pour le même nombre d'itérations. On dirait donc que le calcul en flottant est à peu près 1,5 fois plus rapide.

Espace mémoire

► § 2.3
p. 28

Pour montrer la différence de stockage entre les deux approches, rappelons tout d'abord que :

- les flottants sont stockés avec un nombre fixe de bits : 64 bits pour la double précision de PostgreSQL, comportant signe, mantisse et exposant ;
- dans le décimal code binaire, chaque chiffre est codé en binaire. Par conséquent la taille dépend du nombre de chiffres composant la valeur représentée.

La table ci-dessous contenant pour plusieurs valeurs de n , le carré, le cube et cette valeur à la puissance 4, va nous permettre d'estimer la taille des différentes types :

puissance			
n	carre	cube	p4
2	4	8	16
0.5	0.25	0.125	0.0625
...

Nous pouvons créer deux tables comme celle-ci, l’une avec le type flottant double precision et pour l’autre, le type numeric. Puis profitons du moment pour écrire une petite procédure permettant de remplir ces deux tables avec des valeurs tirées au hasard. Voici le début du corps de cette procédure :

```
-- on efface tout et on recommence
delete from puissancef;
delete from puissancec;
for i in 1..nmax loop
    rand=random();
    insert into puissancef(n) values(rand);
    insert into puissancec(n) values(rand);
end loop;
```

Puis il faudra continuer en calculant les trois puissances en ajoutant une commande comme celle-ci :

```
update puissancef set carre = n*n,
                    cube   = n*n*n,
                    p4      = n*n*n*n;
```

Sans oublier d’exécuter cette commande pour la table puissancec. Alors, la commande⁶ :

```
select n,carre,p4 from puissancef
```

affichera les valeurs de n puis de n^2 , n^3 et n^4 calculées avec des nombres à virgule flottante, dont voici un extrait :

n	carre	p4
0.899297813419253	0.808736557220649	0.654054818985108
0.566603062208742	0.321039030104323	0.103066058850325
0.250962065998465	0.0629819585702178	0.00396672710534063
0.0258549810387194	0.000668480044512541	4.46865569911488e-07
0.691908655688167	0.478737587816206	0.229189677988079
0.853206010069698	0.727960495619054	0.529926483181938

6. Désolé pas assez de place pour le cube...

0.941068713087589		0.88561032275233		0.784305643765486
0.284109318163246		0.0807181046671845		0.00651541242106255
0.899579461663961		0.809243207847623		0.65487456944751
0.571924432180822		0.327097556125356		0.10699281122318

La commande ci-dessous, qui n'affiche pour des raisons de place que la puissance 4 :

```
select p4 from puissancen
```

nous permet de « comparer visuellement » la précision de chacune des valeurs (ici uniquement pour la puissance 4) :

```
-----+
0.654054818985109014195443953867148145057386602713498803404081 |
0.103066058850324766960216763495463853105154717742380902734096 |
0.003966727105340644291986360982304796466070899685932796250625 |
0.0000004468655699114870642426076538336182139309238623256622900496 |
0.229189677988079965399265249501085554571406554413948071972321 |
0.529926483181938162801233038590078547448725323472557387929616 |
0.784305643765487712874047472869509321050282356024157327392241 |
0.006515412421062559180613094902232423792592311197262160458256 |
0.654874569447509344200381545198304863254281374812215111049441 |
```

Nous avons ici l'illustration que la valeur 0.566603062208742 (2^e ligne du **select** sur la relation *puissancecf* ci-dessus) élevée à la puissance 4 sera stockée : 0.103066058850325 en flottant avec le type double precision, et :

```
0.103066058850324766960216763495463853105154717742380902734096
```

avec le type *numeric*, valeur exacte de 0,566603062208742⁴.

Le titre de cette section étant « espace mémoire » et la question vous brûlant la langue : *quel est donc le rapport des tailles des deux relations* ?

- *puissancecf* qui contient pour chaque tuple, 4 flottants double précision (la documentation de PostgreSQL nous permet de dire que chaque flottant occupe 64 bits)
- *puissancen* dont les tuples sont composés de 4 valeurs de type *numeric*

Voici ce que donne PostgreSQL quand on l'interroge⁷ sur ce sujet :

```
yahozna=> select pg_relation_size('puissancecf');
pg_relation_size
-----
222150656
```

7. Voir 7.5.3 page 312 pour une présentation des outils permettant d'obtenir la taille d'un objet de la base.

Soit environ 200 Mo. Et :

```
yahozna=> select pg_relation_size('puissancef');
pg_relation_size
-----
          416686080
```

soit pratiquement le double.

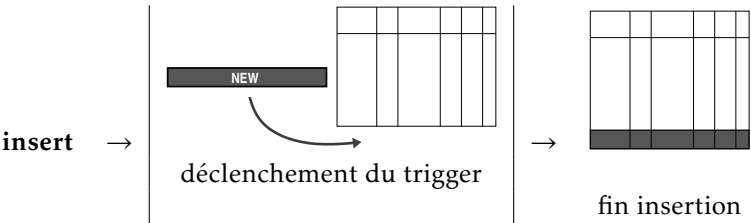
6.5 Trigger

Un trigger est un *traitement associé à une table* qui peut être déclenché par un événement de type LMD (ceux qu’on étudiera) mais aussi par un événement de type LDD. Ainsi, pour les trois événements **insert**, **update** et **delete** on peut définir un traitement qui s’exécutera pour *tous les tuples concernés* par l’opération. Ces triggers sont des triggers « niveau tuple » (ou *row level trigger* dans la langue de William⁸). Ils traduisent l’idée suivante :

- avant (ou après) chaque insertion (ou effacement ou modification)
- exécuter le traitement *T*
- pour chaque tuple concerné

6.5.1 Trigger lors d’un **insert**

Lorsqu’on pose un trigger pour un **insert**, on accède, pendant l’exécution de celui-ci, au tuple **NEW** *avant* son insertion. Ce tuple contient les informations que le **insert** s’appête à insérer.



Imaginons un exemple : pour notre table des musiciens on souhaite s’assurer de deux choses :

8. SHAKESPEARE, faut suivre les gars⁹ ...
9. Et les filles, bien-sûr.

- que les noms soient stockés avec leur première lettre en majuscule et le reste en minuscule
- que des espaces qui auraient malencontreusement été saisis en début ou fin de nom soient supprimés

Ces deux « nettoyages » peuvent être délégués à des triggers. On va tout d’abord associer à la table `personne` un trigger. C’est-à-dire qu’on va écrire en SQL :

```
create trigger trig_formater_nom
before insert on musicien
for each row
execute procedure formater_nom();
```

Incantation chamanique signifiant qu’il faudra exécuter la procédure `formater_nom()` avant chaque appel à la requête **insert** sur la table `musicien`. Le problème est que cette routine n’existe pour l’instant pas dans notre base. Par conséquent l’incantation ci-dessus échouera lamentablement avec le message :

```
ERROR:  function formater_nom() does not exist
```

qui est on ne peut plus clair. Il nous faut donc créer la fonction en question :

```
create function formater_nom() returns trigger
as
$$
begin
    NEW.nom:=initcap(trim(NEW.nom));
    return NEW;
end;
$$
language plpgsql;
```

Une fois cette fonction créée on pourra appeler la requête :

```
create trigger trig_formater_nom ...
```

qui avait échoué lors de notre première tentative faute d’existence de la fonction. Examinons maintenant d’un peu plus près cette fonction. Elle fait appel à deux outils dans la catégorie traitement de chaînes de caractères :

- `initcap(...)` mettant les premières lettres de chaque mot en majuscules et le reste en minuscules;
- `trim(...)` capable de supprimer en tête ou en fin de chaîne, n’importe quel caractère, par défaut les espaces.

On notera également que dans cette procédure particulière, qui doit retourner le type `trigger`, on a accès au tuple en cours de modification via le symbole `NEW`. Ce tuple contient le nouveau tuple que l'on s'apprête à insérer. Ainsi si l'on écrit :

```
insert into personne(id,nom,prenom,origine)
values(17,'_Jarra_', 'Victor', 'CH')
```

alors on aura pendant l'exécution du trigger, dans le tuple `NEW` :

id	nom	prenom	origine
17	jaRRA	Victor	CH

Ainsi dans la fonction `formater_nom()`, `NEW.nom` contient au départ `_Jarra_`. Puis :

- `trim(' jaRRA ')` renvoie `'jaRRA'`
- et `initcap('jaRRA')`, c'est-à-dire `initcap(trim('_jaRRA'))`, renvoie `Jarra'`

Finalement le tuple `NEW` contient :

17	Jarra	Victor	CH
----	-------	--------	----

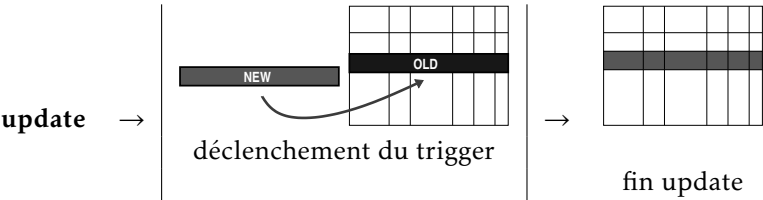
Et c'est ce tuple qui sera finalement inséré.



La fonction appelée par le trigger doit retourner le type **trigger** et vous aurez remarqué qu'elle n'accepte pas d'argument. Les seules informations accessibles par cette fonction seront, le cas échéant, le tuple `NEW` et le tuple `OLD`.

6.5.2 Trigger lors d'un **update**

Pour les triggers se déclenchant sur un **update**, on a accès pendant l'exécution de celui-ci, aux tuples `NEW` et `OLD` *avant* la mise à jour effective :



Nous vous présentons ici quelques exemples d'application des triggers sur une requête **update**. Imaginons que l'on veuille garder trace des dates de modifications sur notre table des musiciens. Pour cela, on ajoute un champ `date_modif` dans notre table :

```
alter table musicien date_modif timestamp
```

La table devient donc :

id	nom	prenom	...	date_modif
----	-----	--------	-----	------------

La première date peut-être insérée automatiquement en stipulant une valeur par défaut pour l'attribut :

```
alter table musicien
alter date_modif set default now()
```

La fonction `now()` nous renvoie la date actuelle sous forme d'une valeur de type `timestamp`, c'est-à-dire à la seconde près.

► § 4.8.2
p. 136

Pour ce qui concerne la *mise à jour* automatique de l'attribut, c'est le mécanisme des triggers qui va nous y aider. On procédera comme précédemment en créant dans un premier temps la fonction dont le trigger déclenchera l'exécution :

```
create function stocker_date_modif()
returns trigger as $$
begin
    NEW.date_modif:=now();
    return NEW;
end;
$$ language plpgsql;
```

Puis on crée le trigger lui-même :

```
create trigger trig_upd_musicien_date
before update on musicien
for each row
execute procedure stocker_date_modif();
```

Grâce au trigger la date à la seconde près de toute modification sur la table `musicien` sera enregistrée.

Pour montrer l'utilisation du tuple `OLD` on pourrait envisager d'interdire la modification des tuples de la table `musicien` dont la dernière mise à jour date de moins d'une minute¹⁰. Pour ce faire il suffit, dans la routine précédente d'écrire¹¹ :

10. Oui oui c'est une idée bizarre...

11. Seul le bloc d'instruction est indiqué ici, le reste est inchangé.

```

begin
  -- si la derniere modification
  -- date de moins d'une minute
  if now() - OLD.date_modif() < '60sec' :: interval
  then
    return false;
  end if;
  NEW.date_modif=now();
end;

échec

```



On remarquera que le fait de faire retourner la valeur **false** à la fonction fera échouer l'opération qui l'a déclenchée. Dans notre exemple c'est l'opération **update** qui ne sera finalement pas effective.

Dernier exemple sur les triggers déclenchés par un **update** : dans un cabinet de pédiatre, une application permet de garder trace de la taille actuelle de chacun des bébés suivis :

bébé		
idBB	nom	taille
7	Pitou	67
5	Pouillou	53
...

On peut imaginer qu'un trigger garantisse que l'attribut *taille* de la relation ne contiennent que des valeurs croissantes, c'est-à-dire que la modification de la taille d'un bébé le fasse uniquement grandir. Pour cela on écrira une procédure qui comme les précédentes devra renvoyer le type trigger et dont nous donnons uniquement le corps ici :

```

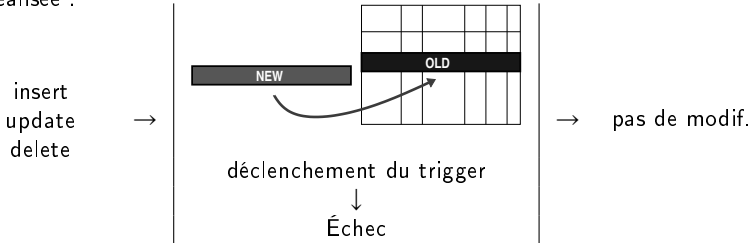
begin
  -- si une saisie diminue la taille du bebe
  if NEW.taille < OLD.taille then
    return false;
  end if;
  return NEW;
end;

```

Le rattachement de cette routine à la relation *bebe* est laissé en exercice au lecteur.

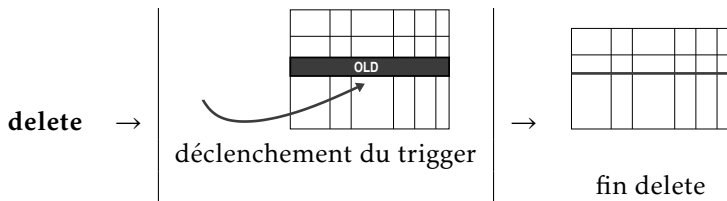


On notera finalement que les triggers, quelle que soit l'opération qui les déclenche, participent à l'intégrité des données. En effet, comme le montre l'exemple précédent, les routines auxquelles ils font appel peuvent être utilisées pour vérifier la cohérence des informations stockées. Dans ce cas, l'opération déclenchant le trigger ne sera pas réalisée :



6.5.3 Trigger lors d'un **delete**

Dans le cas d'un trigger se déclenchant sur un **delete**, on a accès pendant son exécution, au tuple OLD *avant* son effacement effectif :



Une application possible des triggers sur la requête **delete** est l'archivage avant suppression effective. En d'autres termes on met en place un dispositif pour mettre de côté les tuples que les requêtes **delete** effacent. On crée d'abord la table d'archivage :

```
create table arch_musicien(
    nom text,
    prenom text)
```

On écrit ensuite la procédure d'archivage :

```
create function archiver_musicien()
returns trigger as $$
begin
    insert into arc_musicien(nom, prenom)
        values (OLD.nom, OLD.prenom);
    return OLD;
end;
$$ language plpgsql;
```

Puis le déclencheur lui-même :

```
create trigger trig_archivage_musicien  
before delete on musicien for each row  
execute archiver_musicien()
```

Une fois le trigger posé sur la table `musicien`, une opération telle que :

```
delete from musicien where origine='US'
```

effacera effectivement tous les musiciens d'origine américaine et archivera leur nom et prénom dans la table `arch_musicien`. Il faut noter que la fonction `archiver_musicien` sera appelée *autant de fois qu'il y a de tuples concernés*.

6.5.4 Triggers niveau table

Nous avons vu que les triggers niveau tuple déclenchaient l'exécution d'une routine *pour chaque tuple concerné* par la requête associée. Ainsi si une opération **update** concerne 250 tuples, alors le trigger déclenchera 250 appels à la routine pour chacun des 250 tuples. Il existe cependant des triggers « niveau table » (*table level trigger*) qui s'exécuteront pour l'ensemble de la table, sans distinction de tuples. Ils sont déclenchés par le même type d'événement que les triggers niveau tuple (**update**, **insert** ou **delete**) mais uniquement une fois par requête.

Limiter le nombre de pistes

Nous vous proposons ici un exemple d'utilisation d'un trigger niveau table qui va consister à vérifier que le nombre de pistes d'un album ne dépasse pas un certain seuil (disons 50, juste pour l'exemple). Soit le modèle suivant :

Album	
idAlb	Titre
2	Jazz From Hell
17	One Size Fits All

comp_album		
idAlb	idPist	num
...
2	10	4
17	11	1
...

Piste	
idPist	Titre
...	...
10	Saint-Étienne
11	Inca Roads
...	...

Pour s'assurer que le nombre de pistes d'un album ne dépasse pas une valeur¹² on devra écrire le trigger :

```
create trigger trig_check_max_piste
after insert or update
on comp_album
execute procedure check_max_piste();
```

La procédure `check_max_piste()` quant à elle devra retourner un trigger comme toutes les procédures déclenchées par des triggers que nous avons créées jusqu'ici :

```
create or replace function check_max_piste()
returns trigger as $$
begin
    -- verifier qu'il n'y a pas
    -- plus de 50 pistes par album
end; $$ language plpgsql;
```

Pour réaliser la vérification laissée dans le code ci-dessus en commentaire, on aura d'abord recours à une agrégation :

```
select count(idpist), idalb from comp_album
group by idalb
having count(idpist)>50
```

qui renverra pour chaque identifiant d'album le nombre de pistes qu'il contient. On pourra ensuite compter le nombre de tuples que renvoie cette agrégation et stocker ce résultat dans une variable¹³ :

```
select count(*) into err
from (
    select count(idpist), idalb from comp_album
```

12. Je vous laisse imaginer qu'on peut adapter cet exemple à d'autres situations savoureuses comme : nombre d'inscrits à une réunion, ou nombre d'articles d'une facture, ou ...

13. Qu'il faudra déclarer de type `integer` dans une clause `declare`.


```

        group by idalb
        having count(idpist)>50
    ) check_comp_album;

```

Il restera à ajouter un test dans la procédure pour vérifier que la variable err contient bien la valeur 0. Dans le cas contraire on lèvera une ►exception :

§ 6.4.8 ◀
p. 228

```

if err>0 then
    raise exception using
        message='Nombre_max_de_pistes_atteint',
        errcode='NPMAX';
end if;

```

Que les numéros de pistes se suivent

Pour s'assurer qu'une suite d'entiers $1, 2, 3, 4, \dots, N$ se suivent on peut utiliser le fait que la somme des N premiers entiers est :

$$\sum_{i=1}^N i = \frac{N \times (N + 1)}{2}$$

Imaginons que l'on veuille mettre en place un scénario permettant d'assurer que les numéros de pistes se suivent. Ce scénario devrait vérifier qu'après une modification sur la relation comp_album la somme des N numéros est égale à $\frac{N \times (N + 1)}{2}$. Voici une étape pour y arriver :

```

select idalb,
       count(idpist)           as nbpistes,
       sum(num)                as somme,
       count(num)*(count(num)+1)/2 as verif
from comp_album group by idalb

```

Cette requête (que l'on peut stocker dans une vue) renvoie pour chaque album, outre l'identifiant :

- le nombre de pistes, le N de notre formule;
- la somme des numéros de pistes, le \sum de notre formule;
- la somme des N premiers entiers $\frac{N \times (N + 1)}{2}$.

En imaginant que cette requête soit stockée dans une vue (appelons là v_num_piste), on pourra dans un trigger intercepter les cas où les numéros de pistes ne se suivent pas, comme ceci :

```

select count(*) into errnumpiste
  from v_num_piste
 where not somme=verif;
if errnumpiste>0 then
  raise exception using
    message='Numeros_de_piste_non_contigus',
    errcode='NPSUI';
end if;

```

Le code ci-dessus peut donc faire l'objet d'une fonction qui sera appelée par un trigger niveau table sur la relation comp_album.



L'idée de ce trigger niveau table est intéressante en elle-même puisqu'elle illustre une fois encore que l'intégrité des données peut être garantie grâce à ces objets (les triggers). On notera cependant ici que maintenir les numéros de pistes contigus via un trigger n'est pas si simple. En effet, si celui-ci est déclenché après les trois opérations LMD (**update**, **delete** et **insert**) alors ces requêtes ne seront autorisées à effectuer que des opérations très limitées : on ne pourra par exemple pas supprimer une piste qui ne serait pas la dernière, d'autres opérations rendant les numéros de pistes non contigus seront également interdites. Ceci peut être très contraignant au niveau applicatif.

6.5.5 Dernières remarques sur les triggers

6

Tout d'abord un trigger peut être momentanément désactivé. Il suffit pour cela de le nommer :

```
alter table enseignant disable trigger trig_prof
```

D'autre part, une erreur fréquemment commise est d'écrire le trigger suivant sur une hypothétique table bidule :

```

create trigger non_non_non
before update on bidule for each row
execute procedure oulala();

```

Avec dans le corps de la fonction oulala :

```
update bidule set ...
```

Cet appel à **update** sur la table bidule provoque l'appel au trigger, qui provoque l'appel au **update**, qui provoque l'appel au trigger, etc. Le SGBD arrêtera de lui-même ces appels récursifs au trigger avec un message d'erreur :

```
ERROR:  stack depth limit exceeded
```

stack désignant la pile, zone mémoire dans laquelle sont stockées momentanément les données avant l'appel d'une fonction...

Enfin, dernière remarque, sachez qu'il est possible d'attacher plusieurs triggers à la même table. L'ordre dans lequel ils seront exécutés va dépendre du SGBD utilisé. PostgreSQL traite le problème des triggers multiples attachés à une table, en l'exécutant selon leur ordre alphabétique.

6.5.6 Ça passait, c'était beau (le retour)



Au paragraphe 3.4 page 80, nous avons contruit un MCD destiné à modéliser la gestion des primes affectées au personnel d'une entreprise (constituée de Schtroumpfs et de Barbapas, mais une entreprise tout de même). Nous avons ensuite implémenté ce modèle sous forme de relations au chapitre 4 à la section 4.7.4 page 132. Entre autres choses, nous étions arrivés à la conclusion que pour assurer une intégrité parfaite des données, il aurait été nécessaire d'ajouter des informations redondantes dans la table `touche`

Touche		Affectable	
idE	idP	idC	idTP
1	2	S	SP
...	...	B	FA

Car ici rien ne garantit que l'employé 1 puisse toucher la prime 2. De manière à garantir la cohérence de nos données, nous allons créer un trigger qui se mettra en échec si l'intégrité référentielle est violée. Avec la table `affectable` ci-dessus et les suivantes :

Employé			Prime			
idE	idC	nom	idP	idTP	libelle	montant
1	S	Farceur	1	FA	prime 1	10
2	B	Barbouille	2	SP	prime 2	30
3	B	Barbidur	3	SP	prime 3	15

on souhaiterait que la requête :

```
insert into touche(idE,idP) values(1,3)
```

se déroule normalement (car l'employé 1 est un schtroumpf, que la prime 3 est de la salsepareille, et que les schtroumpfs ont le droit d'en toucher). Mais que la requête :

```
insert into touche(idE,idP) values(2,3)
```

échoue car l'individu 2 est un barbabapa et que ceux-ci ne sont pas autorisés à toucher de primes de type salspareille.

Vous en avez rêvé, SQL l'a fait : grâce à un trigger au moment l'insert ou de la mise à jour. En deux temps, on crée d'abord une fonction :

```
create or replace function touche_prime()  
returns trigger as $$  
declare  
  newidC   char;  
  newidTP  char(2);  
  aff      int;  
begin  
  -- récupérer la catégorie de l'employé  
  select idC into newidC from employe  
    where idE=NEW.idE;  
  -- récupérer le type de la prime  
  select idTP into newidTP from prime  
    where idP=NEW.idP;  
  -- prime affectable ?  
  select 1 into strict aff from affectable  
    where idc=newidC and idtp=newidTP;  
exception  
  when no_data_found  
    raise exception 'Pb_Categorie_/_type_prime'  
end; $$ language plpgsql;
```

Puis dire gentiment¹⁴ au SGBD que cette fonction doit être exécutée avant le insert ou le update pour chaque tuple :

```
create trigger trig_touche_prime  
before insert or update on touche  
for each row execute procedure touche_prime();
```

Deux petites remarques concernant la fonction mise en œuvre :

1. notons l'utilisation du mot clé **strict** sans lequel une absence de valeur ne lèvera pas l'exception **no_data_found**;

14. Reportez-vous au paragraphe 6.2.2 page 215 pour voir comment on peut stocker les instructions qui suivent dans un fichier et exécuter ce fichier depuis la console **psql**.

2. la manière dont on sort en mode échec du trigger : il suffit de lever une exception.

Ce trigger nous permet donc de contrôler la validité de l'affectation de la prime en allant chercher dans la relation affectable la présence du couple (catégorie, type de prime). Si le couple n'existe pas (un utilisateur dont la catégorie n'est pas compatible avec le type de la prime), le `select` ne renverra aucune valeur, l'exception `no_data_found` sera donc levée, on aura donc le message suivant :

```
yahozna=> insert into touche(idE,idP) values(2,3);  
ERROR: Pb Catégorie / type prime
```

On voit ici le trigger en action, pour affecter la prime 3 à l'employé 2 :

- l'employé 2 est de la catégorie B ;
- la prime 3 est de type SP ;
- le couple (B,SP) n'est pas dans la table affectable

En d'autres termes, il n'est pas prévu de nourrir les barbabapas avec de la salsepareille. C'est donc une tentative de violation de contrainte d'intégrité référentielle, on annule donc l'insertion (ou la mise à jour).

6.6 Utiliser un autre langage



PostgreSQL et consorts offrent la possibilité d'écrire des routines dans un autre langage que SQL. Pour ce qui est de PostgreSQL, vous pourrez créer vos propres routines grâce aux langages suivants : le langage C, Perl, Python et quelques autres.

Nous vous proposons ici un exemple utilisant conjointement Python et le mécanisme des triggers. L'idée est simple : imaginons une application permettant à un utilisateur de stocker des *documents* sous forme de *fichiers stockés en dehors de la base*. On ne stockerait dans la base que la référence (nom) du fichier et les méta-données (par exemple la description du document) qui lui sont associées. Cette application permettrait interactivement de :

- déposer un document ;
- effacer un document ;
- renommer un document.

Maintenant que vous êtes des spécialistes du modèle relationnel, vous pressentez donc le fait que chacune des ces trois opérations va

respectivement donner lieu à un **insert**, **delete** et **update** dans une table qui pourrait être structurée comme ceci :

doc		
id	reference	description
1	bidule.png	mes vacances
2	truc.flac	ma chanson
14	machin.tex	mon texte

Nous allons stocker la référence sur le fichier et la description des documents dans la relation construite comme ceci :

```
create table doc(id serial,
                 reference text,
                 description text)
```

Nous construisons ensuite deux triggers qui se déclencheront automatiquement :

- l’effacement du fichier si un **delete** survient sur la table ;
- le changement de nom du fichier si un **update** survient.

On fera l’hypothèse que les fichiers déposés le sont dans un répertoire fixe et n’en bougent pas.

6

6.6.1 D’abord créer le langage

Pouvoir accéder à Python depuis une procédure de la base de données suppose d’abord que l’extension soit disponible sur votre système. Pour les Linux basés sur les paquets Debian (Ubuntu, Mint entre autres) ce paquet se nomme : `postgresql-plpython3-X.Y` où `X.Y` correspond à la version de PostgreSQL.

Enfin, *en tant qu’administrateur*, c’est-à-dire en tant qu’utilisateur `postgres`, il faudra installer le langage dans la base de données en lançant la commande :

```
create language plpython3u
```

Vous pouvez également vous persuader que le langage est installé en examinant la table `idoin`¹⁵ dans le catalogue :

```
select * from pg_language
```

qui devrait faire apparaître le contenu d’une relation avec la liste des langages installés, dont Python3.

15. En 2017, un étudiant de 1^{er} cycle de l’enseignement supérieur, tape alors méticuleusement : `select * from idoine`.

6.6.2 Modifier un fichier

Nous allons dans un premier temps créer la fonction qui sera exécutée lors de l'opération d'**update**:¹⁶

```
create or replace function rename_doc()  
    returns trigger  
  
as  
$$  
    # message dans la console  
    plpy.notice('coucou')  
$$  
language plpython3u;
```

NB : n'oubliez pas que cette routine doit être créée par l'utilisateur postgres. On pourra noter que pour l'instant cette routine ne fait rien d'autre que de vous faire un petit coucou lorsqu'elle est appelée par le trigger. Trigger que vous devrez définir avec le code ci-dessous :

```
-- effacer l'existant le cas echeant  
drop trigger if exists trig_upd_doc on doc ;  
-- associer le trigger et la table  
create trigger trig_upd_doc  
before update on doc for each row  
execute procedure rename_doc();
```

La phase suivante est d'essayer de déplacer le fichier pendant l'opération d'**update**. Pour y arriver, il faut savoir que pendant l'exécution de la fonction trigger, une variable (de type tableau associatif) nommée TD contient plusieurs champs dont :

- new : le tuple après la modification
- old : le tuple avant la modification

Le rôle de ces deux tuples est illustré au § 6.5.2 page 243. Nous pouvons par exemple modifier le corps de la fonction Python comme ceci :¹⁷

```
plpy.notice(TD["old"][ "reference" ]  
            +" => "  
            +TD["new"][ "reference" ])
```

16. Référez-vous au § 6.2.2 pour voir comment intégrer cette fonction dans votre base.

17. J'indique ici uniquement le code Python, c'est-à-dire tout ce qui se trouve entre les \$\$.

Après avoir chargé la nouvelle version de cette fonction dans la base, une commande telle que :

```
update doc
      set reference='bidule.tex'
      where id=14
```

produira dans la console SQL le message :

```
NOTICE: machin.tex => bidule.tex
```

L'expression Python `TD["old"]["reference"]` renvoie la valeur de l'attribut `reference` du tuple `OLD`. Et de manière identique, l'expression `TD["new"]["reference"]` renvoie la valeur de ce même attribut pour le tuple `NEW`.

Nous n'avons donc pour l'instant qu'un message d'information affiché dans la console, il reste donc à demander, en Python, le changement de nom du fichier dont on connaît les deux références, avant et après modification. Le code Python permettant de renommer un fichier est le suivant :

```
import os
os.rename(⟨ancienne ref.⟩ , ⟨nouvelle ref.⟩)
```

Adapté à notre problème — je vous rappelle que nous devons renommer un fichier à la suite d'un **update** — ce code deviendra :

```
plpy.notice(...) # le petit msg precedent
os.rename('/usr/local/data/'
          +TD["old"]["reference"],
          '/usr/local/data/'
          +TD["new"]["reference"])
```

Une fois mise en base de données, cette fonction permettra de synchroniser la valeur de l'attribut `reference` et la référence du fichier lui-même. Vous aurez compris que `/usr/local/data` est supposé être le dossier où se trouvent les fichiers.

6.6.3 Effacer un document

Si vous avez ingurgité et digéré la section précédente, vous ne trouverez sans doute rien à redire à la fonction suivante :

```
create or replace function
del_doc()
returns trigger as
```



```

$$
import os
# message
ply.notice('effacement_',
           +TD["old"][ "reference" ])
# effacement du fichier
os.unlink('/usr/local/data/'
          +TD["old"][ "reference" ])
$$
language plpython3u;

```

qui, après un message laconique dans la console, effacera le fichier auquel fait référence un tuple effacé par un **delete** :

```
delete from doc where id=1
```

Cette requête supprimera le tuple et effacera le fichier dans le dossier /usr/local/data.

6.6.4 Conditions d'utilisation

Outre le fait que les fonctions devront être créées dans la base par l'administrateur (utilisateur logique postgres), quelques précautions sont à prendre pour mettre en œuvre ce qui précède.



Tout d'abord le dossier pris en exemple (/usr/local/data) doit être accessible en écriture et en lecture par l'utilisateur logique faisant tourner le serveur de base de données. Sur un système Unix avec PostgreSQL, il s'agit également de l'utilisateur postgres. Il faudra donc vous assurer que ça soit le cas, à coup de **chown**, **chgrp** ou **setfacl** selon vos besoins.



En outre il faut savoir que l'architecture proposée dans cet exemple suppose que le Sgbd et le programme d'application qui permettra à l'utilisateur de déposer un fichier soient tous deux en mesure d'accéder au répertoire de dépôt. Ça ne sera pas forcément votre cas.



Enfin, il faut savoir que laisser les droits à un script d'accéder aux services du système d'exploitation (accès au fichier, aux processus, possibilité d'exécuter des programmes externes, etc.) est considéré comme une faille potentielle de sécurité qui peut être exploitée. C'est la raison pour laquelle le langage procédural utilisé est affublé de la lettre u (**plpython3u**) pour **untrusted**. Vous êtes avertis;-)

6.7 Accès concurrents

Nous allons maintenant aborder la situation où deux (ou plus) processus veulent modifier les données en même temps. Cette section a pour objectif d'introduire les mécanismes intégrés aux SGBD entrant en œuvre dans ce cas précis. On comprend intuitivement que l'accès simultané à une information en lecture ne pose pas de problème particulier mais que la modification ou l'effacement d'une même donnée par plusieurs processus risque d'en poser.

Nous vous proposons dans un premier temps de décortiquer les quatre concepts incontournables pour comprendre comment les systèmes de gestion de base de données font face aux accès concurrents :

1. les *contraintes d'intégrité*, déjà abordées en profondeur dans le chapitre traitant de la construction des données ;
2. la notion d'*atomicité*, un peu abordée dans le chapitre traitant de la manipulation des données ;
3. l'*isolation* des données ;
4. le *verrouillage* de données.

Enfin, nous vous montrerons que lorsque ces 4 mécanismes se mettent en branle conjointement, il est possible de gérer correctement la situation classique de la réservation mutuelle de ressources : « *il reste 3 places pour le concert de Justine Bibeure¹⁸ et quelques milliers de personnes cliquent frénétiquement sur leur souris dans le doux espoir d'en obtenir une...* »

Avant d'exposer ces quatre concepts indispensables, nous vous proposons en guise de préambule, d'une part d'ouvrir une petite discussion autour des exceptions et d'autre part, d'examiner le comportement des séquences ◀ dans le contexte des accès concurrents.

► § 4.5.2
p. 113

6.7.1 Exception et accès concurrents

À la section 6.4.8 page 228 nous vous présentions une procédure permettant, à partir d'un triplet (inom, iprenom, iemail) pour le nom, le prénom et l'email, d'insérer un nouvel individu ou de le mettre à jour dans le cas où un individu existerait déjà avec cet email. Sans la connaissance du mécanisme des exceptions, un utilisateur naïf pourrait écrire une procédure dont le squelette serait :

18. Célèbre chanteuse du début du XXI^e siècle.

```

declare
  existe int;
begin
  select count(*) into existe
    from personne where email=iemail;
  if existe>1 then
    update ...
  else
    insert ...
  end if;
end;

```

En imaginant le pire dans le contexte d'une utilisation par plusieurs utilisateurs de manière simultanée, on peut imaginer que le séquençement dans le temps soit :

1. utilisateur 1 : la personne existe-t-elle? non.
2. utilisateur 2 : la personne existe-t-elle? non.
3. utilisateur 1 : donc j'insère.
4. utilisateur 2 : donc j'insère. \Rightarrow **erreur (violation unicité)**.

Pour rappel, le squelette de cette procédure avec l'utilisation des exceptions était :

```

begin
  insert into personne ...
exception
  when unique_violation then
    update personne ...
end;

```

Une telle construction gère correctement le cas où plusieurs processus exécutent en même temps ce bloc d'instructions.

6.7.2 Séquences et accès concurrents

Les ►séquences, nous l'avons vu, sont des objets permettant de § 4.5.2 ◀
générer les entiers pour les clés de substitution. On y accède via p. 113
trois méthodes ou fonctions :

1. **setval** pour initialiser à une valeur donnée;
2. **nextval** pour incrémenter la séquence;
3. **currval** pour demander la valeur courante.

Pour apprécier le fonctionnement des séquences, imaginons d'abord ce que nous aurions dû faire si nous ne les connaissons pas. Pour ajouter un nouveau musicien dans la relation *musicien* :

Musicien		
<i>idMus</i>	prenom	nom
1	Zappa	Frank
2	Van Vliet	Don

il faudrait :

1. d'abord trouver la valeur max \mathcal{M}
2. insérer le nouveau musicien avec $\mathcal{M} + 1$ comme clé

Si ces deux instructions sont exécutées *de manière isolée*, tout ira bien :

1. appli cherche le max et trouve $\mathcal{M} = 2$
2. appli insère nouveau tuple avec $\mathcal{M} + 1 = 3$

En revanche, si on imagine que deux applications demandent l'exécution de ces deux instructions de manière concurrente, dans le pire des cas, le séquençement dans le temps pourrait être par exemple :

1. appli1 cherche le max et trouve $\mathcal{M} = 2$
2. appli2 cherche le max et trouve $\mathcal{M} = 2$
3. appli1 insère nouveau tuple avec $\mathcal{M} + 1 = 3$
4. appli2 insère nouveau tuple avec $\mathcal{M} + 1 = 3$ (!)

L'instruction 4 génère bien entendu une violation d'unicité sur la clé primaire. Avec une séquence, les deux applications appli1 et appli2 vont exécuter l'instruction **nextval**. Vous pourriez le tester vous même en lançant deux consoles et en exécutant la séquence :

D'abord :

```
bd=> select nextval('seq_mus');
      nextval
-----
          3
(1 row)
```

Puis :

```
bd=> select nextval('seq_mus');
      nextval
-----
          4
(1 row)
```

Imaginons maintenant que nous souhaitions, comme dans la procédure `ajouter_saxophoniste` (cf. § 6.4.3 page 220), insérer une entrée dans une table avec l'identifiant du musicien nouvellement créé. Cet identifiant peut être obtenu grâce à la fonction `currval`. Si chacune des sessions ci-dessus faisaient appel à cette fonction on aurait :

```
bd=> select currval('seq_mus');
currval
-----
      3
(1 row)
```

```
bd=> select currval('seq_mus');
currval
-----
      4
(1 row)
```



Il est donc particulièrement intéressant de remarquer que les séquences sont à la fois des objets « globaux » puisque que chaque **nextval** semble incrémenter une zone mémoire partagée, mais également des objets « cloisonnés » puisque **currval** renvoie à chaque « session » sa propre valeur courante (dans l'exemple ci-dessus, la session de gauche, renvoie la valeur 3 et non 4).

6.7.3 Transactions : retour sur l'atomicité

§ 5.2.4 ◀
p. 176

Nous avons introduit le concept d'atomicité dans le chapitre traitant de la manipulation des données. Nous rappelons donc ici que l'idée de l'atomicité des requêtes — celles qui *modifient* les données — peut se résumer en une phrase :

*Soit **tous les tuples** impliqués par la requête seront modifiés*
*Soit **aucun***

L'exemple donné à la section à laquelle on fait référence ci-dessus concernait la requête suivante :

```
delete from pays where ...
```

Cette requête :

- *effacera tous les tuples de la relation pays* si aucune contrainte d'intégrité référentielle n'est violée, en d'autres termes si aucun tuple d'une relation n'y fait référence ;
- *n'effacera aucun tuple de la relation pays* si la suppression d'un des tuples viole une contrainte d'intégrité ;
- **en aucun cas**, le SGBD n'effacera qu'une partie des tuples concernés par la clause **where**.

Maintenant que nous avons en tête le fait que les requêtes sont atomiques, nous allons voir qu'il est possible de *rendre l'exécution de plusieurs requêtes* atomique. En écrivant :

```
begin
  requete_1
  requete_2
  ...
  requete_N
commit
```

Grâce au bloc **begin/commit** on indique au SGBD que l'on souhaite que *toutes les requêtes* (1 à N) doivent être exécutées ou *aucune*.

Dans le cas où une erreur — quelle qu'elle soit — survient le SGBD remettra la base dans l'état où elle était au moment du début de la transaction. C'est-à-dire au moment de l'instruction **begin**. Ce mécanisme de « retour au début » s'appelle ici un *rollback*. On « ré-enroule » ou on « rembobine » les requêtes que l'on a exécutées jusqu'ici. Dans l'exemple présenté ci-dessous, le *rollback* est *implícite*, c'est-à-dire que ça n'est pas l'utilisateur/programmeur qui le déclenche mais le SGBD.

```
-- debut transaction
begin
  requete_1
  requete_2
  ...
  requete erreur -- erreur ici !!
  ...
  requete_N
commit
```

On revient « au début »

Pour comprendre ce qu'est un **rollback** explicite, on peut imaginer que vous tapotiez les commandes suivantes dans votre console SQL :

```
yahozna=> begin ;
yahozna=> delete from musicien ;
yahozna=> delete from pays ;
```

À ce stade, si aucune contrainte ne l'interdit, tous les tuples de la table *musicien* ainsi que ceux de la relation *pays* sont effacés. Si maintenant vous tapez :

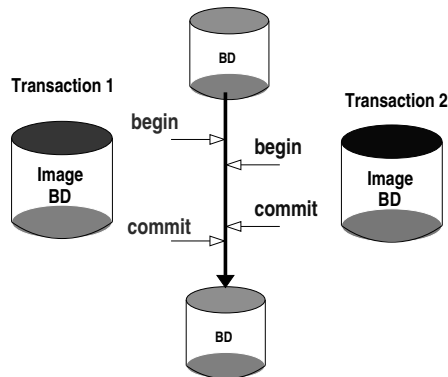


FIGURE 6.2 – Isolation des données : pendant l'exécution des transactions, chacune d'elles voit sa propre version de la base !

```
yahozna=> rollback;
```

Vous « annulez » en quelque sorte, les opérations **delete** que vous venez d'effectuer. L'état actuel de la base sera l'état dans lequel elle était au moment de votre **begin**¹⁹.

6

6.7.4 Isolation

L'*isolation* des données est un mécanisme dont l'effet le plus frappant est le suivant : si deux transactions commencent et qu'elles modifient chacune des données de la base, elles verront chacune leur propre version de celle-ci, pendant la durée des transactions (voir figure 6.2 dans laquelle le temps défile vers le bas). Pour vous en convaincre, vous pouvez simuler deux accès concurrents en ouvrant deux consoles :

```
bd=> begin;
```

```
bd=> begin;
```

Les deux transactions ont commencé...

19. Que je vous souhaite de ne pas avoir oublié de saisir...

Une requête de modification :

```
bd=> update musicien set
bd-> nom='Zoppo' where idmus=3 ;
```

On jette un œil de chaque côté :

```
bd=> select nom from musicien
bd-> where idmus=3 ;
    nom
-----
Zoppo
(1 row)
```

```
bd=> select nom from musicien
bd-> where idmus=3 ;
    nom
-----
Zappa
(1 row)
```

À ce stade, on constate donc que chaque transaction voit sa propre version de la base : la transaction de gauche voit la modification, celle de droite non. En d'autres termes la transaction de droite ne voit pas les modifications apportées par la transaction *non terminée* de gauche.

6

On continue la mission, en terminant la transaction de gauche :

```
bd=> commit ;
```

```
bd=> select nom from musicien
bd-> where idmus=3 ;
    nom
-----
Zoppo
(1 row)
```

On constate ici que la transaction de droite peut voir les modifications de la transaction (terminée) de gauche...



On retiendra qu'à ce niveau d'isolation — qui est celui par défaut de PostgreSQL — nommé *read committed*, une transaction voit les modifications apportées par une transaction terminée. On peut cependant si nécessaire, basculer une transaction dans un niveau d'isolation plus strict, dans lequel une transaction ne pourra pas voir les modifications d'une transaction non terminée.

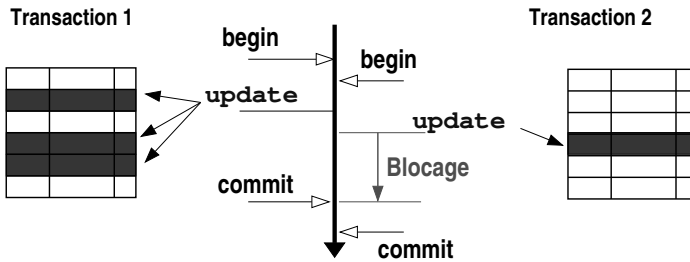


FIGURE 6.3 – Verrouillage d'un tuple : la transaction 2 qui tente un **update** après la transaction 1, est bloquée jusqu'à ce que celle-ci se termine (ici avec un **commit**).

6.7.5 Verrouillage

Voici maintenant un autre mécanisme entrant en jeu dans la gestion des accès concurrents : le *verrouillage*. Il existe plusieurs situations dans lesquelles le SGBD va garder un accès exclusif à un tuple lorsqu'il est en train de le modifier. Nous vous proposons ici un exemple illustrant le verrouillage d'un tuple lors d'une opération **update**. L'idée générale est illustrée à la figure 6.3 : lorsqu'une transaction commence une opération de mise à jour sur un tuple, celui-ci est verrouillé pendant la durée de la transaction. Une autre transaction qui tenterait de modifier ce même tuple serait bloquée pendant toute la durée de la transaction « modificatrice ».

Voyons maintenant l'exemple. Comme pour l'atomicité, commençons deux transactions :

```
bd=> begin;
```

```
bd=> begin;
```

Une requête de modification par la transaction de gauche :

```
bd=> update musicien set
bd-> nom='Zippo' where idmus=3 ;
bd=> UPDATE 1
```

Puis une autre requête de modification sur le même tuple par la transaction de droite :

```
bd=> update musicien set
bd-> nom='Zuppu' where idmus=3 ;
```

Ce que tente de montrer cet exemple, c’est que lorsque la transaction de gauche modifie le tuple identifié par **idmus=3**, alors celle de droite *reste bloquée*. Puis :

Fin de la transaction :

commit

qui entraîne un déblocage :

UPDATE 1



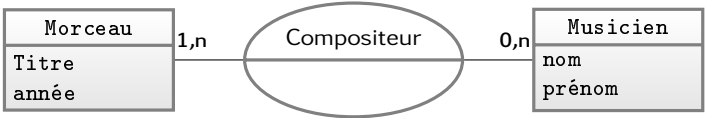
On peut donc retenir que les tuples sont verrouillés par les requêtes de modification (**update** ou **delete**), entraînant l'attente de toutes les autres transactions qui tenteraient d'accéder aux tuples en cours de modification. Cette attente, ou blocage, est levée une fois que la transaction qui modifie le tuple se termine par un **commit** ou un **rollback**.

6.7.6 Contraintes déférables

Avant d’attaquer une étude de cas faisant intervenir les différents mécanismes des accès concurrents, nous exposons dans ce paragraphe une fonctionnalité associée aux contraintes d’intégrité permettant de les désactiver au début d’une transaction et de *reporter* leur mise en route à la fin de celle-ci. Les anglophones disent :

- *defer a constraint* : reporter une contrainte
- *deferrable constraint* : contrainte reportable

Le mécanisme est une chose, voyons maintenant à quoi il peut bien servir. Pour cela revenons sur le MCD partiel ci-dessous :



Il sera implémenté en relationnel comme ceci :

Morceau			compose		Musicien		
idMorc	titre	...	idMorc	idMus	idMus	nom	...
10	10	1	1	Zappa	...
20	10	2	2	Varèse	...
			20	1			

Comme vu précédemment, il y aura une clé primaire composite sur la table `compose` :

```
alter table compose add constraint pk_compose
primary key(idMus,idMorc)
```

Imaginons qu'il existe une relation faisant référence à compose :

T			
...	idMus	idMorc	...

Ce lien sera matérialisé par une contrainte d'intégrité de clé étrangère :

```
alter table T add constraint fk_T_compose
foreign key(idMus,idMorc)
references compose(idMus,idMorc)
```

Supposons maintenant qu'au niveau applicatif, existe un utilitaire permettant de gérer les compositeurs d'un morceau, et que cet utilitaire présente à l'utilisateur une liste de musiciens à cocher pour chaque morceau :

G-Spot Tornado

- ☐ Frank Zappa
- ☐ Terry Bozzio
- ☐ Don Van Vliet

Annuler
OK

Au moment de la pression sur la touche OK, l'application enverra au Sgbd :

- l'identifiant du morceau ;
- la liste des musiciens cochés.

Et le traitement consistera en les deux étapes suivantes :

1. effacer les « anciens musiciens » compositeurs (ceux initialement cochés), le cas échéant ;
2. ajouter les « nouveaux compositeurs » (ceux qui viennent d'être cochés).

La première étape sera réalisée grâce à un habile :

```
delete from compose where idMorc=17
```

Puis pour chaque musicien *m* coché on exécutera furtivement un :

```
insert into compose(idMorc,idMus) values(17,m)
```



Si vous avez tout suivi jusqu'ici, vous aurez nécessairement réagi au moment du **delete** ci-dessus, puisqu'en effet la table T fait référence à la table `compose` dans laquelle on efface. La clé étrangère `fk_T_compose` tirera donc la sonnette d'alarme en faisant échouer le **delete** de l'étape 1 s'il existe des tuples de la relation T faisant référence à `compose`. Mais dans la mesure où le **insert** de l'étape 2 devrait tout remettre dans l'ordre, il serait légitime de souhaiter que cette contrainte soit momentanément levée. C'est exactement ce que proposent les contraintes reportables.

De manière à utiliser le mécanisme des contraintes reportables, on spécifiera *au moment de la création de la contrainte*, que celle-ci peut être reportée à la fin d'une transaction, grâce au mot clé **deferrable** :

```
alter table T add constraint fk_T_compose
foreign key(idMus,idMorc)
references compose(idMus,idMorc)
deferrable
```

Puis, dans le bloc de transaction — qui sera très certainement une procédure stockée — on écrira :

```
begin
-- report de la contrainte
set constraints fk_T_compose deferred
-- effacement des tuples
delete from compose where idMorc=17
-- insertion des nouveaux
for ... loop
insert into compose(idMorc,idMus)
values(17,...)
end loop
end
```

La première instruction de ce bloc permet donc d'indiquer au SGBD :

- qu'il peut lever la contrainte dont le nom est stipulé pendant toute la transaction
- qu'il devra la remettre en route à la fin de la transaction



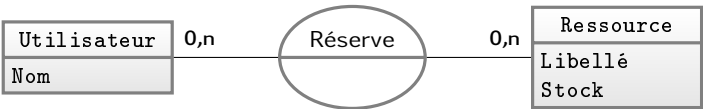
Bien évidemment à la fin de la transaction, lorsque la contrainte reportée sera de nouveau activée, le Sgbd vérifiera que les données respectent les conditions de celle-ci et interrompra son traitement si ça n'est pas le cas, comme il le fait habituellement.

6.7.7 Étude de cas : réservation de ressources

Voyons maintenant comment les différents mécanismes exposés jusqu’ici — *contrainte d’intégrité, atomicité, verrouillage* et *isolation* — en agissant conjointement, peuvent résoudre un problème générique : la réservation de ressources. Il s’agit des situations assez courantes où plusieurs utilisateurs se jettent désespérément sur les dernières places de concert de leur idole au Stade de France, ou sur les derniers exemplaires de leur smartphone préféré en vente sur le site <http://www.entube.com>, ou ... bref, vous voyez « c’que j’veux dire ».

! À partir d’un modèle conceptuel simple, nous vous proposons maintenant, de constater comment le Sgbd va répondre à ce problème de réservation en vous assurant que la dernière « ressource » (la place de concert, l’article de rêve, le billet à prix incroyable pour les Seychelles, etc.) soit attribuée à un seul utilisateur.

Voici le MCD proposé :



Donc pour faire court : on dispose d’un ensemble d’utilisateurs identifiés par un attribut de type clé de substitution, des ressources identifiées de manière identique et une association permettant de mémoriser qu’une ressource a été réservée par un utilisateur. Le modèle relationnel est donc le suivant :

réserve	
idU	idR

utilisateur		
idU	nom	
10	α	
20	β	

ressource		
idR	libelle	stock
1	Tarte tatin	22
2	Tarte au citron	1
3	Tarte aux fraises	3

On supposera dans ce modèle que l’ajout d’une entrée dans la relation *réserve* signifiera la réservation d’un exemplaire de la ressource. Il y a bien sûr d’autres moyens plus adaptés pour gérer la

réserve de plusieurs exemplaires d’une ressource, mais cela n’aurait que peu d’impact sur notre propos : *les accès concurrents à une même ressource*. À titre d’exemple si l’utilisateur nommé α réserve une tarte tatin, on devra ajouter dans la table réserve le tuple :

réserve	
idU	idR
10	1

Il faudra également veiller à noter qu’il y a une tarte en moins dans notre stock :

ressource		
idR	libelle	stock
1	Tarte tatin	21
...

Si bien que pour réserver une ressource \mathcal{R} pour un utilisateur \mathcal{U} , il faudra simplement exécuter les deux requêtes suivantes :

```
insert into reserve(idR,idU) values( $\mathcal{R},\mathcal{U}$ )
```

puis décrémenter le stock²⁰ :

```
update ressource set stock=stock-1 where idR= $\mathcal{R}$ 
```

Voyons maintenant ce qui se passe si deux utilisateurs différents se jettent comme des affamés sur la dernière tarte au citron. Imaginons dans un premier temps le scénario suivant :

utilisateur α	utilisateur β
insert dans réserve	insert dans réserve
update de ressource	update de ressource

À la fin de l’exécution de ces requêtes, les données sont les suivantes :

réserve		ressource		
idU	idR	idR	libelle	stock
10	2	1	Tarte tatin	22
20	2	2	Tarte au citron	-1
		3	Tarte aux fraises	3

20. Ce qui pourrait être fait par un trigger.

Ce qui est tout simplement catastrophique :

- deux réservations de tarte au citron;
- stock de tarte à -1 !

Premier réflexe que vous avez déjà eu, je le sais : il faut mettre une contrainte d'intégrité sur le stock pour l'empêcher d'être négatif :

```
alter table ressource
add constraint stock_valide
check (stock >= 0)
```

Ceci ne résout pas entièrement le problème : en effet grâce à cette contrainte, le **update** de l'utilisateur β de notre scénario va échouer, laissant la base dans un état incohérent :

réserve	
idU	idR
10	2
20	2

ressource		
idR	libelle	stock
1	Tarte tatin	22
2	Tarte au citron	0
3	Tarte aux fraises	3

C'est-à-dire :

- stock de tarte à 0 (c'est mieux)
- mais deux réservations de tarte au citron (alors qu'il n'en restait qu'une, je vous le rappelle).

Finalement ²¹, c'est en regroupant les deux requêtes **insert** et **update** dans un bloc de transaction que nous allons résoudre les problèmes. Si nos deux utilisateurs exécutent :

```
begin -- on commence une transaction
insert into reserve(idR,idU) values (R,U)
update ressource
set stock=stock-1 where idR=R
commit -- fin de la transaction
```

Simulons notre scénario de réservation en parallèle de la dernière tarte au citron. À gauche l'utilisateur α (**idU=10**) et à droite β (**idU=20**). Chacun d'eux commence une transaction :

```
bd=> begin;
```

```
bd=> begin;
```

21. Vous me voyez venir avec mes gros sabots...

Puis disons que c'est β qui fait le **insert** le premier, suivi de près par α qui insère lui aussi sa réservation :

```
bd=> insert into reserve(idU,idR)
bd-> values(20,20);
INSERT 0 1
bd=>
```

```
bd=> insert into reserve(idU,idR)
bd-> values(10,20);
INSERT 0 1
bd=>
```

On peut noter qu'à ce stade, compte tenu de l'isolation des données, aucun des utilisateurs ne voit la réservation faite par l'autre. Imaginons maintenant que c'est α qui fait le premier la mise à jour du stock de tarte au citron et que β lui emboîte le pas :

```
bd=> update ressource
bd-> set stock=stock-1
bd-> where idR=2 ;
UPDATE 1
bd=>
```

```
bd=> update ressource
bd-> set stock=stock-1
bd-> where idR=2 ;
```

6



Mécanisme important ici : la transaction de droite est bloquée par celle de gauche qui a effectué un **update** sur le tuple d'identifiant 2 de la relation `ressource`.

```
bd=> commit;
COMMIT
bd=>
```

```
ERROR: new row for relation
"ressource" violates check
constraint "stock_valide"
bd=>
```



Autre mécanisme important ici : au moment où la transaction de gauche se termine avec un **commit**, celle de droite est débloquée, elle tente alors d'exécuter son **update** et se heurte à la contrainte d'intégrité assurant que le stock est positif ou nul.

Finalement, la transaction de droite est nécessairement sans effet puisque le **update** a échoué. Si bien que la tentative de commit :


```
bd=> commit;
ROLLBACK
bd=>
```

se soldera nécessairement pas un **rollback**.



C'est donc le mécanisme d'atomicité qui garantit ici que la base est restaurée dans l'état où elle était au début de la transaction de droite. Ou en d'autres termes, toutes les requêtes de cette transaction sont annulées. Par conséquent, l'insertion de la réservation induite par le **insert** ne sera pas effective.

Si bien que finalement la base est dans l'état suivant :

réserve	
idU	idR
10	2

ressource		
idR	libelle	stock
1	Tarte tatin	22
2	Tarte au citron	0
3	Tarte aux fraises	3

On a donc bien :

- le stock des tartes au citron à 0 !
- et une seule réservation (faite ici par α)

On pourrait objecter qu'il n'est pas rare de pratiquer le surbooking de tartes au citron. Nous répondrons qu'il est laissé en guise d'exercice la modification de la contrainte d'intégrité posée sur l'attribut stock qui permettrait la mise en œuvre de cette pratique douteuse.

6

6.8 Interaction avec un programme : PHP

Cette dernière section du chapitre consacré aux traitements stockés, présente quelques pistes autour des *interactions entre une application et les appels SQL qu'elle contient*. Le langage choisi ici est le langage PHP. Le squelette d'une portion de code contenant une requête SQL est le suivant :

```
// 1. Établissement de la connexion
$conn    = sql_connexion(...);
// 2. Exécution d'une requête
$requete = "select * from bidule";
$res     = sql_query($conn,$requete);
```

```
// 3. Exploitation des tuples si il y en a
while ($T=sql_fetch($res)) {
    // Traitement avec le tuple $T
}
// 4. déconnexion
sql_deconnexion($conn);
```



Notons bien d'une part, que les routines `sql_xxx()` dans le code ci-dessus n'existent pas. Dans une situation réelle, il s'agira de routines préfixées par le Sgbd sous-jacent — par exemple `pg_query()` pour PostgreSQL, `my_query()` pour MySQL, etc.

6.8.1 Requêtes paramétrées

Les applications sont très souvent amenées à exécuter des requêtes qui vont être *construites à partir de données fournies par l'utilisateur*. Une telle situation peut être illustrée par le code suivant dans lequel on imagine que la variable `$nom` contienne une valeur introduite par l'utilisateur de l'application :

```
$nom="..."; // donnée fournie par l'utilisateur
$requete="select id,prenom from individu
        where nom='$nom'";
$res=sql_query($conn,$requete);
// etc.
```

Codée de cette manière, une requête paramétrée par une valeur provenant d'un humain, peut poser deux problèmes :

1. le premier surgira lorsque l'humain aura la mauvaise idée de saisir « d'Alembert ». Dans ce cas la requête à exécuter pourra se lire ainsi :

```
select id,prenom from individu
    where nom = 'd'
    Alembert'
```

Une telle requête déclenchera un vilain message d'erreur pour le malheureux utilisateur de l'application, qui ne comprendra d'ailleurs pas, le plus souvent, ce qu'il a fait de mal.

2. le deuxième cas concernera les humains qui contrairement à ceux qui ne savent pas ce qu'ils ont fait de mal, savent, eux, parfaitement ce qu'ils sont en train de faire. Un tel utilisateur malintentionné pourra, lorsqu'on lui demande de saisir un nom, taper par exemple :

« 'hop'; delete from individu;-- ».

La requête envoyée au SGBD par l'application sera donc :

```
select id,prenom from individu
      where nom='hop';
delete from individu;
-- '
```

Ce qui aurait *un effet désastreux*, même si j'en entends dire « *cela suppose quand même que l'utilisateur malicieux aurait connaissance que ma base contient une table nommée individu et d'autre part, que la connexion établie sur la base soit une connexion autorisant la modification de cette table.* »

Quoi qu'il en soit, **une parade radicale** contre ce type de problème (au passage pour ceux qui l'ignoraient, ce qui est montré au point 2 ci-dessus, se nomme une attaque par **injection de code SQL**), existe *dans tous les langages que vous utiliserez pour dialoguer avec un SGBD*. Ce mécanisme est appelé en anglais *bind variables*. Nous vous fournissons ici gratuitement la version PostgreSQL :

```
$nom="..."; // donnée fournie par l'utilisateur
$requete="select id,prenom
          from individu
          where nom=$1";
$res=pg_query_param($conn,$requete,
                    array($nom));
```

On écrit donc la requête avec des `$x` pour chaque partie variable — sans apostrophe même pour des valeurs de types chaîne de caractères. Puis on passe à la fonction exécutant la requête, un tableau contenant chacune des valeurs. Dans le SGBD Oracle, sans rentrer dans les détails techniques des fonctions PHP, on passera par des paramètres *nommés* :

```
$requete="select id,prenom
          from individu
          where nom=:nom_ind";
```

Puis on liera (*to bind* en anglais), la variable à une valeur :

```
oci_bind_by_name(...,":nom_ind",$nom);
```

6.8.2 Appeler une routine

Il est tout naturellement possible d'appeler une routine depuis votre programme, comme n'importe quelle autre requête SQL :

```
$requete="select
          ajouter_saxophoniste('Washington',
                               'Kamasi');"
$res=sql_requete($conn,$requete);
```

Dans le SGBD Oracle on passera par un bloc « début/fin » :

```
$requete="begin
          ajouter_saxophoniste('Washington',
                               'Kamasi');
end;"
$res=sql_requete($conn,$requete);
```



Le système de requête paramétrée présenté à la section précédente peut bien entendu être appliqué aux appels de routines depuis votre application.

6.8.3 Passer un tableau à une routine

6



Il n'est pas rare qu'une interface de saisie dans une application produise un ensemble de valeurs. Il peut alors être nécessaire de passer cet ensemble à une routine pour le traiter.

```
// valeurs résultant de l'interface de saisie
$ids=array(2,4,7,23);
```

Supposons qu'il existe dans notre base une routine capable de traiter chacune de ces valeurs. Une telle procédure ressemblerait à :

```
create function traite(ids in int[])
returns void
as $$
begin
    -- rien pour l'instant
    null;
end; $$ language plpgsql;
```

Notons ici la syntaxe de déclaration de l'argument de type « tableau d'entiers ». Cette routine peut être appelée grâce à l'ordre SQL suivant (*oui il y a bien des apostrophes autour des accolades*) :

```
select traite('{2,4,6,1,3,5}');
```

On passe ici le tableau constitué des entiers 2,4,...,3,5 à la routine `traite()`. Le traitement lui-même, réalisé par la routine, pourrait ressembler à :

```
declare
  id integer;
begin
  -- parcours du tableau ids
  for id in array ids loop
    raise notice 'Element_%',id;
    perform sous_traite(id);
  end loop;
end;
```

L'appel précédent produirait :

```
yahozna=> select traite('{2,4,7,23}');
NOTICE: Element 2
NOTICE: Element 4
NOTICE: Element 7
NOTICE: Element 23
traite
-----
(1 row)
```

6

Finalement, le code Php qui permettrait d'appeler une telle procédure serait :

```
$ids=array(2,4,7,23); // les valeurs saisies
$requete="select
          traite('{".implode(",",$ids)."}')";
```



Dans le contexte précis de la routine `traite()` ci-dessus, il serait également possible de ne pas passer par un argument de type tableau, mais de construire dynamiquement une requête constituée de plusieurs appels à la routine `sous_traite()`, sous la forme :

```
begin
  perform sous_traite(2);
  perform sous_traite(4);
  ...
end;
```

Une telle requête pourra être construite au niveau applicatif :

```
$ids=array(2,4,7,23); // les valeurs saisies
$requete="begin ";
foreach($ids as $id) {
    $requete.=" perform sous_traite(".$id.")\n";
}
$requete.="end";
$res=sql_query($conn,$requete);
```



Nous finissons cette section pour montrer qu'avec Oracle, la construction d'une procédure attendant un argument de type tableau sera réalisée sur le même principe mais avec une syntaxe différente. On commencera par déclarer²² un type « tableau d'entiers » :

```
create or replace type tabint as array of int;
```

La routine SQL s'écrira quant à elle comme suit :

```
create procedure traite(ids in tabint) as
begin
    for i in ids.FIRST .. ids.LAST loop
        sous_traite(ids(i));
    end loop;
end;
```

Notez l'utilisation de l'opérateur () au lieu de [] pour accéder à un élément du tableau dans le langage procédural d'Oracle. Notez également qu'il n'est pas nécessaire de déclarer le compteur i. L'appel SQL lui-même devra être de la forme :

```
begin
    traite(tabint(2,4,7,23));
end;
```

Et le code Php pour le produire est laissé en guise d'exercice :-)

6.8.4 Récupérer la valeur d'un argument « out »



Pour finir ce chapitre en beauté nous abordons ici le sujet quelque peu obscur présentant les mécanismes qui vous permettront de récupérer les valeurs d'arguments de type out lors d'un appel de procédure.

Si vous avez lu attentivement la section 6.4.9 page 231, vous serez convaincu que ceci ne pose pas de problème particulier avec PostgreSQL puisque celui-ci gère les arguments de type out comme le

22. Une seule fois.

résultat d'une requête **select**. Ainsi, en reprenant l'exemple de la routine `somme_diff`, on pourra intégrer l'appel dans le code Php :

```
$requete="select * from somme_diff(4,7)";
$res=mysqli_query($conn,$requete);
$stmt=mysqli_fetch($res);
```

Le variable `$tuple` sera alors un tableau associatif composé de deux champs `s` et `d` auquel vous pourrez accéder via les célèbres expressions `$tuple['s']` et `$tuple['d']`.

Pour ce qui est d'Oracle, on passera par les variables liées :

```
$requete="begin somme_diff(4,7,:som,:dif);end";
$req=oci_prepare($conn,$requete);
oci_bind_by_name($req,":som",$somme,8);
oci_bind_by_name($req,":dif",$difference,8);
oci_execute($req);
```

Au retour de `oci_execute` les variables `$somme` et `$difference` contiendront les valeurs de retour de la routine SQL `somme_diff`. Le 4^e argument de la routine `oci_bind_by_name` permet de préciser, dans le cadre d'un résultat (*out*), l'espace de stockage nécessaire. Nous avons donc ici indiqué 8 octets pour l'entier récupéré en retour de l'appel à la procédure `somme_diff`.

Et bien voilà...

6

Dans l'hypothèse où vous auriez lu ce livre jusqu'ici de manière séquentielle, depuis le début, et que tout vous aurait paru clair et limpide, vous êtes donc prêts pour :

1. analyser un système d'information et le modéliser grâce au formalisme entité/association ;
2. l'implémenter dans une base de données à grands coups de **create table** et **alter table** en posant toutes les contraintes d'intégrité nécessaires à la cohérence de vos données ;
3. mettre à jour vos données et générer toutes sortes d'extractions via des **select** avec moult jointures et autres agrégations ;
4. créer des vues, stocker des traitements et faire face aux accès concurrents des applications qui seront développées autour de votre base.

Félicitations;-)



- 7.1 Psql : la console à tout faire
- 7.2 Gestion des droits
- 7.3 Index
- 7.4 I18n : internationalisation
- 7.5 Tables « système »
- 7.6 Sauvegarde et restauration
- 7.7 Épilogue : la petite sirène

Du côté de chez le DBA

*Trying to explain music
is like trying to dance architecture.*

Thelonious Monk.

C^E CHAPITRE qu'au regard de la table des matières vous pourriez juger quelque peu « fourre-tout », s'attarde sur des outils et mécanismes qui concerne l'**administrateur d'une base de données** (DBA pour *database administrator*). Vous trouverez tout d'abord une liste des outils à connaître sur l'interface en mode console de PostgreSQL. Est ensuite présentée la *gestion des droits* dans une base de données. À la lumière de l'exposé sur les algorithmes de ►recherche, vous apprécierez ensuite la présentation des *index*. Vous trouvez alors des pistes pour configurer votre PostgreSQL dans une autre langue que celle par défaut. Vous découvrirez ce qui se cache dans les tables dites « système », puis vous découvrirez les outils basiques pour être capable d'assurer une *sauvegarde* et une *restauration* d'une base PostgreSQL. Finalement le chapitre est clos par un exemple concret (la gestion de la base des entreprises françaises et leurs établissements) mobilisant une bonne partie des concepts présentés dans ce chapitre ainsi que leur mise en œuvre dans des scripts Unix.

§ 2.7 ◀
p. 51

7.1 Psql : la console à tout faire

PostgreSQL est fourni avec un utilitaire nommé `psql` qui vous permettra d'effectuer la grande majorité des tâches de maintenance. D'autres interfaces sont disponibles, parmi les plus connues :

- `pgadmin` (www.pgadmin.org);
- interface Web : `phppgadmin` (phppgadmin.sourceforge.net).

Nous vous proposons ici de nous arrêter sur l'interface de base (`psql`), aride, laide et spartiate, comme tout bon utilisateur de la ligne de commande les aime. Tout d'abord, le lancement :

```
psql -h <hôte> -d <base> -U <utilisateur>
```

où :

- `<hôte>` est la machine sur laquelle tourne PostgreSQL ;
- `<base>` est le nom de la base ;
- `<utilisateur>` est l'utilisateur logique utilisé pour la connexion.

Dans cette console, il faut comprendre que deux types de commandes sont possibles :

1. les commandes de `psql` commençant par un backslash (\);
2. les commandes SQL.



Nous présentons au paragraphe 7.2.6 page 295 quelques-unes des subtilités inhérentes aux connexions locales ou distantes sur un Sgbd tel que PostgreSQL.

7

7.1.1 Prompt (invite de commande)

La console `psql` vous indique qu'elle est prête à répondre docilement à vos ordres grâce à un prompt (mot anglais pour invite de commande) contenant le signe « égal » :

```
yahozna=>
```

Si jamais vous aviez *un autre caractère* que le signe `=`, le danger est imminent ! Si après avoir appuyé fébrilement sur la touche **Entrée**, vous obtenez le prompt :

- `yahozna->` cela signifie que `psql` attend une suite à votre commande ;
- `yahozna)>` cela signifie que vous n'avez pas fermé une parenthèse ;

- yahoza>'> c'est que vous avez commencé une chaîne de caractères sans la refermer.

En fonction du contexte vous pouvez continuer en corrigeant puis en tapant de nouveau sur la touche **Entrée**.



Nous présentons un tout petit peu plus loin (§ 7.1.4 page 287) quelques pistes pour modifier l'apparence de psql.

7.1.2 Passer des commandes SQL

Dans psql il faut penser à terminer sa commande SQL par un point-virgule :

```
yahoza=> select * from latabledelamortquitue ;
... résultat du select ...
yahoza=>
```

Mais vous pourriez également taper votre requête comme ceci :

```
yahoza=> select                                puis Entrée
yahoza-> *                                       puis Entrée
yahoza-> from latabledelamortquitue ;         puis Entrée
... résultat du select ...
yahoza=>
```



C'est le caractère « ; » dans la commande ci-dessus, à la fin de la commande SQL qui déclenche l'envoi de celle-ci au serveur. Sans la saisie de ce point virgule, votre requête reste dans une zone mémoire (buffer) de psql. Et sauf problème de parenthèse ou chaîne de caractères mal fermée, c'est alors le prompt « yahoza-> » qui s'affichera une fois la touche **Entrée** pressée.

Pour sortir d'un mauvais pas lorsque le prompt n'est pas celui qui attend patiemment vos commandes, il est toujours possible d'utiliser la combinaison de touches **Ctrl+C**:

```
yahoza=> insert into table                      puis Entrée
yahoza-> bidule(id,nom values(1,'truc')); puis Entrée
yahoza>                                oubli de la parenthèse, on presse Ctrl+C
yahoza=>
```

D'autres possibilités vous sont offertes pour revenir au « bon » prompt (\Rightarrow), vous les découvrirez si vous prenez le temps de lire la documentation de psql, en particulier ce que renvoie la commande \? (immédiatement ci-après) et sa section « query buffer ».

7.1.3 Commandes spécifiques à la console

Pour commencer :

```
yahozna=> \?
```

nous donne un inventaire exhaustif de toutes les commandes spécifiques à psql. Lorsque vous avez un doute sur la syntaxe d'une requête SQL :

```
yahozna=> \h delete from
```

affiche une aide sur la commande **delete**. Enfin,

```
yahozna=> je commence une commande
yahozna-> \r
Query buffer reset (cleared).
yahozna=>
```

peut aider à sortir de l'écriture d'une requête..

7

Examiner le schéma

C'est la commande \d et ses dérivées qui vous donneront toutes les informations utiles sur les objets de votre base (relations, vues, contraintes associées, fonctions, etc.). Toute nue, cette commande liste les objets de la base :

```
yahozna=> \d
          List of relations
Schema |      Name      |  Type  |  Owner
-----+-----+-----+-----
public | seq_vehicule   | sequence | alpha
public | utilisateur    | table   | postgres
public | v_user         | view    | alpha
public | vehicule       | table   | alpha
....   ...        ...      ...
```

Notez que les commandes \dt, \dv (et d'autres) ne listent respectivement, que les tables ou les vues. Vous pouvez alors obtenir des informations sur l'objet lui-même, par exemple :

```
yahozna=> \d vehicule
          Table "public.vehicule"
  Column |          Type          | Modifiers
  -----+-----+-----
   idv   | integer                | not null
  immat  | timmatriculation       |
  modele | text                   |
Indexes:
    "vehicule_pkey" PRIMARY KEY, btree (idv)
    "vehicule_immat_key" UNIQUE CONSTRAINT, btree (immat)
Referenced by:
    TABLE "autorisation" CONSTRAINT "autorisation_idv_fkey"
    FOREIGN KEY (idv) REFERENCES vehicule(idv)
```

Apparaîtront alors, entre autres choses, des informations sur les attributs de la relation, les contraintes de clés primaires, d'unicité, de clés étrangères. En fonction de la version de la console psql vous pouvez avoir besoin d'ajouter le caractère + :

```
yahozna=> \d+ v_user
          View "public.v_user"
  Column | Type | Modifiers | Storage | Description
  -----+-----+-----+-----+-----
   idu   | integer |          | plain   |
  nom    | text   |          | extended |
View definition:
  SELECT utilisateur.idu, utilisateur.nom
  FROM utilisateur left join ...
```

Sans le « + » la définition de la vue ne sera pas affichée.

Entrée/sortie

Si dans la console vous osiez taper :

```
yahozna=> \o data.txt
```

alors la sortie de toutes les commandes suivantes (qu'elles soient spécifiques à la console ou SQL) sera envoyée dans le fichier indiqué

(ici `data.txt`). Pour restaurer la sortie des commandes dans l'écran de la console, il suffit de taper :

```
yahozna=> \o
```

Au passage on pourra noter que `psql` permet de produire un affichage sans alignement (commande `\a`) et avec uniquement les tuples (sans les entêtes, commande `\t`) :

```
yahozna=> select * from truc
x | y
---+---
1 | 2
5 | 6
(2 rows)
yahozna=> \a
Output format is unaligned.
yahozna=> select * from truc ;
x|y
1|2
5|6
(2 rows)
yahozna=> \t
Tuples only is on.
yahozna=> select * from truc ;
1|2
5|6
```

Ces deux commandes, suivies par un judicieux `\o monfichier.txt`¹ peuvent vous aider à construire un fichier de données à partir de votre base. Il est même possible, grâce à la commande `\H`, d'indiquer que le format de sortie sera du Html.

Inversement il est tout à fait possible d'exécuter un ensemble de requêtes enregistrées dans un fichier texte, grâce à la commande spéciale `\i` :

```
yahozna=> \i requetes.sql
```

qui va docilement exécuter toutes les requêtes SQL présentes dans le fichier `requetes.sql`. Elles devront être terminées par un point-

1. Ou n'importe quel autre nom qui vous plaira, bien entendu.

virgule. Ce mécanisme d'inclusion/exécution de fichier est indispensable à la ►mise au point des vues et des procédures stockées.

§ 6.2.2 ◀
p. 215

Import/export

Dans le même ordre d'idée, la console psql vous permet de produire des fichiers de la famille CSV, pour « comma separated values » (valeurs séparées par des virgules). Pour cela vous pouvez vous référer au paragraphe 5.6 page 201 du chapitre 5.

7.1.4 Personnaliser psql

Avant d'attaquer la gestion des droits, nous proposons ici deux options pour changer quelque peu l'apparence de psql. Le principe consiste à changer des variables internes via la commande `\pset`. D'abord interactivement :

```
yahozna=> \pset null
Null display is "".
yahozna=> \pset null 'ø'
Null display is "ø".
yahozna=> select nom,naissance,deces from musicien;
   nom   | naissance | deces
-----+-----+-----
 Zappa   | 1940-12-21 | 1993-12-04
 Bozzio  | 1950-12-27 | ø
(2 rows)
```

Cette première option (null) vous permet donc d'afficher autre chose que rien ("") pour toutes les valeurs NULL. Ceci pourra par exemple les distinguer de la chaîne vide. Ensuite :

```
yahozna=> \set
...
PROMPT1 '%/%R%# '
...
yahozna=> \set PROMPT1 '%n:%/%R%# '
moi:yahozna=>
```

ajoute au prompt, le nom de l'utilisateur connecté. Ceci va nous être utile pour la section suivante traitant des droits. Pour information, ces deux petits ajustements peuvent être enregistrés dans un

fichier de votre répertoire privé sur votre système Unix préféré :
`~/.psqlrc :`

```
\set PROMPT1 '%n:%/%R%# '
\pset null '∅'
```

et ainsi être actifs à chaque lancement de psql.



Bien d'autres options sont disponibles, vous l'imaginez bien, vous êtes d'ailleurs sans doute déjà en train de compulser fébrilement la documentation de psql pour en savoir plus...

7.2 Gestion des droits

C'est une pratique courante — et recommandée — de créer au sein de la base de données plusieurs utilisateurs logiques ayant chacun des privilèges spécifiques. À l'instar d'un système d'exploitation, lorsque celui-ci est fraîchement installé, le seul utilisateur existant est le « super utilisateur » ou « administrateur ». Cet utilisateur a tous les droits pour agir sur tous les objets :

- lire ou modifier les données ;
- changer la structure ou détruire les tables ;
- modifier ou détruire les procédures ;
- créer d'autres bases ou d'autres utilisateurs ;
- accorder ou révoquer ses propres privilèges aux utilisateurs existants ;
- ...

7

7.2.1 Concepts fondamentaux

On va voir que toute la gestion des accès est possible dans un SGBD grâce aux concepts suivants :

(Groupe d')utilisateurs : les « individus logiques » identifiés sur le système par un login et un mot de passe. Les utilisateurs et groupes d'utilisateurs peuvent être *propriétaires* d'objets ;

Objets : tous les objets (tables, vues, procédures, etc.) pouvant être liés à un (groupe d')utilisateur via un (ou des) privilège(s) particulier(s) ;

Droits/Privilèges : la liste des types d'actions possibles que l'on va accorder ou non à certains utilisateurs (ou groupes) sur

certaines objets : le droit de lire des données, le droits d'ajouter des données, le droit d'exécuter une procédure, etc.

Généralement, on différencie quatre types d'utilisateur selon l'usage du SGBD :

Utilisateur standard : il utilise les données, c'est-à-dire qu'il fait appel aux opérations du volet LMD² de SQL ;

Utilisateur créateur : en plus de l'utilisation, il est amené à créer des des objets personnels contenant des données (volet LDD³ de SQL) ;

Administrateur : il utilise des objets d'administration : sauvegarde, gestion des accès, gestion de l'espace alloué aux données ;

Installeur et système : il crée et utilise des objets d'administration :

- installation du SGBD sur l'OS ;
- communication entre le programme SGBD et l'OS ;
- traitement des demandes de tous les utilisateurs.



Ce qui suit va vous guider pas à pas vers l'administration des utilisateurs et des droits dans une base de données PostgreSQL, pour ce qui est des concepts principaux, ceux indispensables pour commencer.

7.2.2 Ajouter/gérer des utilisateurs

Supposons que vous vous connectiez sur votre SGBD en ayant suivi les recommandations du ►kit de démarrage du chapitre 1 :

```
❏ psql yahozna
```

Une fois connecté, on pourra obtenir des informations :

```
yahozna=> \conninfo
You are connected to database "yahozna" as user "moi"
```

Supposons maintenant que vous souhaitiez créer un nouvel utilisateur dont le propos serait de pouvoir lire des informations de la table `musicien`. Si vous tentiez un :

2. Langage de manipulation de données : le volet de SQL permettant d'agir sur les données contenues dans les relations (ce qui est présenté dans le chapitre 5).
3. Langage de description de données : le volet de SQL présenté au chapitre 4.

```
create role melomane
```

vous n'auriez pas le droit en tant qu'utilisateur moi, de créer d'autre utilisateur :

```
ERROR: permission denied to create role
```

- § 7.5 Ceci peut également être constaté en examinant une des tables système ◀ :
p. 308

```
yahozna=> select  rolname,rolcreaterole from pg_authid ;
rolname  | rolcreaterole
-----+-----
postgres | t
moi      | f
```

L'attribut `rolcreaterole` de la table `pg_authid` indique si oui ou non, le rôle a la permission d'en créer d'autres. À ce stade, il est nécessaire de faire intervenir le super utilisateur (celui qui a tous les droits), afin :

- soit de lui demander de créer le nouvel utilisateur ;

```
create role melomane
```

- soit d'autoriser l'utilisateur moi à créer de nouveaux utilisateurs, au début de la configuration du bazar ⁴ :

```
| createuser --createrole moi
```

- soit d'autoriser l'utilisateur moi à pouvoir le faire lui-même :

```
update pg_authid set rolcreaterole=true
      where rolname='moi';
```

Enfin, la commande ci-dessus étant la version *hacker-du-dark-web-qui-tue*, une plus raisonnable serait :

```
alter role moi createrole
```

7.2.3 Accorder/révoquer des droits

Deux commandes permettent d'accorder ou révoquer les droits :

1. **grant** pour accorder un droit
2. **revoke** pour retirer un droit

4. Souvenez-vous du kit de démarrage, § 1.9 page 17 au tout début du livre.

On pourra par exemple écrire :

```
grant select on musicien to melomane
```

commande qui accordera à l'utilisateur melomane le droit de lire le contenu de la table musicien. Il est intéressant de noter que la commande \dp affichera des informations utiles sur les droits. En particulier la colonne *access privileges* qui vous dira, dans le cas présent pour la table musicien :

- d'une part : `moi=arwdDxt/moi`, indiquant qu'en tant que créateur et donc propriétaire (*owner*) de la relation, l'utilisateur moi a tous les privilèges sur celle-ci ⁵ :

a : **insert** pour ajouter (*append*);

r : **select**, pour lire (*read*);

w : **update** pour écrire (*write*);

d : **delete** pour effacer (*delete*);

- d'autre part : `melomane=r/moi`, indiquant que l'utilisateur melomane a le droit de lecture (r pour *read*), transmis par l'utilisateur moi.

Si jamais vous regrettiez d'avoir autorisé melomane à lire les données de votre table, il vous suffira d'écrire :

```
revoke select on musicien from melomane
```

pour vous rassurer.

7.2.4 Transmettre ses droits à d'autres

Supposons maintenant l'existence d'un troisième larron — l'utilisateur coach — qui aurait pour but d'intervenir sur la relation musicien. Si l'utilisateur melomane souhaitait transmettre à coach ses droits de lecture sur la table musicien :

```
melomane:yahozna=> grant select on musicien to coach;
WARNING: no privileges were granted for "musicien"
```

il serait averti qu'il ne possède pas les droits pour le faire. En revanche, si le droit de lecture avait été initialement accordé à l'utilisateur melomane avec *autorisation de transmission* :

5. Nous passons ici volontairement sous silence les droits D, x et t.

```
moi:yahozna=> grant select on musicien to melomane
moi:yahozna-> with grant option;
GRANT
```

les choses auraient été différentes. En effet, la clause **with grant option** indique que melomane sera habilité à transmettre lui-même ce droit à un autre utilisateur. Ceci apparaîtra avec un astérisque dans la sortie de \dp :

```
moi:yahozna=> \dp

Schema |      Name      |...| Access privileges | ...
-----+-----+-----+-----+-----
public | musicien      |...| moi=arwDxt/moi+   | ...
      |                |...| melomane=r*/moi   | ...
```

Dans ces conditions, melomane pourra à son tour autoriser coach à lire les données de la table musicien :

```
melomane:yahozna=> grant select on musicien to coach;
GRANT
```

Et la commande \dp renverra alors toute la chaîne de transmission :

```
moi=arwDxt/moi
melomane=r*/moi
coach=r/melomane
```

Enfin, on notera que l'utilisateur melomane à qui l'on a octroyé le droit lecture (avec autorisation de transmission à d'autres), n'a pas pour autant le droit de transmettre autre chose. Ainsi :

```
melomane:yahozna=> grant update on musicien to coach;
WARNING:  no privileges were granted for "musicien"
```

indiquera donc qu'aucun privilège n'autorise à transmettre le droit de mise à jour sur la table musicien.



De ces manipulations autour des droits, on retiendra finalement trois idées qui résument une partie de notre propos :

1. le propriétaire/créateur d'un objet a tous les droits et à ce titre peut autoriser d'autres utilisateurs à manipuler les objets ;
2. un utilisateur à qui on a accordé un droit ne peut pas nécessairement le transmettre à d'autres ;
3. c'est lors de l'octroi d'un droit qu'on précise s'il pourra être transmis à d'autres.

7.2.5 Rôle et groupes d'utilisateur

Supposons que pour gérer l'accès à nos relations, nous ayons plusieurs utilisateurs : *melomane*, *coach*, *luthier*, etc. Dans cette situation il peut être fastidieux d'accorder à chaque utilisateur ses propres droits, surtout s'ils sont à peu près les mêmes pour tout le monde. L'idée est donc ici de :

1. créer un groupe d'utilisateurs ;
2. accorder les droits à ce groupe ;
3. ajouter les utilisateurs dans ce groupe.

On voit immédiatement l'intérêt d'une telle configuration : on règle les droits au niveau du groupe et le cas d'un nouvel utilisateur sera rapidement configuré puisqu'il suffira de le joindre à ce groupe. Voici comment ceci est réalisé dans PostgreSQL :

1. création du groupe :

```
create role voir
```

2. les utilisateurs concernés rejoignent le groupe :

```
grant voir to melomane ;  
grant voir to coach ;  
...
```

3. ce groupe a le droit de lire les données de la table *musicien* :

```
grant select on musicien to voir
```

Si vous tentiez d'exécuter ces commandes depuis votre console, vous devriez buter sur le fait que vous n'avez pas le droit de créer de rôle. La commande de l'étape 1 doit donc être exécutée par l'administrateur ou un utilisateur autorisé. Ensuite, il est tout à fait possible de désigner un utilisateur *administrateur du groupe*⁶ :

6. Le prompt sous forme d'un dièse⁷, indique que c'est l'administrateur qui est connecté.

7. « Hashtag » comme disent⁸ les d'jeuns.

8. Qui devraient d'ailleurs dire « hash » tout court, mais ne nous égarons pas.

```
yahozna=# grant voir to moi with admin option;
GRANT ROLE
```

Ceci permettra à l'utilisateur moi :

- d'ajouter lui-même les utilisateurs au groupe voir;
- de modifier lui-même les droits de groupe, par exemple décider que finalement tous les membres de ce groupe ont également le droit de lire les données de la table instrument :

```
moi:yahozna=> grant select on instrument to voir;
```

Il est possible dans la console psql de lister les rôles :

```
bascasable=> \du
```

List of roles		
Role name	Attributes	Member of
coach		{voir}
moi		{voir}
melomane		{voir}
postgres	Superuser	
	Create role, Create DB, ...	{}
voir	Cannot login	{}

7

On notera finalement qu'il est possible de modifier les attributs d'un rôle grâce à célèbre commande **alter**. Ainsi, comme nous l'avons indiqué au début du chapitre :

- on peut accorder à un utilisateur le droit de créer lui-même des utilisateurs :

```
alter role barnabe createrole
```

- ou même des bases de données :

```
alter role ernest createdb
```

- ou tout simplement, soyons fous, d'être administrateur :

```
alter role bart superuser
```

- modifier le mot de passe :

```
alter role ernest password 'xxxx'
```

- ...

7.2.6 Droits et connexions au Sgbd

En règle générale, et même si selon les SGBD et les systèmes utilisés cela peut différer par l'utilisation de raccourcis implicites ou explicites, un utilisateur a besoin de cinq informations pour se connecter :

Serveur : définit l'ordinateur sur lequel se trouvent les informations auxquelles on veut accéder ;

Base de données : définit la base de données contenant les informations sur le serveur ;

Nom d'utilisateur : définit un *user* autorisé de la base ;

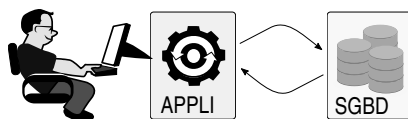
Mot de passe : vérifie que l'on connaît (un peu) l'*user* dont on se sert ;

Rôle : définit quels sont les droits d'accès aux données et privilèges de chaque *user* dans la base.

Les informations ci-dessus apparaissent d'ailleurs toutes (sauf le mot de passe — qui sera saisi interactivement — et le rôle) dans la commande :

```
psql -h zeus -d yahozna -U robert
```

Dans le schéma ci-dessous que nous présentions en introduction :



lorsque qu'une application se connecte au SGBD, elle le fait donc en précisant « login » et mot de passe. **NB** : chaque application n'est pas condamnée à ne se connecter qu'une seule fois au SGBD. En réalité, dans le cas de PHP et de la plupart des langages, une section du type :

```
$conn = pg_connexion('dbname=yahozna
                      user=moi ...');
...
pg_close($conn);
```

peut survenir un grand nombre de fois ! Si bien qu'une même application pourra se présenter au SGBD avec des logins différents. Sans entrer dans le détail de l'administration de PostgreSQL, notons que c'est dans le fichier `pg_hba.conf` qu'on spécifiera toutes les connexions autorisées. Ce fichier contient au moins :

```
local all postgres peer
```

signifiant que l'utilisateur postgres (Unix) pourra se connecter en tant qu'administrateur (utilisateur postgres) si la connexion est initiée depuis la même machine (local) et ce, pour toutes les bases. Lorsque vous tapez :

```
psql mabase
```

PostgreSQL cherche à vous connecter avec un utilisateur portant le même nom que l'utilisateur système avec lequel vous lancez la commande psql (pour rappel, nous avons appelé cet utilisateur moi). Si vous souhaitez pouvoir vous connecter sous l'utilisateur coach *tout en étant connecté en tant que moi* sous votre Unix, il faudra taper :

```
psql mabase coach
```

Mais il faudra au préalable ajouter dans le pg_hba.conf :

```
local yahoza coach ident map=m_yahoza
```

puis ajouter dans le fichier pg_ident.conf qui doit se trouver dans le même répertoire que le précédent :

```
m_yahoza moi coach
m_yahoza moi melomane
```

permettant à l'utilisateur Unix moi de se faire passer pour l'utilisateur PostgreSQL coach et aussi en passant, pour melomane. Notez que le nom de la table des correspondances⁹ — ici m_yahoza — est à votre discrétion.

Enfin, en fonction de l'environnement déployé — dans lequel l'application ne tourne pas nécessairement sur la machine où s'exécute le SGBD — on devra sans doute ajouter des lignes telles que :

```
host yahoza melomane 192.168.3.17/32 md5
```

précisant que la base de données yahoza sera accessible depuis la machine d'adresse IP (192.168.3.17) par l'utilisateur melomane à condition qu'il montre patte blanche avec un mot de passe.

9. Appelée map dans le jargon de PostgreSQL.

7.2.7 Encore un peu plus loin...

En parcourant la documentation officielle de PostgreSQL, vous trouverez bien d'autres fonctionnalités autour des droits dans une base de données. Nous listons ci-dessous une partie de celles-ci :

Autres droits : on pourra spécifiquement autoriser à *exécuter une fonction, référencer une table, se connecter à une base, de créer des triggers*, etc.

Droits sur les colonnes : il est possible d'autoriser la modification d'une relation en se limitant à une colonne :

```
grant update(prenom) on musicien  
to melomane
```

Schéma : PostgreSQL et Oracle proposent de créer à l'intérieur d'une base de données des *schémas* qui peuvent être vus comme des dossiers contenant chacun les objets habituels (relation, vues, procédures, etc.). Une fois le schéma créé, on devra préfixer ses objets avec le nom d'icelui. Par exemple, l'administrateur dirait :

```
create schema jazz
```

Puis, pour les utilisateurs qui seraient autorisés (par un **grant** bien sûr) à agir sur ce schéma :

```
create table jazz.free(f int , lib text)
```

Dans PostgreSQL, il existe un schéma par défaut qu'il n'est pas nécessaire d'indiquer comme préfixe : le schéma public. C'est lui qui apparaît lorsque vous listez les objets d'une base avec la commande \d.

Package : en plus des schémas, Oracle propose de rassembler les procédures dans des *packages*. Ceci permet d'organiser les traitements par thèmes. Une procédure stockée dans un package devra être préfixée par le nom de celui-ci pour être appelée.

Profile : dans certains SGBD dont Oracle, en plus des rôles on ajoute une notion de profil utilisateur (*profile*) qui permet de limiter la consommation des ressources par user. Il ne s'agit plus de contrôler l'accès aux données mais de gérer les ressources système (CPU, RAM, etc.) qu'utilise le SGBD. Ces ressources n'étant pas infinies, cela permet de répartir entre plusieurs users le temps d'accès aux données.

7.3 Index



Si vous voulez avoir une chance de comprendre l'essentiel de ce que nous racontons ici, il est indispensable de lire les quelques pages consacrées aux algorithmes fondamentaux de recherche.

► § 2.7
p. 51

Pour présenter le concept d'*index*, nous nous appuierons sur une relation contenant une clé de substitution et un attribut de type texte dont les valeurs sont générées aléatoirement¹⁰. En voici un extrait :

truc	
id	libelle
6391103	YITSBJKHWW
8904368	ZYMXRTLPEF
1701551	LIWZMOAPFY
4038765	VGUMOPZTOZ
...	...

De manière à illustrer l'intérêt des index dans les bases de données, nous avons créé cette table avec dix millions de tuples. En examinant via la console, la structure de la table on trouvera :

```
yahozna=> \d truc
```

Column	Type	Modifiers
id	integer	not null
libelle	character varying(10)	

Vous noterez donc *que nous avons sciemment omis de doter l'attribut id d'une clé primaire*. Si bien qu'une requête telle que :

```
select * from truc where id=6789322
```

contraindrait le SGBD à *scanner séquentiellement* tous les tuples de la relation truc jusqu'à trouver (ou non) la valeur 6789322 recherchée dans l'attribut id. Ceci peut être constaté grâce à la commande explain qui montre à l'utilisateur ce que le *planificateur de requête* a l'intention de faire, ainsi qu'une idée du coût de l'opération :

10. Les fonctions permettant de réaliser cette génération sont présentées dans la section traitant des procédures stockées.

```
yahozna=> explain select * from truc where id=6789322;
               QUERY PLAN
-----
Seq Scan on a  (cost=0.00..179053.25 rows=1 width=15)
  Filter: (id = 6789322)
```

Les deux valeurs de *cost* (coût en anglais) indiquent respectivement le coût pour obtenir les premières valeurs de la requête et le coût pour l'intégralité des tuples. On peut également demander à PostgreSQL de simuler la requête, le temps est alors indiqué :

```
yahozna=> explain (analyze true)
yahozna-> select * from truc where id=6789322;
               QUERY PLAN
-----
Seq Scan on a  (cost=0.00..179053.25 rows=1 width=15)
              (actual time=212.603..799.900 rows=1 loops=1)
  Filter: (id = 6789322)
 Rows Removed by Filter: 9999999
 Planning time: 0.036 ms
 Execution time: 799.915 ms
```

On constate donc que la recherche séquentielle exige une petite seconde pour localiser le tuple.

7.3.1 Index implicite avec une clé primaire

En dotant l'attribut *id* de la table *truc* d'une clé primaire :

```
alter table truc add constraint pk_truc
primary key(id)
```

Un index est donc explicitement créé, comme vous l'avez sans doute déjà constaté :

```
yahozna=> \d truc
      Column      |      Type      |      Modifiers
-----+-----+-----
 id              | integer        | not null
 libelle         | character varying(10) |
Indexes:
    "pk_truc" PRIMARY KEY, btree (id)
```

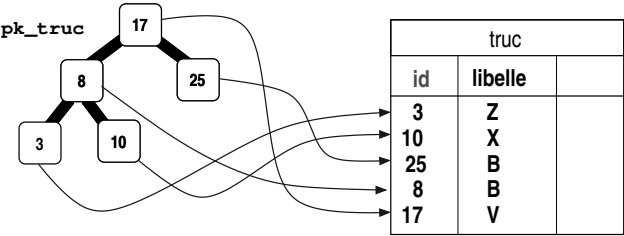


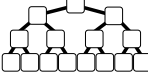


FIGURE 7.1 – Un index posé sur une table via la clé primaire : c’est un arbre (ici binaire pour simplifié) dans lequel les données de la clé sont rangées de manière à en optimiser la recherche.

La dernière ligne de cet affichage nous indique donc que la clé primaire a généré pour nous un index. Au passage, notons le mot « *btree* » qui devrait, si vous avez lu le chapitre 2 page 21, vous mettre la puce à l’oreille :

*Bon sang mais c’est bien sûr !
un arbre pour stocker toutes les clés !!
et ainsi accélérer les recherches !!!*

La figure 7.1 illustre cette idée : la table `truc` est dotée d’une structure supplémentaire sous la forme d’un arbre de recherche dont les nœuds pointent sur chacun des tuples via la clé primaire `id`. Si bien que lorsqu’il faudra chercher quel est le tuple dont l’attribut `id` vaudra telle valeur, vous l’aurez compris, le SGBD parcourra l’arbre et non pas la table elle-même. Comme montré au paragraphe 2.7.3 du chapitre 2, le parcours d’un arbre (binaire) de recherche est proportionnel à la hauteur de celui-ci. Et le nombre de données mémorisables dans un arbre est de l’ordre de 2^{Hauteur} . Pour rappel :

Arbre	hauteur H	Nombre de données
	2	$3 = 2^2 - 1$
	3	$7 = 2^3 - 1$
	4	$15 = 2^4 - 1$
...
...	10	$1023 = 2^{10} - 1$
...	20	$1048575 = 2^{20} - 1$
...	30	$1073741823 = 2^{30} - 1$



Nous illustrons ici l'intérêt et le principe des index avec un arbre binaire. Ce qui n'est pas le cas dans l'exemple concret de ceux construits via les clés primaires de PostgreSQL, puisque celui-ci a recours aux B-Arbres (cf. 2.7.4 page 63). Le principe n'en reste pas moins vrai : on exploite une structure arborescente conçue pour accélérer — de manière quasi optimale — la recherche d'une information particulière dans un ensemble en contenant plusieurs.

Et comme précédemment on peut interroger le SGBD afin qu'il nous informe sur ses intentions :

```
yahozna=> explain
yahozna-> select * from truc where id=6789322;
          QUERY PLAN
-----
Index Scan using pk_truc on truc
  (cost=0.43..8.45 rows=1 width=15)
Index Cond: (id = 6789322)
```

Le coût permettant d'obtenir tous les tuples (8,45) est à rapprocher du coût de cette recherche sans index, vu plus haut, à savoir 179053,25. En ajoutant l'option idoine pour une estimation du temps on a :

```
yahozna=> explain
yahozna-> (analyze true)
yahozna-> select * from truc where id=6789322;
          QUERY PLAN
-----
Index Scan using pk_truc on truc
  (cost=0.43..8.45 rows=1 width=15)
  (actual time=0.012..0.012 rows=1 loops=1)
Index Cond: (id = 6789322)
Planning time: 0.058 ms
Execution time: 0.027 ms
```

Ce qui nous confirme également que le temps d'exécution de la requête passe d'environ 800 millisecondes (sans index) à environ 0.03 milliseconde avec. *L'index apporte donc ici un gain de rapidité d'exécution dans un rapport d'environ 20000!*

7.3.2 Index sur un champ non unique

Considérons maintenant la commande :

```
select * from truc
where libelle='QZDFIJCWDV'
```

Celle-ci entraîne le scan complet de la table (et donc des dix millions de tuples) afin de savoir si oui ou non la chaîne 'QZDFIJCWDV' est présente dans la table. On comprend aisément qu'un tel traitement sera d'autant plus contraignant que le nombre de tuples est important. C'est la raison pour laquelle, on peut créer un index sur la table :

```
create index idx_truc_lib
on truc ( libelle )
```

En examinant la table via la console on trouvera :

```
yahozna=> \d truc
Column |          Type          | Modifiers
-----+-----+-----
id      | integer                | not null
libelle | character varying(10) |
Indexes:
    "pk_truc" PRIMARY KEY, btree (id)
    "idx_truc_lib" btree (libelle)
```

La clause Indexes fait ici apparaître :

- une entrée pour l'index de la clé primaire;
- une entrée pour l'index nouvellement créé.

Cette configuration est illustrée à la figure 7.2 page suivante. Une analyse de la requête donne les résultats suivants pour notre table test :

```
yahozna=> explain analyze select * from truc
yahozna-> where libelle='DTHAMTAAHK' ;
          QUERY PLAN

-----
Index Scan using idx_truc_lib on truc
  (cost=0.43..8.45 rows=1 width=15)
  (actual time=0.020..0.020 rows=0 loops=1)
    Index Cond: ((libelle)::text = 'DTHAMTAAHK'::text)
Planning time: 0.099 ms
Execution time: 0.042 ms
```

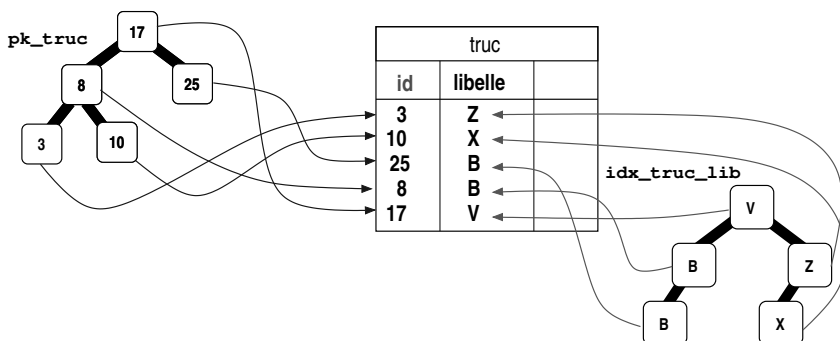


FIGURE 7.2 – Deux index posés sur une table : un via la clé primaire, l'autre sur le champ `libelle`. Chacun des arbres contient les valeurs des attributs associées (`id` et `libelle`). Toute recherche sur l'un ou l'autre de ces attributs sera accélérée de manière optimale, par la structure arborescente.

Pour information, la même requête sans index, donne un temps de l'ordre de 900ms, c'est-à-dire environ 20000 fois plus important !

7.3.3 Les index, la panacée ?

Avantage : l'accélération des traitements, en particulier ceux où il y a un **where**, y compris les **update** et **delete**, ainsi que les jointures.

Inconvénients : en créant un index, on ajoute des données — celles de l'arbre — dans la base, par conséquent la taille de la base augmente (voir plus bas au § 7.5.3 page 312 pour un ordre d'idée sur notre table test). Qui plus est, une modification des données entraîne une mise à jour de l'index et donc un ralentissement des traitements de mises à jour — même si les structures de données associées sont conçues pour optimiser la modification de leur contenu (voir le chapitre 2 à ce sujet).

Il y a un index, oui ou non ?!

Poser un index sur un attribut d'une relation ne garantit pas que celui-ci sera sollicité. C'est le *query planer* (le planificateur de requête) qui va décider en fonction de différents critères, s'il est judicieux de parcourir l'index pour votre requête.

La relation est trop petite — Il est fort probable qu'en dessous d'un certain nombre de tuples, parcourir l'index n'apporte pas de gain de temps. Dans cas, malgré la présence d'un index, le planificateur pourra décider de faire un « seq scan » sur la table.

L'opérateur n'est pas adapté — Un index est mis à contribution lors d'une requête incluant un **where** uniquement si ce **where** est suivi d'une expression faisant appel à certains opérateurs de comparaison. Ainsi, pour la requête :

```
select * from truc where libelle like 'THEUSZ%'
```

le planificateur *ne fera pas appel à l'index!* (mais lisez plus loin...)

Trier une relation — Avec un index sur le libellé de notre table truc, l'index est bien mis à contribution pour une requête comme celle-ci :

```
select * from truc order by libelle
```

L'arbre et la table — Ne perdons pas de vue que les données sont dans la relation (la table) et une partie de celles-ci également dans l'index (l'arbre). Ainsi il est intéressant de comprendre les subtilités qui vont suivre. D'abord ¹¹ :

```
yahozna=> explain select id from truc where id>5000000;
          QUERY PLAN
-----
Index Only Scan using pk_truc on truc
```

Notons ici le **Only** qui précise qu'il suffit de parcourir l'arbre pour obtenir les données demandées (attribut **id** dans la clause **from**). En d'autres termes *il est inutile de faire appel aux données de la table*. Par contre, en demandant un attribut qui n'est pas dans l'arbre, on aura la surprise suivante :

```
yahozna=> explain
yahozna-> select libelle from truc where id>5000000 ;
          QUERY PLAN
-----
Seq Scan on truc
```

11. L'affichage est tronqué par souci de concision.

Oui vous avez bien lu, ici le planificateur estime que ça ne vaut pas le coup d'utiliser l'arbre puisqu'il faudrait pour chaque tuple dont l'id est supérieur à 5000000, aller chercher le libelle dans la table elle-même. Autant tout chercher dans la table.

Taille des index

On a vu que chaque index crée un arbre. Si on se limite aux deux index de la table truc (l'un pour la clé primaire, l'autre pour le libellé), nous avons pour nos dix millions de tuples, « en gros »¹² :

- 400 Mo pour la table ;
- 200 Mo pour l'index associé à la clé primaire ;
- 300 Mo pour l'index associé au libellé.

Les index augmentent donc la taille de la table de 100 % !

Index sur des expressions



Ce qui suit est exclusivement réservé au Sgbd PostgreSQL. Il faudra que vous cherchiez, si nécessaire, l'équivalent pour votre propre système.

Nous avons constaté plus haut, que malgré la présence d'un index sur l'attribut libellé de la relation truc celui-ci n'était pas utilisé lors d'une recherche comme :

```
select * from truc
where libelle like '%TAAHK'
```

Pour permettre à PostgreSQL d'utiliser un index sur ce type d'expression, on créera un index spécial destiné aux opérateurs sur le type chaîne de caractères varchar :

```
create index idx_lib_like
on truc (libelle varchar_pattern_ops)
```

Une fois cet index posé, on pourra se conforter dans l'idée qu'il est bel et bien utilisé dans le cadre d'un **select** :

```
yahozna=> explain select libelle
yahozna-> from truc where libelle like 'D%K';
               QUERY PLAN
-----
Index Only Scan using idx_lib_like on truc (cost=...)
```

12. La taille des objets est obtenue grâce aux fonctions système présentées au § 7.5.3 page 312.

```

Index Cond: (      (lib ~>=~ 'D'::text)
                AND (lib ~<~ 'E'::text) )
Filter: ((libelle)::text ~~ 'DH%AK'::text)

```

Autre requête n'effectuant pas de Index Only Scan :

```

yahoyna=> explain select libelle from truc
yahoyna=> where libelle like 'D%';
               QUERY PLAN
-----
Bitmap Heap Scan on truc (...)
  Filter: ((libelle)::text ~~ 'D%K'::text)
    -> Bitmap Index Scan on idx_lib_like (...)
          Index Cond: (((libelle)::text ~>=~ 'D'::text)
                        AND ((libelle)::text ~<~ 'E'::text))

```

Dans ce cas, le planificateur de requête de PostgreSQL estime qu'il faut dans un premier temps utiliser l'index (Bitmap Index Scan) pour localiser les entrées correspondant à la condition, puis dans un deuxième temps récupérer les données de la table à proprement parler (Bitmap Heap Scan). La lecture de PostgreSQL (2017a) peut aider à comprendre les rouages internes des index dans PostgreSQL.



Un exemple d'utilisation des index sur un cas concret — la liste des établissements des entreprises françaises connue sous le nom de base SIRENE — est présenté à la fin de ce chapitre, à la section 7.7 page 317.

7

7.4 I18n¹³: internationalisation



Cette section présente les étapes à suivre pour configurer votre PostgreSQL en français. On supposera ici que le système d'exploitation sous-jacent est un Linux.

Il faudra d'abord s'assurer que les « locales » sont générées sur votre système, et ce grâce à la commande :

```
locale -a
```

qui devrait afficher entre autres choses :

¹³. Recomptez si vous voulez, il y a bien 18 lettres entre le premier i et le dernier n du mot internationalisation.

```
...
fr_FR.utf8
en_GB.utf8
...
```

S'il n'y a pas de ligne commençant par `fr_FR`, il faudra explicitement générer cette locale avec la commande :

```
| locale-gen fr_FR.utf8
```

Vous procéderez ensuite à la création d'un « cluster » PostgreSQL avec la locale française : tout d'abord demandez à votre système d'afficher la liste des clusters ¹⁴ :

```
| pg_lsclusters
```

```
Ver Cluster Port Status Owner    Data dir   Log file
9.5 main     5432 online postgres 9.5/main   pg-9.5-main.log
```

Il y a donc ici un cluster accessible sur le port 5432 (port par défaut). Créons-en un sur le port 5433, pour une configuration en français :

```
| pg_createcluster -p 5433 --locale fr_FR.utf8 9.5 fr
```

qui se fendra d'un :

```
Creating new cluster 9.5/fr ...
config /etc/postgresql/9.5/fr
data   /var/lib/postgresql/9.5/fr
locale fr_FR.utf8
socket /var/run/postgresql
port   5433
```

Un nouvel appel à la commande `pg_lsclusters` devrait afficher :

```
Ver Cluster Port Status Owner    Data dir   Log file
9.5 fr      5433 down   postgres 9.5/fr     pg-9.5-fr.log
9.5 main    5432 online postgres 9.5/main   pg-9.5-main.log
```

Vous noterez que votre nouveau cluster — que nous avons nommé `fr`, cela ne vous aura pas échappé — est « *down* ». Qu'à cela ne tienne, un petit :

```
| pg_ctlcluster 9.5 fr start
```

le mettra en route.

14. Les chemins vers le dossier et les fichier de log ont été ici raccourcis pour des raisons de mise en page.

► § 1.9
p. 17

À ce stade nous vous invitons à reprendre les instructions du kit de démarrage pour créer une base et un utilisateur dans ce cluster. N'oubliez pas non plus d'ajouter les droits de connexion. En gros :

```
- createdb -p 5433 yahozna
- createuser -p 5433 moi
```

devraient suffire à créer une base et un utilisateur. Vous devriez alors être en mesure de vous connecter sur le cluster français en ajoutant, comme ci-dessus, l'option `-p` à la commande `psql` :

```
■ psql -p 5433 yahozna
```

Enfin, pour vous convaincre que votre système est en français, testez la commande suivante (voir 4.8.3 page 137 pour `to_char`) :

```
yahozna=> select to_char(now(), 'TMday DD TMmon');
           to_char
-----
dimanche 16 avril 2017
```

qui affiche — grâce aux deux préfixes `TM` — sous vos yeux ébaubis, le jour de la semaine et du mois en français !



Il peut être utile de définir le fuseau horaire par défaut de votre cluster, s'il ne correspond pas à celui de votre système d'exploitation, en ajoutant dans le fichier `postgresql.conf` du répertoire `/etc/postgresql`, la ligne : « `timezone='Europe/Paris'` »

7 7.5 Tables « système »

7.5.1 Catalogue

Une chose surprenante dans les SGBD tient dans le fait qu'une grande partie de leur configuration est également stockée sous forme relationnelle. En d'autres termes :

- la liste des processus en cours d'exécution est une vue ;
- la liste des tables d'une base est stockée dans une table ;
- la liste des contraintes d'intégrité également ;
- ainsi que la liste des utilisateurs de la base et leurs droits ;
- et que la liste des bases aussi !
- ...

Bref, vous l'aurez compris, il existe un ensemble de tables et de vues permettant d'obtenir des informations, non seulement sur les objets de la base eux-même, mais aussi sur la configuration du SGBD. Dans PostgreSQL, ces objets sont préfixés par « `pg_` ». Pour se faire

quelques nœuds au cerveau, sachez que nous pouvons répondre à la question¹⁵ : « *combien y a-t-il de tables commençant par pg_ ?* », précisément en faisant une requête sur la table système pg_tables :

```
select * from pg_tables
      where tablename like 'pg\__%'
```

qui renverrait quelque chose comme :

schemaname	tablename	tableowner	...
-----+	-----+	-----+	...
...
pg_catalog	pg_constraint	postgres	...
pg_catalog	pg_index	postgres	...
pg_catalog	pg_operator	postgres	...
pg_catalog	pg_am	postgres	...
...			

La liste des vues système, dont le nom est également préfixé par pg_, peut également être obtenue comme suit :

```
select * from pg_views
      where viewname like 'pg\__%'
```

Et hop, un petit extrait également :

schemaname	viewname	viewowner
-----+	-----+	-----+
...
pg_catalog	pg_tables	postgres
pg_catalog	pg_indexes	postgres
pg_catalog	pg_stats	postgres
pg_catalog	pg_locks	postgres
...		

Au passage vous apprendrez que les objets pg_tables et pg_views sont des vues. Dans le même ordre d'idée, il peut être intéressant de connaître la liste des bases de données opérées par votre PostgreSQL en tapant — vous l'auriez deviné tout seul :

```
select * from pg_database
```

15. Qui vous brûle les lèvres, ne vous mentez pas à vous-même.

Si vous vouliez savoir quels sont les utilisateurs logiques référencés sur votre SGBD préféré, fendez-vous d'un :

```
select * from pg_user
```

Par ailleurs,

```
yahozna=> select * from pg_timezone_names
yahozna-> where name like '%Paris%';
      name      | abbrev | utc_offset | is_dst
-----+-----+-----+-----
posix/Europe/Paris | CEST   | 02:00:00   | t
Europe/Paris      | CEST   | 02:00:00   | t
(2 rows)
```

► § 4.8.5
p. 141

vous informe qu'il existe manifestement deux noms pour le fuseau horaire de Paris, et que ses habitants sont passés à l'heure d'été (is_dst = *is daylight saving true*).

Un petit dernier pour la route : obtenir la liste des contraintes d'intégrité. Celles-ci sont stockées dans la table pg_constraint et peuvent être listées comme ceci :

```
yahozna=> select conname,contype,conrelid
yahozna-> from pg_constraint ;
      conname      | contype | conrelid
-----+-----+-----
instrument_pkey    | p       |    99047
prime_pkey         | p       |   140940
prime_idtp_fkey    | f       |   140940
deces_valide       | c       |   99055
....
```

On constate ici que la colonne contype indique le type de contrainte (p pour clé primaire, f pour clé étrangère, c pour contrainte de type **check**, etc.). La colonne conrelid quant à elle, contient un entier identifiant la relation de la base sur laquelle est posée la contrainte.

En transtypant la colonne, on peut obtenir le nom en clair de cette table :

```
yahozna=> select conname,contype,conrelid::regclass
yahozna-> from pg_constraint ;
```

conname	contype	conrelid
instrument_pkey	p	instrument
prime_pkey	p	prime
prime_idtp_fkey	f	prime
deces_valide	c	musicien
....		

Il y a bien entendu un grand nombre d'attributs attachés à la table `pg_constraint` qui vous permettront si nécessaire, de produire dans le résultat du **select** :

- la table cible dans le cas d'une clé étrangère (`conrelid`);
- le type d'action en cas de suppression de données référencées par un clé étrangère (`confupdtype`);
- si la contrainte est « reportable » (`condeferrable`);
- ...

7.5.2 Quelques stats

La table `pg_stat_activity` vous permettra d'observer l'activité en cours sur vos bases et en particulier les requêtes en cours d'exécution. Cette table système comporte, entre autres, les attributs suivants :

```
yahozna=> select datname as base,
yahozna->          username as user,
yahozna->          application_name as app, query
yahozna-> from pg_stat_activity;
```

base	user	app	query
loz	lozano	psql	select datname, ...
mabase	invite	php	select schtroumph from gargamel

(2 rows)

Grâce à cette table, on pourra par exemple découvrir que derrière le \d de psql se cache un monstrueux :

```

select n.nspname as "Schema",
c.relname as "Name",
case c.relkind
    when 'r' then 'table'
    when 'v' then 'view'
    when 'm' then 'materialized_view'
    when 'i' then 'index'
    when 'S' then 'sequence'
    when 's' then 'special'
    when 'f' then 'foreign_table'
end as "Type",
pg_catalog.pg_get_userbyid(c.relowner) as "Owner"
from pg_catalog.pg_class c
    left join pg_catalog.pg_namespace n
        on n.oid = c.relnamespace
where c.relkind in ('r','v','m','S','f','')
    and n.nspname <> 'pg_catalog'
    and n.nspname <> 'information_schema'
    and n.nspname !~ '^pg_toast'
and pg_catalog.pg_table_is_visible(c.oid)
order by 1,2

```

J'espère que vous savourerez cette jointure qui va donc chercher parmi tous les objets de la base :

- les différents objets de la base dans la relation `pg_class`;
- le type de ces objets dans l'attribut `pg_class.relkind`.

7

7.5.3 Tailles des objets

Tous les objets (relations, vues, index, procédures, etc.) finissent par occuper de la place en termes de fichiers du système d'exploitation sous-jacent. Sur un PostgreSQL hébergé par un système Linux, le répertoire qui contient ces fichiers est, sauf indication contraire :

```
/var/lib/postgresql/9.x/main
```

si votre version est la 9.x et le cluster actif est le cluster par défaut (cf. § 7.4 page 306 pour la signification de ce mot dans le contexte de PostgreSQL). Relativement à ce répertoire vous pouvez, si vous êtes curieux, voir dans quel fichier est stocké chaque objet :

```

yahoza=> select pg_relation_filepath('matable');
pg_relation_filepath

```



```
-----
base/16384/16501
```

Vous pouvez même obtenir la taille en octets d'un des objets de votre base, par exemple notre table de test comportant dix millions de tuples :

```
yahozna=> select pg_relation_size('truc');
pg_relation_size
-----
          442818560
(1 row)
yahozna=>
```

Ou celle de l'index implicitement créé par la clé primaire :

```
yahozna=> select pg_relation_size('pk_truc');
pg_relation_size
-----
        224641024
(1 row)
yahozna=>
```

On constate ici que l'index va augmenter la taille de notre base à hauteur de 50% de la taille de la relation elle-même (environ 400 méga octets). Pour terminer sur la taille des index, on en profitera pour se persuader que la taille d'un « arbre-index » pour stocker des entiers (stockés chacun sur 4 octets) est moins importante que celle d'un même type d'arbre pour des chaînes de caractères 10 octets :

```
yahozna=> select
yahozna-> pg_size_pretty(pg_relation_size('idx_truc_lib'));
pg_size_pretty
-----
          301 MB
(1 row)
yahozna=>
```

vous noterez au passage — cadeau de la maison — la fonction `pg_size_pretty` qui formate « joliment » une taille initialement exprimée en octets, en méga-octets.

7.6 Sauvegarde et restauration

La sauvegarde d'une base est une activité incontournable pour la gestion de vos données. Ces deux tâches — la *sauvegarde* mais surtout la *restauration* — font même l'objet d'un métier à part entière : l'administrateur de bases de données (*database administrator*) qui est aussi en charge de gérer l'espace occupé par les données et leur croissance. Cette activité peut être vue comme l'assurance tous risques qui vous permet d'être sûr que vous ne perdrez jamais rien¹⁶. Il existe plusieurs techniques et politiques de sauvegarde et le SGBD que vous utiliserez pourra vous fournir des fonctionnalités spécifiques et plus perfectionnées que la méthode basique (mais néanmoins utile à connaître) que nous proposons ci-dessous, après l'introduction bien longue de ce chapitre.

7.6.1 Stratégies de sauvegarde

Il existe autant de stratégies de sauvegardes que de types d'utilisation des bases de données. Dans certaines applications dont la disponibilité n'est pas critique, on peut mettre le système à l'arrêt complet. Il n'est alors pas disponible (ou pas utilisé) pendant un certain temps. On parle de *sauvegarde à froid*. Dans d'autres cas, on doit être capable de sauvegarder alors que le système est ouvert aux utilisateurs : on parle de *sauvegarde à chaud*.

En plus de la disponibilité du système, il faut aussi prendre en considération la *fréquence*. De celle-ci dépend l'étendue de la perte de données que l'on s'autorise : une sauvegarde quotidienne qui ne fonctionne pas et c'est la restauration d'une journée de données qui devient impossible. On considérera également le type de données à conserver : est-ce qu'il est important de conserver le *résultat* uniquement ou l'*historique* de toutes les transactions effectuées, comme dans les systèmes bancaires ?¹⁷

La plupart des problèmes ne se rencontre pas lors de la sauvegarde (étape *a priori* où tout se passe bien) mais lors de la restauration. En effet, si nous avons besoin de restaurer *a posteriori* c'est qu'un problème mineur¹⁸ ou majeur est apparu : crash de disques

16. C'est une vue de l'esprit selon la loi de Murphy : si vous utilisez une base, vous allez faire des erreurs et finir par perdre des éléments.

17. Sur mon compte bancaire, un crédit de 10 € suivi d'un débit de 10 € n'aura pas la même importance qu'un débit de 10000 € suivi quelques jours plus tard d'un crédit de 10000 €. Pourtant le résultat est le même, sauf pour les aggios.

18. Fausse manipulation dans l'interface chaise/clavier, le plus souvent.

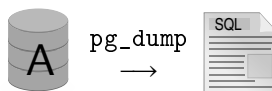
durs, plantage de baies entières, incendie ou inondation dans le bâtiment¹⁹. Il faut alors pouvoir restaurer en évitant ou minimisant si possible la perte de données²⁰ et ce dans le moins de temps possible afin de limiter l'arrêt de la base pour les utilisateurs. On parle alors de *niveau de service* ou de *taux de disponibilité* du système.

On ne peut donc considérer une méthode de sauvegarde comme fiable que si **l'ensemble du processus — sauvegarde puis restauration — a été testé et est re-testé régulièrement**. Cela doit devenir un processus ennuyeux tellement il est rodé, au lieu d'être un grand saut dans l'inconnu comme trop souvent.

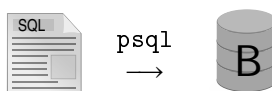
7.6.2 Dump/restore basique

Revenons-en à une méthode basique de sauvegarde / restauration : une des façons de procéder pour la *sauvegarde et la restauration* dans le cadre de PostgreSQL consiste, dans les grandes lignes, à :

1. **sauvegarder** tout ou partie des données (ou même uniquement le schéma des données) via l'utilitaire `pg_dump`, *dans un fichier texte* contenant des commandes SQL ;



2. **restaurer** votre base (ou une autre) en transmettant ce fichier à `psql` ce fichier :



Ainsi, pour sauvegarder la base à laquelle vous vous connectez habituellement, en écrivant quelque chose du genre :

```
| psql ma_super_base
```

19. Cf. par exemple la fuite de watercooling chez OVH en juin 2017 entraînant la mise à l'arrêt pendant une journée de 50000 sites web sur 3 millions et la perte des données sur une période de 1h à 22h pour certains. (<https://www.ovh.com/fr/blog/hebergements-web-post-mortem-incident-29-juin-2017/>)

20. Note d'un administrateur de base de données : « *le moins j'ai à restaurer, le mieux je me porte. Car je peux toujours ajouter des erreurs ou écraser des données non encore sauvegardées en restaurant les données plus anciennes* ».

vous écrirez :

```
pg_dump ma_super_base
```

La commande ci-dessus envoie dans le terminal le code SQL qui, *s'il est exécuté dans une base vide*, reconstruirait intégralement la base `ma_super_base`. Par conséquent :

```
pg_dump ma_super_base > sauvegarde.sql
```

va stocker toutes ces commandes dans le fichier `sauvegarde.sql`. La restauration peut alors être réalisée comme suit :

```
psql -f sauvegarde.sql une_autre_base_que_jai
```

qui, à condition que la base `une_autre_base_que_jai` soit vide, restaurera l'intégralité des données de `ma_super_base` dans celle-ci. Parmi les options de `pg_dump` susceptibles de vous intéresser :

Ne (pas) sauvegarder (que) les données : c.-à-d. ne sauvegarder que les schéma ou que les données :

- `--data-only` : ne sauvegarder que les données ;
- `--schema-only` : ne sauvegarder que la description des données.

Choisir les éléments : on peut stipuler les objets (tables, vues, etc.) que l'on veut sauvegarder :

- `--table=bidule` pour ne sauvegarder que la table `bidule` ;
- `--table=truc*` pour sauvegarder les tables dont le nom commence par `truc` ;
- `--exclude-table=machin` pour ne pas inclure la relation `machin` dans la sauvegarde.

Repartir sur du propre : l'option `--clean` produira une série d'instructions SQL qui détruira tous les objets avant de tout recréer. Ceci peut permettre, dans certaines limites, de faire la restauration dans une base non vidée de son contenu.

Par exemple, la commande :

```
pg_dump tbo --exclude-table=mv* \
            --clean \
            --schema-only > sauve.sql
```

crée un fichier `sauve.sql` contenant les commandes permettant de créer l'ossature de la base `tbo` (même si celle-ci n'est pas vide), sans y inclure les objets dont le nom commence par « `mv` ».

7.7 Épilogue : les merveilleuses aventures de la petite sirène



Cette dernière section à la fin de ce chapitre orienté « administration » a pour but, sur un exemple concret et vécu, de faire fonctionner ensemble plusieurs outils et concepts déjà rencontrés jusqu'ici : le langage procédural de SQL, les outils d'administration, les index et leur utilité sur des tables volumineuses, la création de scripts pour la génération et la mise à jour automatique de relations, et d'en rencontrer de nouveaux, comme les commandes **truncate** et **vacuum**.

Depuis le 1^{er} janvier 2017, dans un souci « d'ouverture des données publiques de l'État et des administrations », une plateforme du service public (<http://data.gouv.fr>) permet d'accéder à la base SIRENE²¹ des entreprises et leurs établissements. Cette base — dont le projet a débuté dans les années 60 — contient aujourd'hui environ 10 millions d'entreprises et 11 millions d'établissements. Pour rappel :

- le numéro *SIREN* (9 chiffres) identifie une *entreprise* française ;
- le *NIC* (5 chiffres) ou *numéro interne de classement*, identifie un *établissement* particulier d'une entreprise ;
- le numéro *SIRET* est la concaténation du siren et du nic. C'est l'*identifiant* d'un établissement (il fera office de clé primaire).

7.7.1 Création de la table des établissements

Le « dessin » proposé pour stocker chaque établissement comporte environ 100 attributs, les différents noms & sigle de l'entreprise, les différents champs d'adresse, des informations sur la taille et l'activité de l'entreprise. Cette énorme table pourrait bien sûr être normalisée pour éviter les redondances mais ça n'est pas le propos ici. Dans sa grande mansuétude, le projet de diffusion de la base SIRENE donne accès à un fichier csv décrivant chacun des champs. En voici un extrait :

```
SIREN;Identifiant de l'entreprise;9;alphanumérique;1
NIC;Numéro interne de classement de l'établissement;5;alphanumérique;2
NUMVOIE;Numéro dans la voie;4;alphanumérique;17
INDREP;Indice de répétition;1;alphanumérique;18
TYPVOIE;Type de voie de localisation de l'établissement;4;alphanumérique;19
```

21. Qui est un acronyme pour *système national d'identification et du répertoire des entreprises et de leurs établissements*.

Approche « full SQL »

Afin de montrer qu'il est possible grâce à SQL d'exécuter dynamiquement des requêtes dont le contenu varie, nous vous proposons ici de créer la table stockant tous les attributs :

```
create table schema_etab (
    chp text,    description text,
    taille int,  type text,
    ordre int);
```

► § 5.6.1
p. 201

La commande de copie de fichier nous permet de peupler cette table avec les données du fichier csv. Nous créons ensuite la table sirene comme une coquille vide :

```
create table sirene();
```

Et pour éviter d'avoir à écrire 100 fois quelque chose comme :

```
alter table sirene add siren varchar(9);
comment on column sirene.siren
    add 'Identifiant_de_l''entreprise';
```

nous allons tenter de faire exécuter ces deux instructions dans une jolie boucle qui sera délicatement insérée dans une routine, dont nous ne vous donnons ici que le corps²² :

```
declare
    C record; -- info sur chaque champ
begin
    for C in (
        select * from schema_etab order by ordre)
    loop
        null; -- pour l'instant on ne fait rien
    end loop;
end
```

À chaque itération de cette boucle, nous avons accès à toutes les informations pour chaque champ de la (future) table sirene. Dans la clause **loop**, nous pourrions par exemple écrire :

```
loop
    raise notice 'champ_:_%s',C.chp;
end loop;
```

22. Nous vous invitons à vous référer au chapitre 6 et en particulier à la section 6.4 page 218 pour avoir tous les détails nécessaires à la création d'une procédure stockée.

Ce qui provoquerait à l'invocation de routine un affichage comme ci-dessous :

```
yahozna=> select add_chp_sirene();
NOTICE: champ : SIRENs
NOTICE: champ : NICs
NOTICE: champ : L1_NORMALISEEs
NOTICE: champ : L2_NORMALISEEs
...
yahozna=>
```

Si l'objectif est maintenant d'ajouter à la table `sirene` chacun des champs sur lesquels on itère, on pourra construire la requête dans une chaîne de caractères (qu'il faudra prendre soin de déclarer), puis demander au SGBD de l'exécuter :

```
loop
  raise notice 'champ_:_%s',C.chp;
  requete:='alter_table_sirene'
    || '_add_' || C.chp ||
    || '_varchar(' || C.taille || ')' ;
  execute requete;
end loop;
```

Approche « script Unix »

À partir du fichier `csv` contenant chacun des champs, il est aussi possible d'utiliser la souplesse des outils Unix pour générer un script SQL de génération des champs. Le script suivant :

```
# on découpe les infos selon les sauts de ligne :
IFS="
"
# pour chaque entrée du fichier csv
for L in $(cat dessin.csv); do
  # nom => champ 1
  chp=$(echo $L | cut -d';' -f1 | sed 's/"//g')
  # taille => champ 3
  size=$(echo $L | cut -d';' -f3 | sed 's/"//g')
  echo "-- champ $chp taille $size"
  echo "alter table sirene add $chp varchar($size);";
done
```

si appelé comme ceci :

```
❏ bash gen_chp_sirene.sh > gen_chp_sirene.sql
```

produira un fichier texte contenant toutes les commandes SQL permettant d'ajouter chaque champ de la base Sirene :

```
-- champ siren taille 9
alter table sirene add siren varchar(9);
-- champ nic taille 5
alter table sirene add nic varchar(5);
-- champ l1_declaree taille 42
alter table sirene add l1_declaree varchar(42);
...
```

Ce fichier pourra bien entendu être exécuté (une fois suffira) grâce à `psql` :

```
❏ psql -f gen_chp_sirene.sql sirene
```

qui dans ce cas (fichier de commande passé en argument) exécutera chaque commande SQL à tour de rôle.

Et la description des champs ?

Vous aurez sans doute remarqué que les commandes :

```
comment on column sirene.nic is 'xxx'
```

ont été superbement ignorées jusqu'à maintenant. Dans la boucle de l'approche « full SQL » ci-dessus (§ 7.7.1 page 318) :

```
loop
    requete:='alter_table_sirene_...';
    execute requete;
end loop;
```

il est maintenant nécessaire d'ajouter la commande **comment** permettant d'ajouter une description à chaque champ. Une façon naïve serait d'insérer une autre requête :

```
requete:='comment_on_column_siren';
||C.chp||'_is_'||C.description;
execute requete;
```


La variable requete contiendrait alors successivement :

```
comment on column sirene_test.SIREN is Identifiant de l'entreprise
comment on column sirene_test.NIC is Numéro interne de classement
comment on column sirene_test.L1_NORMALISEE is Première ligne
...
```

Il manque par conséquent le délimiteur ' autour de la description elle-même. Il faut donc ajouter une apostrophe dans la chaîne, ce qui peut être fait en SQL, *en ajoutant 2 apostrophes*. Le code devient alors :

```
requete:='comment_on_column_siren';
||C.chp||'_is_'||C.description||''';
execute requete;
```

Étant donné que C.description peut lui-même contenir une apostrophe, on doit la remplacer par une double si nécessaire. On aura finalement (avec la boucle) :

```
loop
  desc:=replace('','''',C.description);
  requete:='comment_on_column_siren';
  ||C.chp||'_is_'||desc||''';
  execute requete;
end loop
```

7.7.2 Insertion des données

La table sirene créée, le fichier stock de base peut alors être téléchargé, par exemple celui-ci :

http://files.data.gouv.fr/sirene/sirene_201712_L_M.zip

qui occupera 1,4 giga-octets. Pour le décompresser il vous faudra trouver environ dix fois plus de place. Une fois dézippé, on pourra peupler la table avec la commande **copy** (cf. § 5.6 page 201) :

```
\copy sirene from 'sirene.csv'
with csv delimiter ',' quote '"' header
```

Pour info :

Nombre de lignes dans le fichier csv	10874739
Temps d'insertion des données	8 minutes 30

7.7.3 Indexation

Le premier index sera celui implicitement créé par la clé primaire :

```
alter table sirene add primary key(siren,nic)
```

Pour avoir une idée de ce qu'apporte l'indexation des données sur un cas concret comme celui d'une base d'établissements comportant 11 millions d'enregistrements, nous donnons ici à titre indicatif quelques-uns des temps obtenus :

```
select siren,nic,l1_declaree from sirene
where siren=... and nic=...
```

Un **explain analyze** de cette requête exécutée sur notre système de test (processeur Intel Core I5 et disque dur SATA 6Go/s, 7200tr/mn) donnera (nous indiquons au passage le temps pour peupler la table et pour créer la clé primaire) :

<i>Temps de création de l'index</i>	<i>environ 3 minutes 50</i>
<i>Recherche sans création de clé primaire</i>	<i>environ 1 minute 30</i>
<i>Recherche après création de la clé primaire</i>	<i>0.2 milliseconde</i>

De même, on pourra indexer le nom de l'entreprise afin d'optimiser les recherches qui seront faites via une égalité parfaite :

```
create index idx_l1 on sirene(l1_declaree)
```

Une recherche du type :

```
select ... from sirene
where l1_declaree='MALAWELEKAAHM'
```

donnera :

<i>Temps de création de l'index</i>	<i>un peu plus de 2 minutes</i>
<i>Recherche sans index</i>	<i>autour d'1 minute 30</i>
<i>Recherche avec index</i>	<i>0.05 milliseconde</i>

Enfin, une recherche de type :

```
select ... from sirene
where l1_declaree like 'ABC%'
```

sera catastrophique, sans index spécialisé, comme indiqué au paragraphe 7.3.3 page 305 :

```
create index idx_patops_l1
on sirene(l1_declaree varchar_pattern_ops);
```

Nous aurons alors :

<i>Temps de création de l'index</i>	<i>1 minute 50</i>
<i>Recherche sans index</i>	<i>autour de 2 minutes 30</i>
<i>Recherche avec index</i>	<i>autour de 5 millisecondes</i>

Dans le SGBD PostgreSQL, les index de type varchar pattern ops refusent obstinément de se déclencher lorsque le motif de recherche commence par le caractère « joker » %. On peut s'en rendre compte en demandant via la commande **explain** à visualiser le plan de requête pour une telle requête :

```
yahozna=> explain
yahozna-> select * from sirene
yahozna->         where l1_declaree like '%XYZ'
Seq Scan on sirene (...)
  Filter: ((l1_declaree)::text ~~ '%XYZ'::text)
  Rows Removed by Filter: 10874703
Planning time: 0.164 ms
Execution time: 95252.972 ms
```

Pour ce type de recherche, les index utilisant des trigrammes (PostgreSQL (2017b)) peuvent vous sauver. Il faudra cependant que l'administrateur de votre base se fende d'un :

```
yahozna=# create extension pg_trgm;
CREATE EXTENSION
yahozna=#
```

Une fois cette extension installée, vous pourrez poser un index s'appuyant sur les trigrammes :

```
create index idx_trgm_l1 on sirene
using gin (l1_declaree gin_trgm_ops);
```

L'index ainsi posé (cf. temps de création ci-dessous), ramène ce type de recherche à un temps tout à fait acceptable puisqu'il se rapproche (5 millisecondes) du temps obtenu avec un index classique.

<i>Temps de création de l'index</i>	<i>environ 2 minutes</i>
<i>Recherche avec index</i>	<i>autour de 5 millisecondes</i>

7.7.4 Petit point sur l'espace de stockage

Nous avons montré précédemment (cf. 7.5.3 page 312) qu'il était possible d'obtenir la taille des objets de notre base de données :

```
yahozna=> select
yahozna-> pg_size_pretty(pg_relation_size('sirene'));
pg_size_pretty
-----
6839 MB
(1 row)
```

Pour ce qui est des quatre index posés jusqu'ici :

```
yahozna=> \d sirene
Table "public.sirene"
  Column      |      Type      | Modifiers
-----+-----+-----
siren         | character varying(9) | not null
nic           | character varying(5) | not null
...           | ...              | ...
Indexes:
"sirene_pkey" PRIMARY KEY, btree (siren, nic)
"idx_l1" btree (l1_declaree)
"idx_patops_l1" btree (l1_declaree varchar_pattern_ops)
"idx_trgm_l1" gin (l1_declaree gin_trgm_ops)
```

le volume consommé est :

sirene_pkey	327Mo
idx_l1	419Mo
idx_patops_l1	419Mo
idx_trgm_l1	474Mo

Pour s'amuser un peu avec les tables système, nous pourrions tenter de faire la somme en octets des 100 champs de notre table sirene. Fouinons dans les tables et dans un premier temps, tentons d'extraire le nom et la taille de chacun des champs de celle-ci :

```
select
    attname, atttypid, atttypmod
from pg_attribute
where attrelid='sirene'::regclass
```

Cette commande nous renvoie tous les attributs, leur « identifiant de type », ainsi que leur « modificateur de type » :

attname	atttypid	atttypmod
ctid	27	-1
...
siren	1043	13
nic	1043	9
ll_normalisee	1043	42
...
indrep	1043	5
typvoie	1043	8
libvoie	1043	36
codpos	1043	9
...

Tous les types de données de notre table étant de type varchar on en déduit intuitivement que 1043 désigne ce type et que le type 27 correspond aux attributs système non visibles par la commande \d. Grâce à la documentation de PostgreSQL, on trouvera que le champ atttypid fait référence à la table pg_type. On pourra donc avoir une vision plus lisible grâce à la jointure suivante :

```
select attname , t.typname , atttypmod
  from pg_attribute a
       inner join pg_type t on a.atttypid=t.oid
 where attrelid='sirene'::regclass;
```

qui affichera le type en clair. En tout état de cause :

```
select sum(atttypmod) from pg_attribute
 where attrelid='sirene'::regclass
 and atttypid=1043
```

nous renvoie gentiment la somme de 2047 octets pour chaque entrée de la table sirene. Souvenons-nous que :

```
yahozna=> select count(*) from sirene;
count
-----
10874739
```

La taille maximale de la base, en ne comptant que les données et en supposant que chaque attribut soit complètement rempli, est donc :

10874739×2047 soit 22260590733 octets

c'est-à-dire une bonne vingtaine de gigaoctets qui est à rapprocher des 6,8 gigaoctets qu'occupe notre table sur le disque.

7.7.5 Mise à jour de la base SIRENE

Revenons aux données de notre base SIRENE. Des mises à jour, quotidiennes et mensuelles, sont proposées par le site `data.gouv.fr` sous forme de fichiers csv. Chacun de ces fichiers contient des informations de création, modification, suppression d'établissements. À titre d'exemple, la mise à jour mensuelle du 4 avril 2017²³ :

taille de l'archive	28 megaoctets
taille une fois décompressée	180 megaoctets
nombre d'entrées	environ 230000
créations d'établissements	environ 90000
fermetures d'établissements	environ 60000
modifications d'établissements	environ 35000

Deux mises à jour par an (début et milieu d'année) sont un peu plus volumineuses pour ce qui est des modifications. Ci-dessous à titre d'exemple, la mise à jour mensuelle du 31 décembre 2017 :

taille de l'archive	461 megaoctets
taille une fois décompressée	4 gigaoctets
nombre d'entrées	environ 5 millions
modifications d'établissements	2710819
créations d'établissements	91603
fermetures d'établissements	72992

Dans ce qui suit, nous proposons quelques pistes pour intégrer ces mises à jour dans notre base des établissements. Le schéma des fichiers de mise à jour est exactement celui de la table des établissements elle-même. On pourra donc créer une table temporaire permettant de stocker les mises à jour à effectuer :

```
create table sirene_maj
as select * from sirene limit 0
```

Notez le **limit 0** qui évite de créer la table en dupliquant les 10 millions d'enregistrements de la table `sirene`! En supposant que nous avons rempli la table `sirene_maj` avec un fichier csv de mise à jour, nous pouvons visualiser les informations de son contenu comme ceci :

```
select vmaj, count(siren) from sirene_maj
group by vmaj
```

23. Il y a dans ces fichiers de mise à jour, deux lignes pour chaque modification : une pour l'état initial, l'autre pour l'état final après modification.

requête qui nous renverra :

vmaj	count
F	2710819
C	91603
E	72992
D	1663
O	2763
I	2710802

Le champ vmaj indique le type de mise à jour :

- **C**réation ou **E**ffacement d'établissements;
- état **I**nitial et **F**inal d'une modification;
- **D**iffusion ou **S**ortie de diffusion, pour les établissements portés par une personne physique.

Grâce à ces informations, nous allons créer une routine pour fermer tous les établissements indiqués dans le fichier de mise à jour :

```
create or replace function sirene_maj_fermeture()  
returns void as ...
```

Le corps de cette routine aura pour objet de stocker dans une table `sirene_fermeture` — qu'il faudra bien sûr créer — les établissements qui ferment²⁴ :

```
begin  
  insert into sirene_fermeture  
    select * from sirene_maj where vmaj=etatF;  
exception  
  when unique_violation then null;  
end;
```

puis de supprimer ceux-ci dans la relation `sirene` :

```
for E in select * from sirene_maj  
          where vmaj='F'  
loop  
  delete from sirene  
    where siren=E.siren and nic=E.nic;  
end loop;
```

24. L'exception permet de relancer la routine sans créer de doublon dans la table stockant les établissements fermés.

Finalement, un script lancé une fois en début de mois devrait :

1. déterminer le fichier à télécharger en fonction de la date;
2. télécharger ce fichier;
3. le « dé-zipper »;
4. le convertir en UTF-8 (initialement en ISO-8859);
5. exécuter les fonctions de mise à jour (dont celle définie ci-dessus);
6. effacer les traces du téléchargement.

Let's go! L'url des mises à jour mensuelles est de la forme :

```
http://files.data.gouv.fr/sirene/sirene_201712_M_M.zip
```

Ce fichier estampillé « 2017/12 » est disponible au début du mois suivant, date à laquelle nous lancerons la tâche planifiée (cron sous Unix). Nous nous baserons sur une fonctionnalité de l'utilitaire `date` du projet Gnu intégré dans le langage de commande des systèmes Linux, qui offre la possibilité de demander la date « *x jours avant la date courante* ». Ainsi :

```
date -d "10 days ago"
```

affichera la date d'il y a 10 jours. La commande `date` pouvant extraire les différents attributs, on pourra écrire dans le script de mise à jour :

```
#!/bin/bash
# le mois d'il y a 10 jours
mois=$(date +%m -d "10_days_ago")
# l'annee d'il y a 10 jours
annee=$(date +%Y -d "10_days_ago")
# finalement le nom du fichier
fic=sirene_${annee}${mois}_M_M.zip
# et l'url
urlmaj=http://files.data.gouv.fr/sirene/$fic
```

Pour le téléchargement lui-même on fera appel à la commande `wget` :

```
wget -nv $urlmaj -O maj.zip
```

qui sauvegardera le téléchargement dans le fichier `maj.zip`. On décompactera ce fichier :


```
unzip -p maj.zip > maj.tmp  
rm -f maj.zip # on efface l'archive
```

Le contenu de l'archive est encodé en ISO8859, on le convertira donc en UTF-8 afin de s'aligner sur l'encodage de notre base :

```
iconv -f ISO885915 -t utf8 maj.tmp > maj.csv  
rm -f maj.tmp
```

La suite du traitement sera faite depuis la base via un script contenant des commandes SQL :

```
psql -f maj-sirene.sql sirene
```

On pourra ensuite faire le ménage définitif :

```
rm -f maj.csv
```

Le script maj-sirene.sql contiendra quelque chose comme²⁵ :

```
delete from sirene_maj; -- petit nettoyage  
\copy sirene_maj from 'maj.csv'  
    with csv delimiter ';' quote '"' header  
select maj_siren_fermeture();  
...
```



Les utilisateurs avertis pourraient à juste titre s'interroger sur le bien-fondé de passer par tous ces fichiers temporaires. L'utilisation de pipes Unix permettrait de télécharger, dézipper et convertir à la volée :

```
wget ... | unzip ... | iconv ... > maj.csv
```

Sauf qu'une telle construction, si elle est élégante, échoue lamentablement, faute de mémoire, sur des très gros fichiers.

Finalement, il nous reste à déclencher cette mise à jour automatiquement en début de mois. Ceci peut être confié à l'utilitaire cron installé sur n'importe quel système Unix. Nous laissons le lecteur le soin de s'y atteler en s'aidant si nécessaire de Lozano (2010).

25. **Attention**, comme indiqué au paragraphe 5.6.1 page 201, la ligne de la commande `\copy` — ici scindée en deux *pour des raisons de mise en page* — doit apparaître sur une seule ligne.

7.7.6 Un peu de ménage

Pour terminer les aventures de la petite sirène, nous attirons ici l'attention du lecteur sur quelques considérations autour de l'effacement des données. Examinons la taille de notre fichier de mise à jour :

```
yahozna=> select pg_relation_size('sirene_maj');
pg_relation_size
-----
154501120
```

Effaçons les données et examinons à nouveau la taille :

```
yahozna=> delete from sirene_maj ;
DELETE 231154
yahozna=> select pg_relation_size('sirene_maj');
pg_relation_size
-----
154501120
```

Surprise, les données de la table occupent toujours le disque ! Ceci est un comportement normal, et si votre SGBD est correctement configuré, un service devrait se charger de faire le ménage au bout de « quelques temps » :

```
yahozna=> select pg_relation_size('sirene_maj');
pg_relation_size
-----
0
```

Dans PostgreSQL le service en question s'appelle autovacuum. Si vous êtes pressés pour le nettoyage, vous pouvez toujours :

1. soit utiliser la commande **truncate** :

```
truncate sirene_maj
```

qui supprimera les données et libérera également le disque ;

2. soit forcer l'exécution du « ramasse-miettes » après le **delete** :

```
vacuum full verbose sirene_maj
```

qui devrait vous afficher quelque chose du genre :

```
INFO: vacuuming "public.sirene_maj"  
INFO: found 231154 removable in 18860 pages  
CPU 0.01s/0.06u sec elapsed 0.07 sec.
```

Le livre s'achève sur une commande qui fait le vide...
There is no dark side in the moon, really. As a matter of fact it's all dark.



- A.1 Prérequis
- A.2 Les sources du manuel
- A.3 Compilation
- A.4 Imprimer
- A.5 Nettoyage
- A.6 Chapitrage

A

Notes de production

NOUS AVONS RASSEMBLÉ ici des éléments permettant d'exploiter les sources de ce document, comment les compiler, avec quelle distribution de \LaTeX , comment les fichiers sont organisés, etc.

A.1 Prérequis

Le présent ouvrage a été compilé sur GNU/Linux Ubuntu 16.04 et 18.04 avec les versions T_EXlive 2015 et 2017, respectivement. Il ne devrait pas y avoir de problème particulier pour le compiler avec d'autres distribution. Les prérequis sont donc :

- Une distribution T_EX, les paquets debian suivants devraient suffire :

texlive-base	texlive-fonts-recommended
texlive-binaries	texlive-lang-french
texlive-extra-utils	texlive-latex-base
texlive-font-utils	texlive-latex-extra
texlive-fonts-extra	texlive-pictures
texlive-science	texlive-latex-recommended
texlive-generic-recommended	
- graph de la suite plotutils
- transfig (fait partie de xfig)
- inkscape
- gs (ghostscript)
- ps2epsi (ghostscript)
- epstopdf (doit faire partie de la distribution T_EX, sinon allez faire un saut sur la page <https://ctan.org/pkg/epstopdf>)
- pdfnup (idem, <https://ctan.org/pkg/pdftjam>)

A.2 Les sources du manuel

A

Les sources sont disponibles sur le GitLab de l'association Framasoft :

```
git clone git@framagit.org:framabook/onlysql.git
```

Et :

```
git pull
```

pour les mettre à jour une fois que vous aurez une copie en local chez vous.

A.2.1 Structure

Les sources du manuel sont organisées selon le principe suivant :

- document maître dans le fichier `framabook.tex`
- sources \LaTeX des chapitres dans le répertoire `corps` avec un fichier par chapitre;
- les styles (`sty` et `cls`) dans le répertoire `styles`;
- les images dans un répertoire `pngs`;
- les sources `xfig` et `inkscape` dans le répertoire `figs`;
- tout ce qui a trait à l'index, à la bibliographie et au glossaire est stocké dans le répertoire `bibidx`;

Les sources des dessins (`.fig` et `.svg`) sont traduits au format pdf dans le répertoire `pdfs`.

A.2.2 Styles

Le fichier `framabook.cls` contient la définition de la classe du manuel. Ce fichier fait appel à une série de packages « du commerce » et une série de packages « maison ». On trouve, pour ces derniers, un fichier source pour :

- boîte avec un titre (`titlebox.sty`), onglets, nota, sommaire, glossaire, renvois (`voir.sty`), code source, commandes unix, lettrine, citations et épigraphes;
- le sommaire;
- la géométrie globale du document;
- l'allure des en-têtes et pieds de pages;
- l'allure des sections/chapitres/etc.;
- des commandes en vrac utilisées dans le document (dans le fichier `manumac.sty`);

Sauf indication contraire, ces fichiers portent un nom ressemblant étrangement à ce qu'ils contiennent.

A

A.3 Compilation

1. Choisir la version à compiler, pour cela dans le document maître `framabook.tex`, choisir l'une des deux options de documents :
 - soit `versionenligne` qui produit un pdf uniquement destiné à être lu
 - soit `versionpapier` pour un pdf destiné à être imprimé

2. Choisir la version couleur ou passer en noir & blanc, dans ce cas ajouter nb dans les options de classe
3. Créer les figures :
 - soit :


```
└─ make mode=nb figs
```
 - soit :


```
└─ make figs
```
4. \LaTeX Passe 1 :


```
└─ pdflatex framabook
```
5. Création de la biblio et de l'index


```
└─ make bib
   └─ make index
```
6. \LaTeX passe 2 : `pdflatex framabook`
7. \LaTeX passe 3 : `pdflatex framabook`

À l'issue de cette séquence de commandes un fichier `framabook.pdf` doit contenir le manuel.



Si nécessaire, un script bash peut produire les trois versions d'un coup de cuillère à pot. Pour cela, il faudra taper :

```
└─ ./makedistrib.sh
```

Les trois fichiers produits seront nommés :

1. `onlysql-nb.pdf` version noir&blanc à imprimer ;
2. `onlysql-couleur.pdf` version couleur à imprimer ;
3. `onlysql-enligne.pdf` version à destination d'un lecteur PDF.

A

A.4 Imprimer

Pour fabriquer le fichier `book.pdf` qui pourra être imprimé sur du A4 avec une reliure longue et 2 pages A5 par page :

```
└─ make book
```

Cette commande exploite le fichier `framabook.pdf` de la section précédente. Si vous vouliez transformer l'un des fichiers produit par le script `makedistrib.sh`, il faudra vous fendre d'un :

```
└─ make MAITRE=onlysql-couleur book
```


A.5 Nettoyage

Pour détruire toutes les figures générées en pdf à partir des sources xfig/svg :

```
| make cleanfigs
```

Pour effacer tous les fichiers temporaires de L^AT_EX :

```
| make cleantex
```

Pour effacer tous les pdf (document maître et chapitres) :

```
| make cleandocs
```

A.6 Chapitrage

Pour créer un fichier pdf en « 2 pages par page » par chapitre :

```
| ./makechaps.sh --2up
```

L'option --2up est facultative.



Bibliographie

Théorème de brewer. https://fr.wikipedia.org/wiki/Théorème_CAP.

Stéphane BORZMEYER : Taille des bases postgresql. <https://www.bortzmeyer.org/postgresql-taille-bases.html>.

Frédéric BOULANGER, Henri DELEBECQUE et Gianluca QUERCINI : Cours d'architecture des ordinateurs. <http://http://wdi.supelec.fr/architecture/Info/BinArith>, 2017.

Tedd CODD : Derivability, redundancy, and consistency of relations stored in large data banks. Rapport technique, IBM, 1969.

Tedd CODD : A relational model of data for large shared data banks. *CACM*, 13(6):377–387, june 1970.

Oracle CORPORATION : Documentation oracle. <http://docs.oracle.com/>, 2012.

J.-C. GRATAROLA : Administration d'une base de données oracle 7, 1997. Université Nice - Sophia Antipolis.

Rick GREENWALD, Robert STACKOWIAK et Jonathan STERN : *Oracle Essentials*. O'Reilly, 2013.

Jean-Philippe GUILLOUX : Initiation à l'optimisation de requêtes sql sous oracle. <https://jpg.developpez.com/oracle/tuning/>, 2008.

Tom KYTE : <http://asktom.oracle.com/>, 1993–2012.

Vincent LOZANO : *Tout ce que vous avez toujours voulu savoir sur Unix sans jamais oser le demander*. Framabook, 2010. ISBN 978-2-35922-023-0. <http://www.framabook.org/unix>.

Jean-Patric MATHERON : *Comprendre Merise*. Eyrolles, 1999.

S. M. MIRANDA et J. M. BUSTA : *L'art des bases de données*. Eyrolles, 2^e édition revue et corrigée édition, 1990.

L. NAVARRO : *Optimisation des bases de données*, chapitre chap. 6 Techniques d'optimisation standard des requêtes. Pearson Education France, 2010.

Documentation PostgreSQL : *Conseils sur les performances*, chapitre 14. PostgreSQLFr, 2017a.

Documentation PostgreSQL : *Module PostgreSQL trigramme*, chapitre F.35. PostgreSQLFr, 2017b.

D. SZALKOWSKI : *Support formation Administration Oracle*. www.dsfc.net, 2006.

Markus WINAND : Use the index, luke. <https://use-the-index-luke.com/>.

Étienne GEORGES : *Base de données Ortems sous Oracle, principes et détails*. Finmatica, 2002.

Glossaire

ACID

Acronyme de « atomicité, cohérence, isolation & durabilité » décrit les propriétés que doit respecter un SGBD pour garantir qu'une transaction soit réalisée de manière fiable, c'est-à-dire sans perte de la cohérence et de l'intégrité des données.

Agrégation

Une agrégation est une requête **select** fusionnant une table en plusieurs sous-ensembles ayant des propriétés communes. Chaque sous-ensemble fait alors l'objet d'un ou plusieurs calculs statistiques (comptage, moyenne, min, max, etc.)

Arbre

Les arbres en informatique sont des structures de données permettant d'accélérer la recherche d'information. Une fois les données stockées sous forme arborescente, la recherche est bien plus efficace qu'une recherche séquentielle (en examinant les éléments les uns après les autres).

Association

Dans le contexte d'un MCD, une association représente un lien entre deux concepts émergeant du système d'information étudié.

Atomicité

C'est l'un des mécanismes les plus importants des SGBD, il assure que soit l'intégralité des requêtes qui compose une transaction est exécutée, soit aucune d'entre elles.

Cardinalités

Dans le contexte d'un MCD formalisé grâce à des entités et des associations, les cardinalités précisent comment sont associées les entités, pour ce qui est du nombre d'éléments de chaque ensemble impliqué. Elles sont indispensables pour la conception du modèle relationnel.

Clé étrangère

Une clé étrangère est une contrainte d'intégrité permettant de matérialiser une association entre deux entités. Elle permet de garantir que deux tables seront reliées par un (ou plusieurs) attributs et ce de manière cohérente (la suppression, l'insertion et la mise à jour seront contrôlées par le SGBD).

Clé de substitution

Appelée également clé artificielle (et *surrogate key* en anglais), c'est une clé fabriquée de toute pièce sous la forme d'un nombre entier. Elle ne porte aucune information, et son seul objectif consiste à identifier un tuple. Il n'est pas rare d'y avoir recours malgré la présence de clé naturelle dans la relation.

Clé primaire

Parmi les clés candidates, c'est-à-dire parmi tous les attributs ou groupes d'attributs pouvant faire office de clé, la clé primaire est celle qui sera préférée pour identifier un tuple et par conséquent faire les liens entre les autres relations via les clés étrangères.

Clé

Une clé est un (ou plusieurs) attribut qui permet d'identifier un élément (un tuple) de la relation de manière univoque. La connaissance de la valeur clé permet de désigner un tuple et un seul dans la relation.

Complexité

Étudier la complexité d'un algorithme c'est expliciter dans quelle mesure évoluera son temps d'exécution lorsqu'on augmentera le nombre des données à traiter. Cette étude sera faite en précisant les opérations critiques à mesurer (par exemple la comparaison de valeurs dans le cadre d'un tri). Son résultat donnera une idée des performances d'un algorithme relativement un autre et ce, indépendamment de la puissance de calcul de l'ordinateur qui l'exécutera.

Contrainte d'intégrité

Une contrainte d'intégrité est un garde-fou posé par le concepteur de la base sur un (ou plusieurs) attribut, pour garantir une certaine cohérence dans les données (unicité d'un attribut, présence d'une valeur obligatoire, valeur maximale, etc.). La contrainte peut être posée uniquement si les données la respecte, et le SGBD fera échouer toute requête tentant de la violer.

Entité

Dans le contexte d'un MCD, une entité représente un concept qui émerge naturellement dans le système d'information étudié. Elle désigne toujours un ensemble d'éléments au sens mathématique du terme.

Float

Terme utilisé pour désigner une technique de stockage des nombres « à virgule ». Les flottants sont caractérisés par une précision (pas de représentation exacte des nombres) contrebalancé par une rapidité des opérations arithmétiques dans lesquels ils sont impliqués.

Forme normale

Ce sont des règles qui permettent de vérifier la cohérence d'un modèle relationnel. Elles sont utiles pour y détecter de manière systématique les redondances.

Index

Un index est une structure qui vient en complément d'une table, dont le but est d'accélérer les recherches. Les index

sont posés sur un (ou plusieurs) attribut d'une table, le SGBD étant en charge de les mettre à jour en cas de modification des données.

Jointure

Une jointure est une requête **select** réalisant le « rapprochement » de deux tables, généralement selon un attribut commun. Le modèle relationnel ayant tendance à éclater les données dans de multiples tables, les opérations de jointures sont très fréquentes et donnent souvent lieu à des vues.

LDD

Langage de description de données : c'est le volet de SQL qui rassemble toutes les fonctions permettant d'agir sur la structure des relations (colonnes, contraintes d'intégrité, etc.).

LMD

Langage de manipulation de données : c'est le volet de SQL qui rassemble toutes les fonctions permettant d'agir sur les données contenues dans les relations (ajout, suppression, mise-à-jour, extraction).

MCD

Modèle conceptuel des données : le « plan » des données qui seront stockées dans la base. Ce modèle peut faire appel plusieurs formalismes, celui choisi dans cet ouvrage est le formalisme *entité/association*

NULL

C'est le mot réservé pour désigner le vide ou l'absence de valeur dans les bases de données relationnelles. Les SGBD mettent en œuvre tous les mécanismes pour traiter le vide comme il se doit, dans les expressions arithmétiques, booléennes, etc.

Normalisation

Dans le contexte du modèle relationnel, les opérations de normalisation consistent à éclater une relation en plusieurs, pour aboutir à un modèle limitant au maximum des redondances, et donc à garantir l'intégrité des données.

Procédure

Les procédures stockées dans les bases de données sont des traitements nommés qui peuvent être déclenchés explicitement par un programme. Ces procédures ont pour but d'exécuter des requêtes — la plupart du temps du volet LMD — à partir de paramètres qu'elles reçoivent en entrée.

Produit cartésien

C'est une opération mathématique sur deux ensembles, que l'on rencontre à différentes reprises dans le monde SQL. Elle consiste à envisager toutes les combinaisons de couples d'éléments pris dans chacun des deux ensembles impliqués dans le produit. On retrouve le produit cartésien derrière la notion de jointure.

Requête

C'est le terme générique — correspondant au Q de l'acronyme SQL, *query* en anglais — pour désigner un ordre passé au SGBD. Une requête peut modifier les données ou la structure d'une table, changer les droits des utilisateurs, retourner des valeurs, etc. En définitive, tout ordre destiné à un SGBD peut-être exprimé via une requête SQL.

Restauration

Restaurer une base c'est la remettre dans un état donné à partir de fichiers enregistrés dans un format qui dépend du SGBD.

SGBD

Système de gestion de bases de données : désigne les différents logiciels, libres ou propriétaires, permettant de gérer des bases de données. Quelques « stars » : Oracle, Microsoft Sql Server, PostgreSQL, ...

SQL

Structured Query Language : langage de requête normalisé pour manipuler une base de données et ce qu'elle contient. Tout ordre destiné à un SGBD peut-être exprimé dans le langage SQL.

Séquence

Une séquence est un objet spécial des SGBD Oracle et PostgreSQL dont le but est d'aider à la génération automatique des clés de substitutions.

Sauvegarde

Sauvegarder une base c'est enregistrer son état à un instant t dans un format qui dépend du SGBD.

Système d'information

On appelle système d'information (SI), un ensemble de données et de traitements dont on a défini le périmètre (limité à une tâche spécifique, un service, une entreprise, etc.). Les données et les traitements du SI étudié, ne sont pas nécessairement numérisés, modélisés ou automatisés. Le rôle du SGBD et des applications qui gravitent autour est justement d'automatiser certaines tâches du SI.

Transaction

Un ensemble de requêtes élémentaires permettant de changer l'état de la base de données, le tout de manière cohérente.

Trigger

Un trigger est un traitement stocké dans la base et associé à une table ; il se déclenchera automatiquement lorsque des opérations de type LMD seront exécutées sur la table. L'exécution d'un **delete** sur une table dotée d'un trigger déclenchera par exemple l'exécution d'un traitement de sauvegarde dans une autre table.

UTF-8

Universal character set Transformation Format est un format de stockage dont l'objet est l'encodage des caractères initialement au format Unicode. Cet encodage garantit une relative optimisation de la taille ainsi qu'une transmission aisée sur les réseaux de communication.

Unicode

Projet démarré 1991 dont le but est de recenser, nommer et codifier tous les caractères de toutes les langues du monde.

Vue

Une vue (*view*) est une requête **select** stockée et nommée, que l'on peut alors manipuler comme une table avec quelques limitations. Certains SGBD comme Oracle et PostgreSQL permettent de stocker également les données générées, ces vues sont alors appelées *vues matérialisées*.



Index

Caractères spéciaux

:= 216
:: 166
% 153
_ 153
|| 155, 173

A

absence de valeur 109
accès concurrents .. 258–273
 atomicité . 176, 261, 273
 exceptions 258
 isolation 263
 séquences 259
 transaction 262
 verrouillage 265
ACID 13
age 161
agrégations 186–194
 avg 188
 count 188
 de chaînes 193
 et jointures 190

 et tri 194
 et vide 192
 filter avant/après ... 192
 fonctions 188
 group by 187
 having 192
 max 188
 NULL 192
alter table 98
 add column 105
 add constraint 101, 106
 check 119
 disable trigger 250
 drop constraint 102
 drop not null 118
 set default 114, 118
 set not null 118
ambiguïté sur les attributs 181
application 6
 (dé)connexion 273
 injection Sql 275
 PHP 273
 routines Sql 276
arbre binaire
 AVL 62

- complexité 61
 - de recherche 58
 - et index 300
 - insertion 60
 - arbres B 63
 - arguments
 - fonctions 219
 - in** 219
 - out** 231, 278
 - procédures 219
 - tableau 276
 - arrondir 224
 - as** 181
 - association 68, 70
 - 1-n 104
 - attributs 73
 - cardinalités 74–75
 - de 1 à plusieurs 104
 - de plus. à plusieurs .125
 - ou entité 77
 - réflexive 73, 130
 - ternaire 72, 128
 - association 1-n
 - MCD→MR 104, 123
 - association de 1 à plusieurs
 - MCD→MR 104, 123
 - association n-n
 - MCD→MR 125, 127
 - association réflexive 73
 - passage MCD/MR .. 130
 - association ternaire 72
 - MCD→MR 128
 - atomicité .. 13, 176, 214, 261, 273
 - des attributs 99
 - attributs
 - ajouter 98
 - ambigus 181
 - association 73
 - atomiques 99
 - d'une entité 70
 - d'une relation 98
 - description 320
 - domaine 99
 - ou entités 76
 - préfixer 99
 - type 99
 - avg** 188
- B**
- B-arbres 63, 300
 - base
 - 2 22
 - 10 22
 - 16 22
 - conversion ← 10 24
 - conversion → 10 23
 - conversion → 16 25
 - base de numération 22
 - base SIRENE 317
 - begin** 262
 - blob 141, 202
 - boucle pour 224, 236
 - btree 300
 - bytea** 141, 202
- C**
- caractères
 - codage 34–40
 - text** 104
 - varchar** 98
 - cardinalités 74–75
 - Chamberlain D. 8, 9
 - chaînes
 - comparaison floue .. 169
 - recherche 169
 - check** 119
 - chr** 223
 - clé 112
 - candidate 112

de substitution 102, 112, 116
 génération par séquence 113
 primaire 100–102
 étrangère 106–109
 clé de substitution 102
 clé primaire 100–102
 et index 299
 nommer 102
 supprimer 102
 clé étrangère 106–109
 et mise à jour 107
 et suppression 107
 schéma 109
coalesce 162, 190
 codage
 caractères 34–40
 entiers 25–27
 dépassement, 26
 entiers non signés ... 26
 entiers signés 27
 et architecture 25
 et représentation ... 27
 flottants 28–34
 Codd 8, 96
 cohérence 14
 cohérences des données . 142
 commandes psql ... 282, 284
comment 99, 320
commit 262, 272
 comparaison floue 169
 complexité
 constante 42
 d'un algorithme 40
 en moyenne 42
 linéaire 42
 opération fondamentale
 41
 pire des cas 42
 quadratique 42

concaténation 155, 173
 contrainte d'intégrité 96, 102
 check 119
 créer un domaine .. 123
 domaine 120
 déférables 266
 expr. régulières 121
 liste 310
 non vide 117
 par défaut 118
 reporter 266
 référentielle 106
 contraintes d'intégrité .. 113
 conversion \leftarrow 10 24
 conversion \rightarrow 10 23
 conversion \rightarrow 16 25
 \copy 201, 321, 329
cos 155
count 188, 189, 198, 199
create
 rule 211
 sequence 114
 table 98
 trigger 242
 view 208
 cron 328, 329
cross join 163, 178
 CSV 201
current_date 161
currval 260
 curseur 226–228

D

date 135
 affichage 136, 137
 chaîne de caractère? 135
 chevauchement 164, 168
 comparaison 164
 current_date 161
 daylight saving 141
 format ISO 138

- fuseaux horaires138
 - heures été/hiver141
 - intervalle244
 - manipuler en SQL .. 159
 - overlaps**164
 - saisie136, 138
 - soustraction ... 159, 244
 - stockage interne ... 135, 136
 - date328
 - default**114
 - delete**171, 173
 - sur une vue210
 - trigger246
 - desc**158
 - dichotomique (recherche) 54
 - difference**197
 - distinction min./majuscules 99
 - domaine
 - contraint d'intégrité 120
 - création123
 - des attributs99
 - droits288–297
 - durabilité14
 - dédoubler170
 - dépendance fonctionnelle 143
 - détruire des tuples173
- E**
- Ellison L.9
 - else**217
 - ensemble
 - entité68
 - entiers
 - codage25–27
 - dépassement26
 - valeur maximale26
 - entiers non signés
 - codage26
 - entiers signés
 - codage27
 - entité68
 - attributs70
 - ensemble68
 - ou association77
 - ou attribut76
 - type70
 - epstopdf334
 - ERP2
 - exceptions228–231
 - accès concurrents .. 258
 - curseurs230
 - maison231, 249
 - no_data_found** . 230, 252
 - notice**217
 - pas de valeur230
 - too_many_rows**230
 - trop de tuples230
 - violation clé étrangère 229
 - violation unicité ... 228, 327
 - execute**319
 - exists**199
 - explain**238, 298
 - analyze**238
 - exporter des données ... 201
 - expression régulière
 - dans une contrainte 121
- F**
- fichiers
 - de commandes Sql . 286
 - en python202–205
 - exporter201
 - externes253
 - importer201
 - sortie psql285
 - stockage . 141, 202, 203, 205
 - floor**224
 - flottants
 - codage28–34

IEEE-754 29
 infini 30
 NaN 31
 précision 31
 zéro 30
 étendue 31
 fonctions
 age 161
 aléatoire 223
 appel 219
 argument tableau .. 276
 arguments 219
 avg 188
 chr 223
 coalesce 162, 190
 corps 220
 cos 155
 count 188
 floor 224
 greatest 168
 initcap 242
 least 168
 length 155
 levenshtein 169
 lower 154
 m.-au-pt dans **pgsql** 215
 max 188
 now 161, 244
 overlaps 164, 168
 pourquoi en écrire .. 214
 random 223
 soundex 169
 string_agg 193
 surcharge 225
 to_char 137, 160
 to_date 138
 trim 242
 upper 154, 172, 173
 valeur renvoyée 221
for 224, 236
foreign key 106–109

on delete 107
 on update 108
 formalisme entité/association
 68
 formes normales ... 142–144
full outer join 183
 fuseaux horaires 138

G

graph 334
greatest 168
group by 187
 gs 334

H

hachage (recherche) 55
having 192, 199
 heures d'été/d'hiver 141

I

I18n 306–308
 IBM System R 9
 identifier un tuple . 100, 112
 IEEE-754 29
if 217
ilike 154
 image (stockage) 141
 importer des données ... 201
in 195
 index 298–306, 322–323
 champ non unique . 302
 et clé primaire 299
 taille 305
 infini (flottants) 30
 INGRES 8
initcap 242
 injection **Sql** 275
 inkscape 334, 335
inner join 179
insert 171, 174, 220

dans une vue 211
select 175
 trigger 241
values 174
 Internationalisation 306–308
intersection 196
 intégrité référentielle ... 106
 ISO
 3166 (pays) 105
 8601 (date) 138
 isolation 263

J

jointures 177–186
 avec plus de 2 tables 184
 clause **on** 180
 cross join 163, 178
 d'une table avec elle-même
 185
 et agrégations 190
 et le vide 180
 externe 181–183
 full outer join 183
 inner join 179
 interne 179
 left outer join 181, 191
 NULL 180
 produit cartésien ... 163,
 178
 right outer join ... 182

L

langage procédural (SQL) 215
 langues 306–308
least 168
left outer join 181
length 155
levenshtein 169
like 153
limit 170

logiciels connexes

cron 328, 329
 date 328
 epstopdf 334
 graph 334
 gs 334
 inkscape 334, 335
 pdfnup 334
 pg_dump 315, 316
 pgadmin 282
 plotutils 334
 ps2epsi 334
 psql 215, 287
 transfig 334
 wget 328
 xfig 335
 logique ternaire 110
loop 224, 236
lower 154

M

max 188, 198
 MCD 68
 exemples 80, 84
 passages au MR 104, 123,
 125, 127, 128
 étude de cas 84–93
 MCD→MR 104
 association 1-n 104, 123
 association n-n 125, 127
 association ternaire .128
 minuscules/majuscules .. 99
 modèle
 conceptuel des données
 68
 relationnel 96
 modèle relationnel
 normalisation 142
 relation 97
 tuples 100
 MySql 3

N

NaN 31
NEW 241, 243
nextval 114, 260
no_data_found 230, 252
non vide 117
normalisation 142
 1FN 143
 2FN 143
 3FN 144
 exemples 144
NoSql 14
now 161, 244
NULL 109
 coalesce 162, 190
 et agrégation 192
 jointures 180
 logique ternaire 110

O

oci_bind_by_name 275
offset 170
OLD 243, 244, 246
on 180
opérateurs
 := 216
 :: 166
 || 155, 173
 and 156
 booléens 156
 concaténation . 155, 173
 ilike 154
 in 195
 like 153
 or 156
 priorité 156
Oracle 3
out 231, 278
overlaps 164, 168

P

passage MCD/MR
 association réflexive 130
pays
 code ISO 105
pdfnup 334
perform 219
pg_dump 315, 316
pgadmin 282
pg_connect 274
pg_fetch 274
pg_query 274
pg_query_param 275
PHP 273
 argument tableau .. 276
 injection Sql 275
 requêtes 274
 routines Sql 276
planificateur 298, 303
plotutils 334
PostgreSQL 4
 Histoire 11
primary key 100–102
priorité des opérateurs .. 156
procédures 218–241
 appel 219
 argument tableau .. 276
 arguments 219
 autre langage 253
 corps 220
 m.-au-pt dans pgSql 215
 pourquoi en écrire .. 214
 Python 254–257
prompt psql 282, 287
précision des calculs 236
préfixer les attributs 99
ps2epsi 334
psql 282–288
 commandes 284
 fichier Sql 286

HTML 285
 personnalisation ... 287
 prompt 282, 287
 sortie dans fichier .. 285
 psql 215, 287
 Python
 bytea 203
 fichiers externes 253
 procédures 254–257
 sécurité 257
 tas d'octets 203
 précision des flottants 31

Q

query planer 298, 303
 quick sort 48

R

raise
 exception 231, 249
 notice 217
random 223
 recherche
 arbre binaire 58
 chaînes de caractères 169
 dichotomique 54
 séquentielle 52
 table de hachage 55
record 217
 relation
 attributs 98
 création 98
 définition 97
 ensemble 97, 158
 liste 309
 structure 98
 taille 312, 324
 tuples 100
 requêtes imbriquées 195,
 197–200

restauration 314–316
return 221
returns 221
 trigger 243
right outer join 182
rollback 262, 272
ROWTYPE 217

S

sauvegarde 314–316
 script Sql 320
select
 clause where 151
 cross join 163, 178
 difference 197
 distinct 170
 dédoublonner 170
 exists 199
 full outer join 183
 group by 187
 imbriqués 195, 197–200
 inner join 179
 intersection 196
 into 217
 left outer join 181
 limit 170
 offset 170
 order by 158
 right outer join .. 182
 rownum 170
 sans relation 157
 simple 150
 tri décroissant 158
 trier 158
 union 195
SEQUEL 9
 séquences 113
 accès concurrents .. 259
 currval 221, 260
 dans une procédure 218
 nextval 114, 260

serial 116
setval 115
serial 116
set default 114
setof 234
setval 115
SIG 2
SIREN 317
SIRET 317
soundex 169
 soustraction de dates 159
split_part 155
Sql
 affectation 216
 boucle pour ... 224, 236
 comment 320
 droits 288–297
 else 217
 execute 319
 explain 298
 Historique 7
 if 217
 index 298–306
 injection 275
 langage procédural . 215
 min./majuscules 99
 truncate 331
 vacuum 331
 variables 215
Sql Server 3
 surcharge de fonctions .. 225
 surrogate key . 102, 112, 116
System R 9

T

tableur 12
 taille d'une relation 312, 324
 tas d'octets 141, 202
text 104
timestamp 138, 244
to_char 137, 160

to_date 138
too_many_rows 230
 traduction 306–308
 traitements
 accès concurrents .. 258
 procédures 214, 218
 Python 254
 triggers 241
 vues 208
 transactions ... 262, 269–273
 atomicité 261, 273
 begin 262
 commit 262, 272
 contraintes 266
 isolation 263
 rollback 262, 272
 verrouillage 265
 transfig 334
 transtypage 166
tri
 dans une agrégation 194
 décroissant 158
 par insertion 44
 rapide 48
 select 158
 Shell 46
 trigger 241–253
 (dés)activer 250
 appels récursifs 250
 delete 246
 insert 241
 niveau table 247
 ordre exécution 251
 update 243
trim 242
 tronquer 224
truncate 331
 tuples 100
 identification 112
 NEW 241, 243
 OLD 243, 244, 246

type

bytea 141
double precision ... 236
 espace mémoire 238
numeric 236
record 217
setof 234
timestamp 138, 244

type des attributs 99

types

serial 116

U

union 195
unique 113
update 171
 trigger 243
upper 154, 172, 173

V

vacuum 331
 valeur par défaut 118
varchar 98
 verrouillage 265
 vide 109
 coalesce 162, 190
 jointures 180
 logique ternaire 110
 vues 208–213
 liste 309
 vues matérialisées 212

W

wget 328
where 151, 171
 wiki 2

X

xfig 335

Z

zéro (flottants) 30

Table des matières

1	Ligne directrice	1
1.1	Vous avez dit « base de données »?	2
1.2	Informatiser un système d'information	4
1.2.1	C'est quoi d'abord un SI?	4
1.2.2	Modéliser	5
1.2.3	Concevoir la base	5
1.2.4	Développer une application	6
1.3	Par quel bout on le prend, ce SGBD?	6
1.4	SQL, une séquelle de SEQUEL	7
1.5	Et un tableur, plutôt? non?	12
1.6	Les SGBD : un milieu plutôt « acid »	13
1.7	Le titre du livre et « NoSQL »?	14
1.8	Comment lire ce livre?	15
1.9	Kit de démarrage	17
1.9.1	Installation	17
1.9.2	Configuration minimale	17
1.9.3	Psql : la console SQL ¹	18
2	Principes fondamentaux	21
2.1	Avant-propos	22
2.1.1	Base, nombre & chiffre	22
2.1.2	Conversion d'une base B vers la base 10	23
2.1.3	Conversion de la base 10 vers une base B	24
2.1.4	Conversion de la base 2 vers la base 16	25
2.2	Codage des entiers	25

2.2.1	Entiers non signés	26
2.2.2	Entiers signés	27
2.3	Nombres à virgule flottante	28
2.3.1	Remarque préliminaire	28
2.3.2	Norme IEEE-754	29
2.3.3	Précision et étendue	31
2.3.4	Calcul d'un flottant IEEE-754	31
2.3.5	Y-aurait-il moyen de calculer « juste »	33
2.4	Codage des caractères	34
2.4.1	La table ASCII (1964)	35
2.4.2	ISO 8859 ou le début de la normalisation	35
2.4.3	Vers l'uniformisation : Unicode et ISO 10646	36
2.4.4	UTF-8 : l'encodage émergent	37
2.4.5	J'Ã©cris en UTF-8, du moins j'essaie...	39
2.5	Performances d'un algorithme	40
2.5.1	Notion d'opération fondamentale	41
2.5.2	Ordre d'idée de l'évolution des temps	42
2.6	Trier	44
2.6.1	Tri par insertion : un mauvais élève	44
2.6.2	Tri Shell	46
2.6.3	Tri rapide	48
2.6.4	Pour en finir avec les algorithmes performants	51
2.7	Chercher	51
2.7.1	Chercher dans une séquence	52
2.7.2	Recherche par table de hachage	55
2.7.3	Recherche par arbre binaire	58
2.7.4	Autres arbres	62
2.8	En résumé, à quoi ça sert, tout ça ?	65
3	Modèle conceptuel des données	67
3.1	Qu'est-ce qu'une donnée ?	68
3.2	Modèle conceptuel des données	68
3.2.1	Entité	70
3.2.2	Association	70
3.2.3	Cardinalités	74
3.3	Difficultés rencontrées	75
3.3.1	Sémantique	75
3.3.2	Attribut ou entité	76
3.3.3	Entité ou association	77
3.4	Micro-études de cas	80
3.4.1	Pizzas	80
3.4.2	Primes	80
3.4.3	Étudiants & enseignants	82
3.4.4	Emploi du temps d'un étudiant/enseignant	83
3.5	Étude de cas : la discographie de FZ	84

3.5.1	But de la modélisation	85
3.5.2	Inventaire des données	85
3.5.3	Construction du MCD (version 0)	86
3.5.4	Construction du MCD (version 1)	86
3.5.5	Construction du MCD (version 2)	88
3.5.6	Construction du MCD (version 3)	89
3.5.7	Construction du MCD (version 4)	91
3.5.8	Construction du MCD (version 5)	91
3.5.9	MCD final (?)	92
4	Bâtir les données	95
4.1	Introduction	96
4.1.1	Structured Query Language	96
4.1.2	Introduction aux contraintes d'intégrité	96
4.2	Ceci n'est pas une table	97
4.2.1	Définir une relation et ses attributs	98
4.2.2	Une table contient des « tuples »	100
4.2.3	Chaque table doit avoir une clé primaire	100
4.3	MCD \rightarrow MR : « 1 à plusieurs »	104
4.3.1	Intégrité référentielle: clé étrangère	106
4.3.2	Ce que contrôle une clé étrangère	107
4.3.3	Représentation schématique	109
4.4	« C'est pas faux »	109
4.4.1	Ce qu'il ne faut pas faire	111
4.4.2	Ce qu'il ne faut surtout pas faire	111
4.5	Comment identifier un tuple	112
4.5.1	Clés primaires et candidates	112
4.5.2	Générer des clés grâce aux séquences	113
4.5.3	Petit plaidoyer en faveur des <i>surrogate keys</i>	116
4.6	Un peu plus loin avec les contraintes	117
4.6.1	Assurer l'existence d'une valeur	117
4.6.2	Valeur par défaut	118
4.6.3	Vérifier que...	119
4.6.4	Exemple d'intégrité de domaine : plaques minéralogiques	120
4.7	MCD \rightarrow MR : « plusieurs à plusieurs »	124
4.7.1	Un petit bilan sur « 1 à plusieurs »...	124
4.7.2	... et on attaque « de plusieurs à plusieurs »	125
4.7.3	Finalement, pour « plusieurs à plusieurs »...	127
4.7.4	Autres cas de MCD \rightarrow MR	128
4.8	Gestion des dates	135
4.8.1	Trois principes à distinguer	135
4.8.2	Représentation interne	136
4.8.3	Affichage : « <i>to_char</i> » est l'outil qu'il vous faut	137
4.8.4	Saisie : « <i>to_date</i> » est l'outil qu'il vous faut	138
4.8.5	Amusons-nous avec les fuseaux horaires	138

4.9	Tas d'octets	141
4.10	Normalisation du modèle relationnel	142
4.10.1	Première forme normale (1FN)	143
4.10.2	Deuxième forme normale (2FN)	143
4.10.3	Troisième forme normale	144
4.10.4	Normalisons...	144
5	Manipuler les données	149
5.1	Demander simplement	150
5.1.1	Attributs sélectionnés	151
5.1.2	Filtrer	152
5.1.3	Opérateurs et fonctions	153
5.1.4	Priorités des opérateurs booléens	156
5.1.5	Avec ou sans relation, avec ou sans attribut	157
5.1.6	Trier (« ceci n'est pas une table », le retour)	158
5.1.7	Chercher et manipuler des dates	159
5.1.8	Recherche sur les chaînes de caractères	169
5.1.9	Outils divers	170
5.2	Modifier les données	171
5.2.1	Mettre à jour	171
5.2.2	Effacer	173
5.2.3	Ajouter	174
5.2.4	Introduction à l'atomicité	176
5.3	Assembler : les jointures	177
5.3.1	Produit cartésien	178
5.3.2	Jointure interne	179
5.3.3	Lever les ambiguïtés sur les noms des attributs	181
5.3.4	Jointures externes	181
5.3.5	Autres jointures	184
5.4	Agréger	186
5.4.1	Principe général	187
5.4.2	Un petit exemple	189
5.4.3	Fonctions d'agrégation et absence de valeur	192
5.4.4	Filtrer avant ou après l'agrégation	192
5.4.5	Agréger des chaînes	193
5.5	Combiner des requêtes	194
5.5.1	Mise en bouche : l'opérateur in	195
5.5.2	Opérations ensemblistes	195
5.5.3	Select de Select de Select de	197
5.5.4	S'il existe...	199
5.6	Utiliser des données externes	201
5.6.1	Remplir les relations à partir de fichiers	201
5.6.2	Stocker des fichiers	202

6 Stocker des traitements	207
6.1 Vues	208
6.1.1 Création des vues	208
6.1.2 Règles d'utilisation	210
6.1.3 Vues matérialisées	212
6.2 Avant-propos sur les procédures stockées	214
6.2.1 Pourquoi stocker des traitements?	214
6.2.2 Mise au point dans PostgreSQL	215
6.3 Un nouveau langage	215
6.4 Procédure	218
6.4.1 Appel d'une fonction	219
6.4.2 Arguments d'une fonction	219
6.4.3 Corps d'une fonction : les saxophonistes!	220
6.4.4 Valeur renvoyée par une fonction	221
6.4.5 Autre exemple pour se dégourdir le clavier	222
6.4.6 Surcharge des procédures/fonctions	225
6.4.7 Curseurs	226
6.4.8 Exceptions	228
6.4.9 Renvoyer plusieurs valeurs : <i>out argument</i>	231
6.4.10 Renvoyer des tuples	233
6.4.11 Calculer avec des nombres à virgule	235
6.5 Trigger	241
6.5.1 Trigger lors d'un insert	241
6.5.2 Trigger lors d'un update	243
6.5.3 Trigger lors d'un delete	246
6.5.4 Triggers niveau table	247
6.5.5 Dernières remarques sur les triggers	250
6.5.6 Ça passait, c'était beau (le retour)	251
6.6 Utiliser un autre langage	253
6.6.1 D'abord créer le langage	254
6.6.2 Modifier un fichier	255
6.6.3 Effacer un document	256
6.6.4 Conditions d'utilisation	257
6.7 Accès concurrents	258
6.7.1 Exception et accès concurrents	258
6.7.2 Séquences et accès concurrents	259
6.7.3 Transactions : retour sur l'atomicité	261
6.7.4 Isolation	263
6.7.5 Verrouillage	265
6.7.6 Contraintes déférables	266
6.7.7 Étude de cas : réservation de ressources	269
6.8 Interaction avec un programme : PHP	273
6.8.1 Requêtes paramétrées	274
6.8.2 Appeler une routine	276
6.8.3 Passer un tableau à une routine	276

6.8.4	Récupérer la valeur d'un argument « out »	278
7	Du côté de chez le DBA	281
7.1	Psql : la console à tout faire	282
7.1.1	Prompt (invite de commande)	282
7.1.2	Passer des commandes SQL	283
7.1.3	Commandes spécifiques à la console	284
7.1.4	Personnaliser psql	287
7.2	Gestion des droits	288
7.2.1	Concepts fondamentaux	288
7.2.2	Ajouter/gérer des utilisateurs	289
7.2.3	Accorder/révoquer des droits	290
7.2.4	Transmettre ses droits à d'autres	291
7.2.5	Rôle et groupes d'utilisateur	293
7.2.6	Droits et connexions au SGBD	295
7.2.7	Encore un peu plus loin...	297
7.3	Index	298
7.3.1	Index implicite avec une clé primaire	299
7.3.2	Index sur un champ non unique	302
7.3.3	Les index, la panacée?	303
7.4	I18n : internationalisation	306
7.5	Tables « système »	308
7.5.1	Catalogue	308
7.5.2	Quelques stats	311
7.5.3	Tailles des objets	312
7.6	Sauvegarde et restauration	314
7.6.1	Stratégies de sauvegarde	314
7.6.2	Dump/restore basique	315
7.7	Épilogue : la petite sirène	317
7.7.1	Création de la table des établissements	317
7.7.2	Insertion des données	321
7.7.3	Indexation	322
7.7.4	Petit point sur l'espace de stockage	324
7.7.5	Mise à jour de la base SIRENE	326
7.7.6	Un peu de ménage	330
A	Notes de production	333
A.1	Prérequis	334
A.2	Les sources du manuel	334
A.2.1	Structure	335
A.2.2	Styles	335
A.3	Compilation	335
A.4	Imprimer	336
A.5	Nettoyage	337
A.6	Chapitrage	337

Table des matières	365
Bibliographie	339
Glossaire	341
Index	349