Bases de données

Le modèle relationnel

Olivier Ridoux - 2020

Motivations

Les **bases de données** (**BDD**) sont des systèmes informatiques qui permettent de **stocker**, **calculer**, et **restituer** des **données**. On entend par données des valeurs comme **12**, **en chanteur**, **rue Morgue**, **Mme L'Espanaye**. Elles peuvent être associées pour former des **informations** comme **la rue Morgue est en chanteur** ou **Mme L'Espanaye habite rue Morgue**, ou la liste des rues en chanteur. C'est en tant qu'outil de gestion de l'information que les BDD sont essentielles.

On peut imaginer des BDD utilisées par une seule personne pour un unique service, comme pour gérer les livres d'un bibliophile, et pour cela presque n'importe quelle solution pourrait convenir. Cependant, on considère le plus souvent qu'une BDD doit pouvoir être utilisée par **plusieurs agents indépendants** et même à des **fins différentes**. Ex. les stocks d'une entreprise de e-commerce doivent être consultés par son service des achats, et sous une forme simplifiée par ses clients.

Pour que le développement d'applications qui utilisent des BDD puisse devenir un **métier** il convient qu'elles proposent des **façons de faire conventionnelles**. Il existe plusieurs façons de faire, mais sans doute la plus répandue aujourd'hui s'appelle le **modèle relationnel**. Ce modèle est disponible au travers de nombreux outils, qui sont pré-installés dans des systèmes variés (hébergement internet, ordinateurs, *smartphones*, *boxes* internet, etc.). L'objectif d'un ensemblement de BDD relationnelle est de fournir des éléments méthodologiques suffisants pour être autonome dans la conception d'applications comme on en trouve dans tous les domaines des technologies de l'information.

Les BDD relationnelles ont aussi une théorie, et même une **théorie qui marche** ! C'est-à-dire une théorie suffisamment puissante pour permettre de **prouver** des choses et **d'automatiser** des traitements fastidieux. Un module d'enseignement de BDD relationnelles entrelace donc des éléments théoriques et des éléments plus opérationnels.

UNIVERSITÉ DE RENNES 1 istic IRISA

Conclusion et perspectives

Les BDD sont un composant essentiel de la plupart des **systèmes d'information**. Dans leur variante relationnelle, le langage d'exploitation **SQL** domine, et est même imité dans d'autres modèles de BDD, comme **SPARQL** pour le **web sémantique**. Il est souvent étendu par des capacités à gérer des données **temporelles** ou **spatiales**, comme dans les **systèmes d'information géographiques** (**SIG**).

La plupart des éléments des BDD relationnelles (sémantique, requêtage, optimisation, normalisation, accès concurrent, etc.) sont susceptibles d'un **traitement formel** car les BDD relationnelles s'appuient sur une **théorie qui marche**.

Plus simples que l'architecture **client-serveur** conventionnelle on peut trouver des BDD minimalistes, mais sémantiquement complètes, dans la plupart des dispositifs portables or des équipements d'informatique domestique. Mais on fait aussi plus complexe avec des BDD réparties sur plusieurs sites ou dupliquées sur des sites miroirs, pour des raisons d'efficacité ou de tolérance aux pannes, ou qui gèrent des données en flux si intenses que le modèle **SQL ACID** n'est plus opérationnel.

L'accès aux données peut aussi être plus sophistiqué que le simple requêtage. Par exemple, la **fouille de données**, ou l'informatique décisionnelle, permet d'extraire des informations qui ne sont ni représentées explicitement, ni déductibles relationnellement, mais plutôt déductibles tendanciellement, ex. une corrélation.

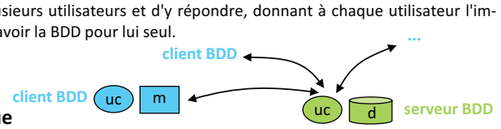
Bibliographie

- « *A Relational Model of Data for Large Shared Data Banks* » par Codd (Communications of the ACM, 1970), pour le texte fondateur.
- « *Foundations of Databases* » par Abiteboul, Hull et Vianu (Addison-Wesley, 1995), pour une somme théorique.
- « **Des bases de données à l'Internet** » par Matthieu (Vuibert, 2000), pour un cours d'introduction.
- « **Les bases de données** » par Conyn-Wattiau et Akoka (PUF, Que-sais-je ? 2003), pour un survol peu onéreux.
- « *The Manga Guide to Databases* » par Takahashi et Azuma (Ohmsha & No Starch Press, 2009), pour les BDD en BD.
- « *Beginning Database Design* » et « *Beginning SQL Queries* » par Churcher (APress, 2012), pour une étude pragmatique.
- « **Double assassinat dans la rue Morgue** » par Poe (1841), pour les exemples.

Bases de données, etc.

Le terme « base de données » désigne tantôt un modèle de logiciel de gestion de bases de données (ex. la BDD **MySQL**), tantôt une installation de ce logiciel (ex. la BDD de mon *smartphone*), tantôt un des contenus de cette installation (ex. la BDD clients). On parlera respectivement d'un **« logiciel de gestion »** de BDD, d'un **« système de gestion »** de BDD et d'une **« base de données »**.

Les BDD sont le plus souvent couplées à un dispositif de stockage **persistant**, ex. un ou plusieurs disques durs. Elles sont souvent installées sur un **serveur** de BDD comportant un **interpréteur** de requêtes capable de lire des requêtes provenant de plusieurs utilisateurs et d'y répondre, donnant à chaque utilisateur l'impression d'avoir la BDD pour lui seul.



Historique

Le modèle des **BDD relationnelles** a été proposé par **Edgar Franck Codd** en 1970. Il proposait de remplacer le point de vue physique sur le stockage des données qui prévalait à l'époque par un point de vue formel fondé sur la **logique des prédicats**. L'essor des BDD relationnelles a ensuite été très long, en partie parce qu'il a fallu apprendre à les mettre en œuvre efficacement. C'est encore un exemple où un nouveau paradigme n'est **pas strictement meilleur** que ses alternatives au moment de son introduction, il est seulement **plus prometteur**.

Le modèle relationnel a mis une vingtaine d'années à dominer le marché avec de très grands fournisseurs, comme **Oracle**, et de nombreux fournisseurs alternatifs, comme **SQLite** ou **MySQL** (racheté par Oracle), tous partageant les mêmes concepts résumés dans l'appellation BDD **SQL** (***Structured Query Language***). Des alternatives comme **Access** privilégient une interface graphique à l'interface linguistique de SQL. De nombreuses variantes existent, en particuliers pour le traitement de données spécifiques, ex. données temporelles ou géolocalisées.

D'autres modèles comme les BDD **orientées objets** ont tenté d'émerger, mais sans grand succès. Plus récemment, le mouvement **NoSQL** (***not only SQL***) et celui du **web sémantique** proposent des alternatives destinées le plus souvent à mieux prendre en compte le volume, l'hétérogénéité, la volatilité et l'éparpillement des données de certaines applications web.

Le modèle relationnel

Les données du modèle relationnel sont organisées en **tables**, ou **relations**. Visuellement, une relation est un tableau avec des colonnes ayant chacune un titre, qu'on appelle **attribut**. L'ensemble des attributs d'une relation constitue son **schéma**. Chaque ligne d'une table est une valuation pour chacun de ses attributs.

Formellement, une relation est un **ensemble** de lignes. Chaque ligne peut être lue comme un **axiome**, ex. **Mme L'Espanaye habite rue Morgue**, et chaque relation comme la **conjonction** de tous ses axiomes. On peut déduire de ces axiomes des propriétés comme **∃ r. Mme L'Espanaye habite r**, mais aussi en croisant plusieurs tables **∃ r. [Mme L'Espanaye habite r ∧ r est en chantier]**. On peut ainsi raisonner sur les relations et les formules, prouver des équivalences, et faire des **déductions**.

On peut aussi adopter un point de vue **algébrique** où les **déductions** sont représentées par des **opérations** algébriques. Connaissant la sémantique de ces opérations on peut prouver des identités remarquables et les appliquer pour transformer des requêtes comme on le fait en mathématiques pour simplifier/factoriser/développer des formules. On appelle cette algèbre **l'algèbre relationnelle**. Les principales opérations de cette algèbre sont les suivantes :

Restriction : $\sigma_{\text{propriété}}(\text{relation}) = \{ \text{ligne} \mid \text{ligne} \in \text{relation} \wedge \text{propriété(ligne)} \}$
La restriction sélectionne des **lignes** d'une relation sur la base de leur **contenu individuel**. Le résultat est une relation de même schéma, plus petite, qui est la conjonction d'une propriété en extension (**relation**) et d'une propriété en intension (**propriété**). Sélectionner des lignes sur la base d'une propriété collective, ex. **les rues dont les habitants dépassent 60 ans en moyenne**, demande d'autres moyens.

Produit : $\text{relation}_1 \times \text{relation}_2 =$
$\{ (\text{ligne}_1, \text{ligne}_2) \mid \text{ligne}_1 \in \text{relation}_1 \wedge \text{ligne}_2 \in \text{relation}_2 \}$
Le produit calcule toutes les combinaisons possibles des lignes de deux relations. Le résultat est une relation qui a tous les attributs de relation_1 et relation_2, et $\|\text{relation}_1\| \times \|\text{relation}_2\|$ lignes.

Projection : $\pi_{\text{attributs}}(\text{relation}) =$
$\{ \text{ligne} \mid \exists \text{ x... dans colonnes autres attributs.(ligne, x...)} \in \text{relation} \}$
La projection sélectionne des **colonnes** d'une relation sur la base de leur **nom**. Le résultat est une relation de schéma plus petit, et avec possiblement moins de lignes.

Opérations ensemblistes : **∪**, **∩** et **** s'appliquent à des relations de même schéma.

L'idiome $\pi_{\text{attributs}}(\sigma_{\text{propriété}}(\text{relation}_1 \times \text{relation}_2 \times \text{relation}_3 \times \ldots))$ recouvre à lui seul une grande part des déductions qu'on peut vouloir faire (ex. page 4).

SQL – *Structured Query Language*

SQL permet l'exploitation des BDD relationnelles. Il comporte 4 sous-langages :

Langage de définition des données : il permet de décrire leur organisation (les **schémas**). Par exemple, la requête

CREATE TABLE habite (nom TEXT, adresse TEXT)

déclare le schéma d'une nouvelle table **habite** qui aura un attribut **nom** et un attribut **adresse**, tous deux textuels. La table est créée vide.

Langage de manipulation de données : il permet de déposer des données, de les mettre à jour et de les rechercher. Par exemple, la requête

INSERT INTO habite(nom, adresse) ("Mme L'Espanaye", "rue Morgue")

ajoute la ligne **(Mme L'Espanaye, rue Morgue)** à la table **habite.** La requête

SELECT nom FROM habite WHERE adresse = "rue Morgue"

retourne la liste des noms de tous les habitants de la rue Morgue, soit

$\pi_{\text{nom}}(\sigma_{\text{adresse = rue Morgue}}(\text{habite}))$.

Les mots **nom**, **habite** et **adresse** y sont correctement interprétés grâce à la déclaration de schéma qui a précédé. Et la requête

SELECT nom FROM habite, etat_voirie
WHERE etat_voirie.voie = habite.adresse AND etat = "en chantier"

retourne la liste des noms de tous les habitants d'une rue en chantier, soit

$\pi_{\text{nom}}(\sigma_{\text{etat_voirie.voie = habite.adresse} \wedge \text{etat = en chantier}}(\text{habite} \times \text{etat_voirie}))$.

Ici, on a croisé les informations de deux tables ; on appelle cela une **jointure**. Une BDD contient **explicitement** les informations stockées dans ses tables, et **implicitement** celles qui peuvent s'en déduire. Une requête SELECT construit donc une table aussi bien qu'une requête CREATE TABLE. SQL permet d'en profiter comme suit

SELECT nom FROM habite,
(SELECT voie FROM etat_voirie WHERE etat = "en chantier")
AS voie_en_chantier
WHERE voie_en_chantier.voie = habite.adresse

soit $\pi_{\text{nom}}(\sigma_{\text{voie_en_chantier.rue=habite.adresse}}($
$\text{habite} \times \pi_{\text{voie}}(\sigma_{\text{etat=en chantier}}(\text{etat_voirie})) \text{ as voie_en_chantier}))$.

Langage de contrôle des données : il permet de dire qui a accès aux données. Par exemple, la requête

GRANT ALL ON habite TO poe@localhost

donne à l'utilisateur **Poe** tous les droits sur la table **habite**.

Langage de contrôle des transactions : voir page 7.

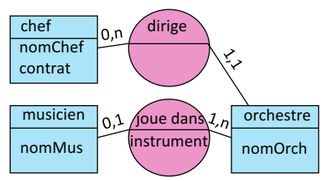
SQL est donc une sorte de langage de programmation pour qui la table est l'unité de calcul et dont la sémantique est donnée par l'algèbre relationnelle.

Conception

Une BDD contient de l'information **explicite** dans ses tables et **implicite** via ses déductions. Une mise à jour peut donc avoir des effets indirects désastreux, ex. perte d'une information implicite. Pour prévenir ce risque, on peut **concevoir** des schémas reconnus plus sûrs, exprimer des **contraintes d'intégrité** (ex. **âge < 200 ans**), ou spécifier des **règles de propagation** des mises à jour.

Une difficulté de la mise à jour vient de la **redondance** de donnée ; c-à-d. quand une même information est représentée plusieurs fois. Dans une telle situation, il est très difficile de s'assurer qu'une mise à jour laisse la BDD dans un état cohérent. Ex. on aurait pu décider de noter l'état de la voirie dans la relation **habite** (ex. **habite(Mme L'Espanaye, rue Morgue, en chantier)**), mais cela entraîne que l'état d'une rue est noté pour chacun de ses habitants, et que lorsqu'il change il faut le mettre à jour partout. Cela entraîne aussi que l'état d'une rue disparaît de la BDD quand son dernier habitant disparaît ! On a le choix entre adopter une **modélisation** qui ne crée pas ces situations ou les supprimer par **normalisation**.

Modélisation : on analyse une situation du monde réel en y identifiant des classes **d'entités** (correspondant ± à des **noms communs**), leurs **attributs** (± des **adjectifs**) et des classes **d'associations** (± des **verbes**). On relie entre elles les classes d'entités, toujours via une association que l'on quantifie à l'aide de **cardinalités** pour exprimer à combien d'entités d'une autre classe peut être associée une entité d'une classe. Ex. on pourra dire que chaque personne ne vit qu'à une adresse, mais qu'à une adresse peuvent vivre plusieurs personnes. On peut alors traduire ces entités et associations en **relations**, pour obtenir un ensemble de relations de bonne qualité face au risque d'incohérence lors d'une mise à jour.



Exemple : Un **chef dirige** 0 ou n **orchestres**, mais un orchestre n'a qu'un chef. Un **musicien joue dans** 0 ou 1 **orchestres**, mais un orchestre a 1 ou n musiciens. Le musicien joue d'un instrument pour son orchestre, mais pourrait jouer d'un autre ailleurs.

Le tout se traduit dans les relations **chef(nomChef, contrat)**, **musicien(nomMus)**, **joue_dans(nomMus, instrument, nomOrch)**, et **orchestre(nomOrch, nomChef)**. Noter comment l'association **dirige** a été absorbée dans la relation **orchestre**.

Normalisation : soit un schéma, ex. **musicien(nomMus, instrument, nomOrch, nomChef, contrat)** et l'énoncé de **dépendances fonctionnelles** (**DF**) entre ses attributs. Une DF $\mathcal{A} \rightarrow \mathcal{B}$ exprime que les valeurs des attributs $\mathcal{A}$ **déterminent** les valeurs des attributs $\mathcal{B}$. Ex. si le nom d'un musicien détermine son orchestre et l'instrument qu'il y joue, on note **nomMus → nomOrch, instrument**, et de façon similaire **nomOrch → nomChef** et **nomChef → contrat**.

On voit alors que la relation **musicien** est **redondante** car le nom et le contrat du chef d'un orchestre seront notés dans toutes les lignes des musiciens de l'orchestre. Cela se détecte sans attendre de peupler la BDD car **nomMus** détermine tous les attributs, **nomOrch** détermine une partie d'entre eux, et **nomChef** une sous-partie. Les DF servent alors de guides pour **décomposer** la relation en sous-relations sans redondance telles que la relation de départ peut être reconstituée. Ici on produirait **musicien(nomMus, instrument, nomOrch), orchestre(nomOrch, nomChef)** et **chef(nomChef)**.

Plusieurs sortes de redondances ont été analysées théoriquement, et sont éliminées de cette façon dans des **formes normales** de forces croissantes.

Normaliser n'est pas une fin en soi. Si une BDD subit très peu de mises à jour, on peut même choisir de la **dénormaliser après** pour des raisons d'efficacité.

On convient que l'interprétateur de requêtes doit garantir les propriétés **d'atomicité**, de **cohérence**, **d'isolation** et de **durabilité**, qu'on appelle collectivement propriétés **ACID**. Des requêtes qui respectent ces propriétés sont appelées des **transactions**.

Atomicité : une requête est interprétée **en entier ou pas du tout**. Si elle doit être interrompue, toutes les mises à jour qu'elle avait commencé à faire sont annulées (commande **ROLLBACK** du langage de contrôle des transactions). Sinon elles sont confirmées (commande **COMMIT**). On enregistre pour cela une liste des mises à jour faites depuis le début de l'interprétation de la requête dans un **journal**.

Cohérence : l'exécution d'une requête laisse la BDD conforme à ses **contraintes d'intégrité**.

Isolation : l'avancement d'une requête est **invisible** depuis d'autres requêtes. Si elle doit faire une mise à jour en plusieurs fois, il est impossible d'observer les situations intermédiaires. On utilise pour cela des **verrous** qui garantissent un accès protégé à la donnée mise à jour.

Durabilité : l'effet d'une requête de mise à jour est **définitif**. Ex. le versement d'un salaire ne doit pas disparaître, même en cas de panne du système ! On obtient cet effet par des **sauvegardes** qui enregistrent des copies de la BDD, et par un **journal** qui enregistre les mises à jour depuis la dernière sauvegarde.

Les propriétés ACID sont coûteuses à mettre en œuvre. Des bases de données très volatiles et volumineuses comme celles des **réseaux sociaux** tendent à s'en affranchir en adoptant des modèles **NoSQL**.

Ces rôles sont transposables dans la plupart des métiers de l'informatique.

Rôles

La théorie des BDD distingue plusieurs rôles d'agents.

L'administrateur de BDD : gère un système de gestion de BDD, ex. l'utilisation des ressources par le système et à l'attribution des droits d'accès au système. Il peut définir un schéma général des données, une politique de sécurité, etc.

L'administrateur d'application : gère l'accès aux données en vue d'une application. Il définit des schémas spécifiques à l'application qui vont masquer le schéma général aux yeux des programmeurs d'application et des utilisateurs.

Le programmeur d'application : développe les applications qui utilisent la BDD. Il ne voit que les schémas qu'a préparés l'administrateur d'application.

L'utilisateur : ne voit que l'interface homme-machine de l'application ; il ne voit pas la BDD, même si celle-ci est sollicitée par l'application.

Prérequis : lire, comprendre, apprendre, puis procéder au pliage.
Pliage : autre face dessus, traits **gris** rentrants, traits **rouges** saillants. Découper selon le trait rouge entre les deux ●, puis achever le pliage.

① Théorie naïve des ensembles

Un **ensemble** est une collection d'objets où le rang et la multiplicité ne comptent pas. On appelle **éléments** les objets de la collection, et on dit qu'un élément **appartient** à un ensemble. Cela se note $e \in E$ pour « *e appartient à E* », et $e, e' \in E$ pour « *e eteé appartiennent à E* ».

On note une collection entre accolades ({…}). Ex. **{a, b, c}** est une collection, et puisque le rang ne compte pas, la collection **{b, c, a}** dénote le même ensemble, et puisque la multiplicité ne compte pas non plus, la collection **{a, b, a, c}** dénote encore le même ensemble.

Dans la théorie naïve des ensembles on ne considère que des objets qu'il est toujours possible de distinguer les uns des autres, mais à part cela n'importe quoi peut être un objet, y compris un ensemble.

Une collection vide, **{}**, constitue un ensemble particulier qu'on appelle **l'ensemble vide**, et qu'on note **∅** ou **{}**. Il n'y a qu'un ensemble vide. On appelle **singleton** un ensemble qui n'a qu'un élément. L'ensemble **{∅}** est donc un singleton. Ne pas confondre l'élément **a** et le singleton **{a}**. Les confondre revient à commettre une erreur de type en programmation. Ne pas confondre non plus **∅** et **{∅}**.

La collection des éléments d'un ensemble s'appelle **l'extension** de l'ensemble. On peut aussi spécifier les éléments d'un ensemble par une propriété qui les distingue de ceux qui n'appartiennent pas à l'ensemble. Cette propriété s'appelle **l'intension** de l'ensemble (l'intenSion n'a rien à voir avec l'intenTion, ou dessein). On exprime une intension soit par une phrase en prenant le risque d'être imprécis ou ambigu, soit par une formule logique en prenant le risque d'être fastidieux. N'importe quelle formule logique fausse (contradictoire ou absurde) est l'intension de **∅**.

Ex. « ***les lettres communes aux mots ``bancal'' et ``tabac''*** » est une intension possible pour l'ensemble constitué de la collection **{a, b, c}**. Les « ***3 premières lettres de l'alphabet*** » est une intension équivalente. Un ensemble peut avoir de nombreuses intensions, certaines même difficiles à comparer.

Si tous les éléments d'un ensemble E_1 appartiennent à un ensemble E_2, on dit que E_1 est **inclus** dans E_2, et on note $E_1 \subset E_2$. On dit aussi que E_1 est un **sous-ensemble** ou une **partie** de E_2. On note $E_1 \subsetneq E_2$ pour signifier l'inclusion sans égalité possible (on parle alors de sous-ensemble **strict**), et $E_1 \subseteq E_2$ pour insister sur l'égalité possible.

① Logique des prédicats

Dans sa définition la plus naïve la **logique des prédicats** est la formalisation de la logique employée tous les jours dans les activités un peu scientifiques. Elle permet d'exprimer des **jugements** sous forme de formules, et de décider formellement si elles sont **vraies** ou **fausses**. Les formules de la logique des prédicats sont définies de la façon suivante :

Prédicats atomiques (ou **atomes**) **:** Ce sont des formules élémentaires qui expriment des jugements sur des objets. Nous ne nous prononçons pas sur la notation à employer dans les atomes ; cela n'a pas vraiment d'importance. Par exemple,
- **Rennes est une capitale**, où **Rennes** est **l'objet** du jugement et **être une capitale** en est le **prédicat**.
- **Rennes est la capitale de la France**, où **Rennes** et la **France** sont des objets et **être la capitale de** est le prédicat.
- **x > y**, où **x** et **y** sont les objets, et où **>** (être plus grand que) est le prédicat du jugement.
- **1273 + 556 est un nombre premier**, où **1273 + 556** est l'objet et **être un nombre premier** est le prédicat.
- **Le Grand théorème de Fermat est vrai**, où le **Grand théorème de Fermat** est l'objet et **être vrai** est le prédicat.
- $\mathcal{P} \neq \mathcal{NP}$, où $\mathcal{P}$ et $\mathcal{NP}$ sont les objets et **≠** (être différent) le prédicat.

On peut attribuer une valeur de vérité, **Vrai** ou **Faux**, à un atome, en se référant à une réalité de terrain, comme pour Rennes, à des théories et des calculs, comme pour la primalité de **1273 + 556**, ou à des preuves externes, comme celle du grand théorème de Fermat. Parfois on ne peut pas, par manque d'information, comme pour **x > y**, ou parce qu'on ne sait vraiment pas, comme pour $\mathcal{P} \neq \mathcal{NP}$.

Formules propositionnelles : Ce sont des formules, élémentaires ou non, qui sont reliées par des connecteurs logiques, généralement **∧**, **∨**, **¬**, ou **⇒**.
- **conjonction,** $\phi_1 \wedge \phi_2$ **:** relie deux formules pour en former une troisième qui est vraie ssi les deux premières le sont. Le connecteur **∧** est commutatif, associatif, et a pour élément neutre la valeur de vérité **Vrai**. Cela permet la notation de conjonction étendue comme $\bigwedge_{i \in [1,n]} \phi_i$. Si on convenait que **Faux < Vrai**, $\phi_1 \wedge \phi_2$ serait le minimum de ϕ_1 et ϕ_2 . Si on convenait que **Faux = 0** et **Vrai = 1**, $\phi_1 \wedge \phi_2$ serait $\phi_1 \times \phi_2$.
- **disjonction,** $\phi_1 \vee \phi_2$ **:** relie deux formules pour en former une troisième qui est vraie ssi au moins une des deux premières l'est. Le connecteur **∨** est commutatif, associatif, et a pour élément neutre la valeur de vérité **Faux**. Cela permet la notation de disjonction étendue comme $\bigvee_{i \in [1,n]} \phi_i$. Si on convenait que **Faux < Vrai**, $\phi_1 \vee \phi_2$ serait le maximum de ϕ_1 et ϕ_2. Si on convenait que **Faux = 0** et **Vrai = 1**, $\phi_1 \vee \phi_2$ serait $1 - (1-\phi_1) \times (1-\phi_2)$.
- **implication,** $\phi_1 \Rightarrow \phi_2$ **:** relie deux formules pour en former une troisième qui est fausse ssi ϕ_1 est vraie alors que ϕ_2 est fausse. Le connecteur **⇒** n'est ni commutatif ni associatif. Si on convenait que **Faux < Vrai**, $\phi_1 \Rightarrow \phi_2$ serait **Vrai** ssi $\phi_1 \leq \phi_2$.
- **négation,** $\neg \phi$ **:** constitue une formule qui est fausse ssi ϕ est vraie. Si on convenait que **Faux = 0** et **Vrai=1**, $\neg \phi$ serait $1 - \phi$.

Formules quantifiées : Ce sont des formules, élémentaires ou non, dont la valeur de vérité s'évalue par rapport à un ensemble d'objets, plutôt que par rapport à des objets pris individuellement.
- **quantification universelle,** $\forall x . \phi(x)$ **:** constitue une formule qui est vraie ssi ϕ est vraie de tous les éléments du domaine. Si le domaine est fini, la quantification universelle est équivalente à une conjonction des applications de ϕ à tous les éléments du domaine : $\forall x \in \{e_1, e_2, \ldots, e_n\}. \phi(x)$ ssi $\bigwedge_{i \in [1,n]} \phi(e_i)$.
- **quantification existentielle,** $\exists x . \phi(x)$ **:** constitue une formule qui est vraie ssi ϕ est vraie d'au moins un élément du domaine. Si le domaine est fini, la quantification existentielle est équivalente à une disjonction des applications de ϕ à tous les éléments du domaine: $\exists x \in \{e_1, e_2, \ldots, e_n\}. \phi(x)$ ssi $\bigvee_{i \in [1,n]} \phi(e_i)$.

Parenthèses et priorité des opérateurs : Il est courant d'assigner des priorités d'opérateurs aux différents connecteurs et quantificateurs afin de spécifier comment se lisent les formules complexes en l'absence de parenthèses. Nous pensons que c'est utile mais absolument pas fondamental car ces conventions dépendent trop des outils, et même quand ce n'est pas le cas, il n'est jamais très prudent de se reposer trop lourdement sur les priorités des opérateurs quand le coût de quelques parenthèses est si faible devant le coût d'une grosse erreur. Ce genre d'expertise est à réserver aux experts. Il vaut mieux ne pas être avare de parenthèses, et nous proposons d'utiliser les parenthèses rondes (**(…)**) pour structurer les connecteurs, et les parenthèses carrées (ou crochets, **[…]**) pour les quantificateurs.

N'oubliez jamais !

Un **modèle** est toujours **imparfait**, parfois **utile**, et c'est tout ce qu'on peut lui demander (d'après George Box 1978, voir aussi a contrario les cartes à l'échelle 1/1 de Jorge Luis Borges 1946 et Lewis Carroll 1893, si parfaites qu'elles sont inutiles).

L'informatique ne travaille que sur la **représentation** des choses. Il faut toujours se demander ce qui a du sens par rapport à ce qui est représenté.

② Algèbre des ensembles

Étant donné un **univers** $\mathcal{U}$ d'objets de références, on peut former une structure algébrique sur les sous-ensembles de $\mathcal{U}$ avec les opérations suivantes :

Produit cartésien (noté **×**) **:** $A \times B$ est l'ensemble formé de tous les couples (a, b) où $a \in A$ et $b \in B$. Bien noter que l'ordre compte et que $(a, b) \neq (b, a)$ sauf si $a=b$. Noter aussi que $A \times \emptyset = \emptyset$.

Ensemble des parties (noté $\mathcal{P}(E)$ ou 2^E) **:** $\mathcal{P}(E)$ est l'ensemble formé de toutes les parties de **E**, donc un ensemble d'ensembles. Noter que $\mathcal{P}(\emptyset) = \{\emptyset\}$ et $\mathcal{P}(\{a\}) = \{\emptyset, \{a\}\}$.

Union (notée **∪**) **:** $A \cup B$ est l'ensemble formé des éléments de A et de ceux de B. Comme la multiplicité ne compte pas, il est indifférent qu'un élément appartienne aux deux ensembles ou seulement à l'un d'entre eux. Par exemple, $\{a, b\} \cup \{c, b\} = \{a\} \cup \{b, c\} = \{a, b, c\}$. L'union est **commutative**, **associative**, **idempotente**, et a **∅ pour élément neutre**. Cela autorise la notation d'union étendue comme $\bigcup_{i \in [1,n]} E_i$. L'intension d'une union d'ensembles est la disjonction des intensions de ces ensembles. Noter que $A \subseteq B$ ssi $A \cup B = B$, et que $\bigcup_{i \in \emptyset} E_i = \emptyset$.

Intersection (notée **∩**) **:** $A \cap B$ est l'ensemble formé des éléments de A qui appartiennent aussi à B. Par exemple, $\{a, b\} \cap \{c, b\} = \{b\}$. L'intersection est **commutative**, **associative**, **idempotente**, et a $\mathcal{U}$ **pour élément neutre.** Cela autorise la notation d'intersection étendue comme $\bigcap_{i \in [1,n]} E_i$. L'intension d'une intersection d'ensembles est la conjonction des intensions de ces ensembles. Noter que $A \subseteq B$ ssi $A \cap B = A$, et que $\bigcap_{i \in \emptyset} E_i = \mathcal{U}$.

Complémentation (notée **∁**) **:** $\complement_A B$ est l'ensemble formé de la collection des éléments de **A** qui n'appartiennent pas à **B** ; le **complémentaire** de B dans A. On note parfois **A\B** ou même **A-B**. Il n'est pas nécessaire que B soit inclus dans A ! Quand l'ensemble de référence est $\mathcal{U}$ on peut se dispenser de le noter : $\complement B = \complement_{\mathcal{U}} B$. Noter que $\complement \complement B = B$. L'intension du complémentaire d'un ensemble est la conjonction de l'intension de l'ensemble de référence et de la négation de l'intension de l'ensemble.

L'intersection et l'union sont **distributives** l'une par rapport à l'autre : pour tout triplet d'ensembles A, B et C, $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ et $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$. La complémentation est une sorte de négation, et suit les **lois de De Morgan** : $\complement_A(B \cup C) = \complement_A B \cap \complement_A C$ et $\complement_A(B \cap C) = \complement_A B \cup \complement_A C$.

Même si l'une fait penser à l'addition et l'autre à la soustraction, **l'union et la complémentation ne sont pas les opposées l'une de l'autre** ; pour certaines paires d'ensembles A et B, on a $(A \backslash B) \cup B \neq A$ ou $(A \cup B) \backslash B \neq A$. Ex. $(\{a\} \backslash \{b\}) \cup \{b\} = \{a, b\}$ et $(\{a, b\} \cup \{b\}) \backslash \{b\} = \{a\}$.

② Idiomes logiques

En pratique, on n'utilise pas n'importe quelle formule de la logique des prédicats. On a tendance à utiliser des expressions idiomatiques qu'il convient de reconnaître et interpréter correctement au premier coup d'œil.

Quantifications dans un domaine : Très souvent, on écrit des formules comme $\forall x \in E . \phi(x)$ ou $\forall x \text{ tq } \psi(x) . \phi(x)$. C'est une façon de dire dans quel domaine doit être évaluée la quantification. Ces formules **doivent** être lues $\forall x . [x \in E \Longrightarrow \phi(x)]$ et $\forall x . [\psi(x) \Longrightarrow \phi(x)]$. Une conséquence directe est qu'une quantification universelle sur un domaine vide est trivialement vraie. Cela paraît une situation étrange, mais c'est banal en informatique, spécialement quand on considère les cas d'initialisation : ex. **Tous les utilisateurs sont …** lorsqu'il n'y a pas d'utilisateurs. De la même façon, on écrit des formules comme $\exists x \in E . \phi(x)$ ou $\exists x \text{ tq } \psi(x) . \phi(x)$. Ces formules **doivent** être lues $\exists x . [x \in E \wedge \phi(x)]$ et $\exists x . [\psi(x) \wedge \phi(x)]$. On voit alors qu'une quantification existentielle sur un domaine vide est trivialement fausse.

Cascades d'implications : On écrit parfois des formules comme $\phi_1 \Rightarrow \phi_2 \Rightarrow \phi_3$, qui pourrait être lue soit $(\phi_1 \Rightarrow \phi_2) \Rightarrow \phi_3$ soit $\phi_1 \Rightarrow (\phi_2 \Rightarrow \phi_3)$. Dans le premier cas, on exprime que $\phi_1 \Rightarrow \phi_2$ est un théorème qui pourrait servir à démontrer ϕ_3. Dans le second cas, la formule est équivalente à $(\phi_1 \wedge \phi_2) \Rightarrow \phi_3$, donc $(\phi_2 \wedge \phi_1) \Rightarrow \phi_3$, et donc $\phi_2 \Rightarrow (\phi_1 \Rightarrow \phi_3)$. Ce qui n'a rien à voir avec la première lecture. Conclusion, faire des économies de bouts de parenthèses avec discernement !

Formules de De Morgan (Augustus De Morgan, 1806-1871) : La négation a à voir avec le complémentaire d'une situation, mais le complémentaire d'une situation compliquée est souvent encore plus compliqué que la situation. Les formules de De Morgan (qui ne sont pas toutes dues à De Morgan) peuvent être mises à profit pour pousser les négations vers l'intérieur des formules où elles s'appliqueront à des formules plus petites.
- $\neg (\phi_1 \wedge \phi_2)$ est équivalent à $\neg \phi_1 \vee \neg \phi_2$ et $\neg (\phi_1 \vee \phi_2)$ est équivalent à $\neg \phi_1 \wedge \neg \phi_2$.
- $\neg (\phi_1 \Rightarrow \phi_2)$ est équivalent à $\phi_1 \wedge \neg \phi_2$.
- $\neg\neg \phi$ est équivalent à ϕ. Pose beaucoup plus de difficultés que la taille de l'identité ne le laisse penser, mais est vrai pour la logique usuelle. Se rappeler combien nous humains sommes peu doués pour les doubles négations.
- $\neg \forall x . \phi(x)$ est équivalent à $\exists x . \neg \phi(x)$ et $\neg \exists x . \phi(x)$ est équivalent à $\forall x . \neg \phi(x)$.

Cascades de quantifications : On imbrique souvent les quantifications. Certaines imbrications doivent faire réfléchir, et d'autres moins.
- $\exists x . \exists y . \phi(x, y)$ est équivalent à $\exists y . \exists x . \phi(x, y)$ et on note souvent $\exists x, y . \phi(x, y)$.
- $\forall x . \forall y . \phi(x, y)$ est équivalent à $\forall y . \forall x . \phi(x, y)$ et on note souvent $\forall x, y . \phi(x, y)$.
- $\forall x . \exists y . \phi(x, y)$ exprime que pour chaque **x** il y a un **y**, **qui peut dépendre de x**, qui a la propriété désirée. Ex. dans $\forall x . \exists y . x \times y = 0$, le même **y** convient pour tous les **x**, mais dans $\forall x . \exists y . x \times y = 1$, à tout **x** correspond un **y** différent. Un y existentiel est donc implicitement une fonction de tous les x universels qui le précèdent.
- $\exists x . \forall y . \phi(x, y)$ exprime qu'un **x** unique a la propriété désirée pour tous les **y**. Ex. $\exists x . \forall y . x \times y = 0$ est vrai, mais pas $\exists x . \forall y . x \times y = 1$.
- $\exists x . [\psi \between \phi(x)]$ est équivalent à $\psi \between \exists x . \phi(x)$ et $\forall x . [\psi \between \phi(x)]$ est équivalent à $\psi \between \forall x . \phi(x)$ si **x** n'apparaît pas dans ψ, et $\between$ représente **∧** ou **∨**.

③ Cardinalité des ensembles finis

Le nombre d'éléments d'un ensemble fini est appelé la cardinalité de l'ensemble. La cardinalité d'un ensemble **E** est notée **card(E)** ou $\|E\|$. La cardinalité des ensembles suit les lois suivantes :

Produit cartésien : $\|A \times B\| = \|A\| \times \|B\|$. Noter la surcharge du signe **×** qui opère tantôt sur des ensembles (le produit cartésien) et tantôt sur des entiers (le produit arithmétique).

Ensemble des parties : $\|\mathcal{P}(E)\| = 2^{\text{card}(E)}$, ce qui explique un peu la notation 2^E pour l'ensemble des parties de **E**.

Union : $\max\{\|A\|, \|B\|\} \leq \|A \cup B\| \leq \|A\| + \|B\|$ car des éléments peuvent appartenir aux deux ensembles, mais ne doivent être comptés qu'une fois.

Intersection : $0 \leq \|A \cap B\| \leq \min\{\|A\|, \|B\|\}$. Plus précisément, $\|A \cup B\| = \|A\| + \|B\| - \|A \cap B\|$.

Complémentation : $\|\complement_A B\| = \|A\| - \|A \cap B\|$.