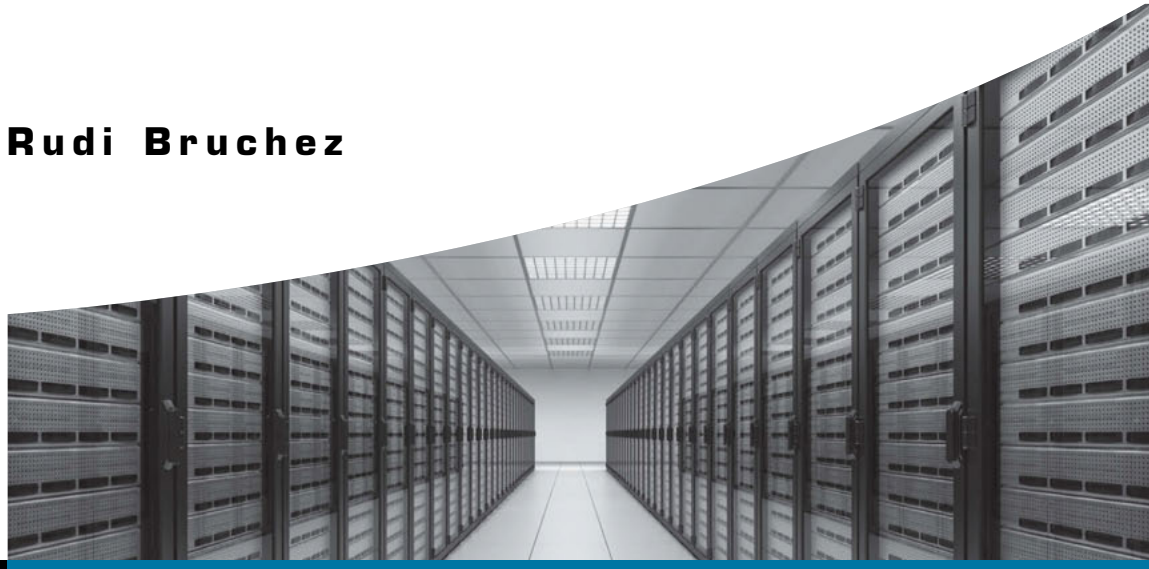
Les bases de données NoSQL et le Big Data

2e édition

Comprendre et mettre en oeuvre

Rudi Bruchez



EYROLLES

Les bases de données NoSQL et le Big Data

2e édition

R. Bruchez

Consultant informatique indépendant, Rudi Bruchez est expert en bases de données depuis presque vingt ans (certifications MCDBA et MCITP). Il assure conseil, réalisation, expertise et formation pour la modélisation, l'administration et l'optimisation des serveurs et du code SQL, ainsi que des services autour de SQL Server et des solutions NoSQL. Il est l'auteur ou coauteur de plusieurs ouvrages français et anglais sur SQL Server et SQL, dont *Optimiser SQL Server* (éditions Dunod) et *SQL* (éditions Pearson).

Des bases pour la performance et le Big Data

En quelques années, le volume des données brassées par les entreprises a considérablement augmenté. Émanant de sources diverses (transactions, comportements, réseaux sociaux, géolocalisation…), elles sont souvent structurées autour d'un seul point d'entrée, la clé, et susceptibles de croître très rapidement. Autant de caractéristiques qui les rendent très difficiles à traiter avec des outils classiques de gestion de données. Par ailleurs, l'analyse de grands volumes de données, ce qu'on appelle le Big Data, défie également les moteurs de bases de données traditionnels.

C'est pour répondre à ces différentes problématiques que sont nées les bases de données NoSQL *(Not Only SQL)*, sous l'impulsion de grands acteurs du Web comme Facebook ou Google, qui les avaient développées à l'origine pour leurs besoins propres. Grâce à leur flexibilité et leur souplesse, ces bases non relationnelles permettent en effet de gérer de gros volumes de données hétérogènes sur un ensemble de serveurs de stockage distribués, avec une capacité de montée en charge très élevée. Elles peuvent aussi fournir des accès de paires clé-valeur en mémoire avec une très grande célérité. Réservées jusqu'à peu à une minorité, elles tendent aujourd'hui à se poser en complément du modèle relationnel qui dominait le marché depuis plus de 30 ans.

Du choix de la base NoSQL à sa mise en œuvre

Cet ouvrage d'une grande clarté dresse un panorama complet des bases de données NoSQL, en analysant en toute objectivité leurs avantages et inconvénients. Dans une première partie, il présente les grands principes de ces bases non relationnelles : interface avec le code client, architecture distribuée, paradigme MapReduce, etc. Il détaille ensuite dans une deuxième partie les principales solutions existantes (les solutions de Big Data autour de Hadoop, MongoDB, Cassandra, Couchbase Server…), en précisant spécificités, forces et faiblesses de chacune. Complétée par une étude de cas réel, la dernière partie du livre est consacrée au déploiement concret de ces bases : dans quel cas passer au NoSQL ? quelle base adopter selon ses besoins ? quelles données basculer en NoSQL ? comment mettre en place une telle base ? comment la maintenir et superviser ses performances ?

Au sommaire

À qui s'adresse cet ouvrage ?

– Aux experts en bases de données, architectes logiciels, développeurs…

– Aux chefs de projet qui s'interrogent sur le passage au NoSQL

www.editions-eyrolles.com

EYROLLES

Les bases de données NoSQL et le Big Data

2e édition

Comprendre et mettre en oeuvre

Rudi Bruchez

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

DU MÊME AUTEUR

C. SOUTOU, F. BROUARD, N. SOUQUET et D. BARBARIN. – **SQL Server 2014.**
N°13592, 2015, 890 pages.

C. SOUTOU. – **Programmer avec MySQL (3e édition).**
N°13719, 2013, 520 pages.

C. SOUTOU. – **Modélisation de bases de données (3e édition).**
N°14206, 2015, 352 pages. *À paraître*.

C. SOUTOU. – **SQL pour Oracle (7e édition).**
N°14156, 2015, 666 pages.

R. BIZOÏ – **Oracle 12*c* – Administration.**
N°14056, 2014, 564 pages.

R. BIZOÏ – **Oracle 12*c* – Sauvegarde et restauration.**
N°14057, 2014, 336 pages.

R. BIZOÏ – **SQL pour Oracle 12*c*.**
N°14054, 2014, 416 pages.

R. BIZOÏ – **PL/SQL pour Oracle 12*c*.**
N°14055, 2014, 340 pages.

C. PIERRE DE GEYER et G. PONÇON – **Mémento PHP et SQL (3e édition).**
N°13602, 2014, 14 pages.

R. BIZOÏ – **Oracle 11*g* – Administration.**
N°12899, 2011, 600 pages.

R. BIZOÏ – **Oracle 11*g* – Sauvegarde et restauration.**
N°12899, 2011, 432 pages.

G. BRIARD – **Oracle 10*g* sous Windows.**
N°11707, 2006, 846 pages.

R. BIZOÏ – **SQL pour Oracle 10*g*.**
N°12055, 2006, 650 pages.

G. BRIARD – **Oracle 10*g* sous Windows.**
N°11707, 2006, 846 pages.

G. BRIARD – **Oracle9*i* sous Linux.**
N°11337, 2003, 894 pages.

Table des matières

Avant-propos

Ce qu'on appelle le « mouvement NoSQL » est encore relativement jeune en France, même s'il existe depuis plusieurs années aux États-Unis. Mais il suscite déjà un vif intérêt et de nombreux débats, plus que ne l'ont fait les précédents écarts par rapport au modèle dominant de la gestion des bases de données, à savoir le modèle relationnel.

L'une des principales raisons de cet intérêt provient de son approche pragmatique. Les moteurs non relationnels regroupés sous la bannière du NoSQL ont été souvent conçus sous l'impulsion d'entreprises qui voulaient répondre à leurs propres besoins. Ils sont également les produits de leur temps, où les machines sont moins chères et la distribution sur plusieurs nœuds est l'une des solutions les plus accessibles de montée en charge.

On pourrait aussi considérer le NoSQL comme le fruit d'une vieille opposition entre le développeur, qui souhaite utiliser ses outils rapidement, de manière presque ludique, et sans aucune contrainte, et l'administrateur, qui protège son système et met en place des mécanismes complexes et contraignants.[1] D'ailleurs, beaucoup de développeurs n'aiment pas les bases de données. Ils sont pourtant obligés de s'en servir, car l'informatique manipule de l'information, laquelle est stockée dans des systèmes de gestion de bases de données (SGBD). Ils vivent comme une intrusion cet élément externe, qui s'intègre mal avec leur code, et qui les oblige à apprendre un langage supplémentaire, faussement facile, le SQL.

Mais le mouvement NoSQL ne peut se résumer à ce rejet du SQL : certains de ses moteurs sont le résultat de recherches et d'expérimentations. D'autres permettent de garantir les meilleures performances possibles en conservant un modèle de données très simple. Sous la bannière NoSQL, ces outils ont ainsi des identités différentes et répondent à des besoins divers, allant du temps réel au *Big Data*.

Un ouvrage impartial

C'est pourquoi écrire un livre sur le NoSQL dans son ensemble pourrait paraître un peu artificiel, puisque en fin de compte, la seule composante qui réunit des outils comme Cassandra, MongoDB ou Redis, c'est l'abandon du modèle relationnel. Mais somme toute, ce n'est pas qu'un petit point commun. Le modèle relationnel est le modèle dominant depuis la fin des années 1980, cela fait

1. Un mouvement, nommé DevOps, essaie d'ailleurs de concilier ces deux points de vue.

plus de trente ans maintenant. S'éloigner de ce modèle comme le font les moteurs NoSQL implique de développer de nouvelles stratégies, de nouveaux algorithmes et de nouvelles pratiques pour la gestion des données. Il y a donc des points communs, des fertilisations croisées que nous allons détailler dans ce livre.

Comme beaucoup de nouvelles technologies qui rencontrent un certain succès, les moteurs NoSQL font l'objet de débats parfois exagérés, où les arguments manquent de précision et d'objectivité. Pour exemple, voici un extrait traduit de l'introduction du livre *Hbase, the Definitive Guide* (Lars George, O'Reilly, 2011), où l'auteur justifie l'utilisation de HBase plutôt qu'un moteur relationnel dans les cas de forte montée en charge :

« La popularité de votre site augmentant, on vous demande d'ajouter de nouvelles fonctionnalités à votre application, ce qui se traduit par plus de requêtes vers votre base de données. Les jointures SQL que vous étiez heureux d'exécuter par le passé ralentissent soudainement et ne se comportent plus de façon satisfaisante pour monter en charge. Vous devrez alors dénormaliser votre schéma. Si les choses continuent à empirer, vous devrez aussi abandonner l'utilisation de procédures stockées, parce qu'elles deviennent trop lentes. En fin de compte, vous réduisez le rôle de votre base de données à un espace de stockage en quelque sorte optimisé pour vos types d'accès.

Votre charge continuant à augmenter et de plus en plus d'utilisateurs s'enregistrant sur votre site, l'étape logique suivante consiste à prématérialiser de temps en temps les requêtes les plus coûteuses, de façon à pouvoir fournir plus rapidement les données à vos clients. Puis vous commencez à supprimer les index secondaires, car leur maintenance devient trop pénible et ralentit la base de données. Vous finissez par exécuter des requêtes qui peuvent utiliser seulement la clé primaire, et rien d'autre. »

Vous avez peut-être maintenant les yeux écarquillés, comme je les ai eus en lisant ce texte. Pour l'auteur, la solution pour optimiser les accès à un moteur relationnel est de dénormaliser, supprimer les jointures, bannir les procédures stockées et enlever les index. Or un moteur relationnel peut justement monter en charge grâce aux optimisations apportées par la normalisation, l'indexation et les procédures stockées qui évitent des allers-retours entre le client et le serveur. Ce genre d'approximations et de contrevérités sur le modèle relationnel est assez commun dans le monde NoSQL. Est-ce que, pour autant, les propos cités ici sont faux ? Pas forcément, cela dépend du contexte et c'est ça qui est important. Dans une utilisation analytique des données, qui force à parcourir une grande partie des tables, les jointures peuvent réellement poser problème, surtout si elles s'exécutent sur un moteur comme MySQL, que beaucoup d'utilisateurs du NoSQL connaissent bien parce qu'ils sont aussi des défenseurs du logiciel libre. MySQL n'avait jusqu'à très récement qu'un seul algorithme de jointure, la boucle imbriquée, qui est inefficace si le nombre de lignes à joindre est important. Quoi qu'il en soit, il est important de préciser de quel contexte on parle pour ne pas induire le lecteur en erreur.

En face, l'incompréhension est également souvent constatée, et le danger qui guette toute discussion sur le sujet est de se transformer en une guerre d'écoles, où les spécialistes et défenseurs du modèle relationnel rejettent avec mépris le mouvement NoSQL, en ignorant les raisons du développement et de l'engouement de ces systèmes qui tiennent en partie aux difficultés et limitations du relationnel.

Tout cela crée des querelles de clocher qui feront des victimes : les utilisateurs et les entreprises qui, eux, recherchent objectivement la meilleure solution de stockage et de gestion de leurs données. Ce livre se propose donc de clarifier la situation, en comparant les deux modèles, en identifiant leurs points forts et leurs faiblesses, mais aussi en essayant de débusquer ce qui se cache derrière les choix de conception des différents moteurs NoSQL. En effet, nous pensons qu'une vision d'ensemble de l'offre de ces nouveaux moteurs vous permettra de vous orienter dans vos choix de développement. Bien entendu, l'approche de cet ouvrage sera également pratique et démonstrative : nous vous expliquerons comment fonctionnent les moteurs NoSQL les plus populaires, pourquoi et comment les choisir, et comment mettre le pied à l'étrier.

À propos des exemples de code

Dans ce livre, nous allons expliquer comment manipuler des bases de données NoSQL, en présentant leurs interfaces, leurs invites de commande, ainsi que des exemples de code client. Pour ces exemples, nous avons choisi d'utiliser Python, principalement parce que c'est un langage très populaire qui compte de nombreux pilotes pour les moteurs NoSQL, et parce que la clarté de sa syntaxe le rend facile à lire et propice à l'apprentissage.

Python, un langage de choix

Peut-être pensez-vous que Python n'est pas un langage d'entreprise au même titre que C++, C# ou Java ? Détrompez-vous. C'est un langage puissant qui est de plus en plus utilisé dans les projets de grande ampleur. Le site codeeval.com, qui permet à ses membres d'évaluer leur compétence en programmation, publie chaque année une estimation de la popularité des langages de programmation, basée sur des tests de codage pour plus de 2 000 entreprises. Le résultat de l'année 2014 (http://blog.codeeval.com/codeevalblog/2014) met ainsi Python au premier rang pour la quatrième année consécutive, avec 30,3 % de popularité.

Toutes nos installations et nos exemples seront réalisés sur une machine Linux, dotée de la distribution Ubuntu Server 14.04 LTS. Python y figurant par défaut, il vous restera à installer `pip` *(Python Package Index)*, l'utilitaire de gestion de paquets Python, qui nous servira à récupérer les différents paquets comme les pilotes pour nos moteurs. Voici la procédure d'installation de `pip` sur Ubuntu.

Installez d'abord les paquets suivants à l'aide de `apt-get`, le gestionnaire de paquets de Debian et d'Ubuntu :

```
sudo apt-get install python-pip python-dev build-essential
```

Puis procédez à la mise à jour de `pip` :

```
sudo pip install --upgrade pip
```

À propos de la deuxième édition

Depuis la parution de la première édition de cet ouvrage, le paysage du NoSQL a beaucoup évolué. Cette nouvelle édition a été remaniée en de nombreux points. Nous avons allégé la partie

théorique sur les différences entre SQL et NoSQL, afin de consacrer plus de pages à une considération essentielle : la nature et le mode d'utilisation des bases de données NoSQL. Nous avons également réduit certaines portions de code, notamment celles détaillant l'installation des moteurs NoSQL, pour traiter de sujets plus importants comme les forces et les faiblesses de chacun de ces moteurs, les cas pratiques d'utilisation et leurs principales fonctionnalités. Par ailleurs, nous avons nettement augmenté la place du Big Data car il s'agit du thème dont l'actualité est la plus forte. Les moteurs NoSQL distribués et les méthodes de traitement distribué sont en pleine évolution actuellement et il est capital de comprendre les enjeux de ces développements pour traiter de larges volumes de données.

Partie I

Qu'est-ce qu'une base NoSQL ?

Le terme « NoSQL » a été inventé en 2009 lors d'un événement sur les bases de données distribuées. Le terme est vague, incorrect (certains moteurs NoSQL utilisent des variantes du langage SQL, par exemple Cassandra), mais présente l'avantage d'avoir un effet marketing et polémique certain. Dans cette partie, nous allons aborder les caractéristiques générales des moteurs NoSQL, historiquement, conceptuellement et techniquement, en regard des bases de données relationnelles, mais aussi indépendamment de cette référence.

1

Des SGBD relationnels au NoSQL

Les défenseurs du mouvement NoSQL le présentent comme une évolution bienvenue de l'antique modèle relationnel. Ses détracteurs le considèrent plutôt comme une régression. Le modèle relationnel est apparu dans les années 1970, devenant rapidement le modèle dominant, et jamais détrôné depuis, un peu comme les langages impératifs (comme C++ et Java) dans le domaine de la programmation. Dans ce chapitre, nous allons présenter un bref historique de l'évolution des bases de données informatiques, pour mieux comprendre d'où viennent les modèles en présence, pourquoi ils ont vu le jour et ont ensuite évolué.

Brève histoire des systèmes de gestion de bases de données

Le besoin d'organiser les données, potentiellement de grandes quantités de données, afin d'en optimiser la conservation et la restitution, a toujours été au cœur de l'informatique. La façon dont nous nous représentons l'ordinateur est une métaphore du cerveau humain. Il nous est évident que l'élément central du fonctionnement intellectuel est la mémoire. Sans le stock d'informations que constitue la mémoire humaine, il nous est impossible de produire le moindre raisonnement, car ce dernier manipule des structures, des éléments connus, reconnus et compris, qui proviennent de notre mémoire.

Stocker et retrouver, voilà les défis de la base de données. Ces deux éléments centraux se déclinent bien sûr en différentes sous-fonctions importantes, comme assurer la sécurité ou se protéger des incohérences. Comme nous allons le voir dans ce chapitre, ces sous-fonctions ne font pas

l'unanimité. Depuis toujours, les avis divergent en ce qui concerne les responsabilités assurées par un système de gestion de bases de données.

Ainsi, les premiers défis des SGBD furent simplement techniques et concernaient des fonctions importantes, stocker et retrouver. Un système gérant des données doit être capable de les maintenir sur une mémoire de masse, comme on disait il y a quelques décennies, et doit offrir les fonctions nécessaires pour retrouver ces données, souvent à l'aide d'un langage dédié. Les débuts furent un peu de la même veine que ce que nous voyons aujourd'hui dans le monde des bases NoSQL : une recherche parallèle de différents modèles.

Le modèle hiérarchique

Le modèle hiérarchique est historiquement la première forme de modélisation de données sur un système informatique. Son développement commença dans les années 1950.

On l'appelle « modèle hiérarchique » à cause de la direction des relations qui s'établissent uniquement du parent vers les enfants. Une base de données hiérarchique est composée d'enregistrements *(records)* qui contiennent des champs et qui sont regroupés en types d'enregistrements *(record types)*. Des relations sont créées entre types d'enregistrements parents et types d'enregistrements fils, ce qui forme un arbre hiérarchique. La différence majeure entre ce modèle et le modèle relationnel que nous connaissons maintenant, est la limitation de l'arbre : il ne peut y avoir qu'un seul arbre et les relations ne peuvent se dessiner que du parent vers l'enfant. Les choix de relations sont donc très limités. L'implémentation la plus connue du modèle hiérarchique est le moteur IMS *(Information Management System)* d'IBM, auparavant nommé ICS *(Information Control System)*, qui fut notamment développé dans le but d'assurer la gestion des matériaux de construction du programme spatial de la NASA pour envoyer des hommes sur la Lune dans les années 1960. IMS existe toujours et selon IBM, la meilleure année du moteur en termes de ventes a été 2003.

Codasyl et Cobol

En 1959, le Codasyl (*Conference on Data Systems Languages*, en français Conférence sur les langages de systèmes de traitement de données) conduit à la création d'un consortium dont l'objectif était de développer des standards de gestion de données et ce développement d'un langage d'accès à ces données. Les membres du Codasyl étaient des entreprises, des universitaires et des membres du gouvernement. Ils établirent d'abord un langage d'interrogation de données, le fameux Cobol *(COmmon Business Oriented Language)*. Ce dernier fut développé la même année à l'université de Pennsylvanie. Cobol fait l'objet d'une norme ISO qui, comme la norme SQL, évolua en plusieurs versions. C'est un langage qu'on nomme « navigationnel » : des pointeurs sont posés et maintenus sur une entité, appelée « entité courante », et le pointeur se déplace au gré des besoins vers d'autres entités. Quand on travaille avec plusieurs entités, chaque entité a son pointeur maintenu sur un article. Il s'agit d'une approche purement procédurale à l'accès aux données : parcourir les ensembles d'entités implique d'écrire des boucles (en utilisant des commandes comme `FIND FIRST` et `FIND NEXT`) qui testent des critères et traitent les données ainsi filtrées, par exemple.

Après avoir établi le langage de traitement de données, les membres du Codasyl établirent un standard de structuration des données, qu'on nomme « modèle de données réseau ». Fondamentalement,

il annonce le modèle relationnel. Le modèle réseau permet de décrire des liens entre des articles. Ces liens sont appelés des *sets*, et on peut les considérer, si on veut faire une comparaison avec le modèle relationnel, comme des tables de correspondance entre des articles, propriétaire d'un côté et type membre de l'autre. Les systèmes de gestion de bases de données réseau n'ont pas l'élégance du modèle relationnel, principalement à cause de leur langage de requête, complexe et navigationnel (Cobol).

Edgard Frank Codd

Edgar Frank Codd est un sujet britannique qui étudia les mathématiques et la chimie en Angleterre, et qui se rendit aux États-Unis en 1948 pour travailler comme programmeur pour IBM. Il obtint son doctorat de *Computer Science* à l'université du Michigan. Au milieu des années 1960, il commença à travailler comme chercheur aux laboratoires de recherche d'IBM à San Jose, en Californie. C'est dans ce laboratoire qu'il élabora l'organisation des données selon un modèle basé sur la théorie mathématique des ensembles. Il publia à ce sujet un document interne en 1969, pour exposer sa théorie au sein d'IBM. Malheureusement, les investissements effectués par la société dans le système IMS n'incitèrent pas IBM à s'intéresser à ce modèle différent. Un an plus tard, Codd publia donc un article intitulé *« A Relational Model of Data for Large Shared Data Banks »* dans *Communications of the ACM*, revue de l'association pour la machinerie informatique *(Association for Computing Machinery)*. Cette association à but non lucratif fut fondée en 1947 pour favoriser la recherche et l'innovation. Elle est toujours active aujourd'hui. L'article pose les bases du modèle relationnel en indiquant les bases mathématiques et algébriques de relations. Il est disponible à l'adresse suivante : http://www.acm.org/classics/nov95/toc.html.

Les relations

Le modèle de Codd privilégiait un système de relations basé uniquement sur les valeurs des données, contrairement à d'autres types de relations utilisant des pointeurs sur des entités (modèle réseau), et une manipulation de ces données à l'aide d'un langage de haut niveau implémentant une algèbre relationnelle, sans se préoccuper du stockage physique des données. Cette approche permet d'isoler l'accès aux données de l'implémentation physique, et même de le faire sur plusieurs niveaux à travers le mécanisme des vues, et d'exprimer des requêtes dans un langage beaucoup plus compact que de manière procédurale comme en Cobol. Il s'agit d'un langage déclaratif, algébrique, en quelque sorte fonctionnel, qui ne s'occupe pas des algorithmes informatiques de manipulation des données. A priori, cette pensée est plutôt étrange : comment concevoir qu'un langage de haut niveau, qui ne s'occupe que de décrire les résultats d'opérations sur des données sans jamais spécifier le comment, puisse être efficace. Pour s'assurer des meilleures performances, il faut normalement être au plus proche de la machine. L'histoire du modèle relationnel a montré que cette pensée était erronée dans ce cas : une description correcte de la requête déclarative, alliée à un moteur d'optimisation efficace du côté du serveur permet au contraire d'optimiser automatiquement les performances.

Le développement des moteurs relationnels

En 1974, le laboratoire de San Jose commença à développer un prototype, le System R, pour expérimenter les concepts avancés par Codd. Ils développèrent un langage de manipulation de

données nommé Sequel *(Structured English Query Language)* et un moteur en deux parties, RSS *(Research Storage System)* et RDS *(Relational Data System)*, une séparation utilisée de nos jours dans tous les moteurs relationnels sous forme de moteur de stockage et de moteur relationnel et qui permet effectivement la séparation entre la gestion physique et la gestion logique des données. System R est l'ancêtre de tous les moteurs de bases de données relationnelles contemporains et a introduit un certain nombre de concepts qui constituent encore aujourd'hui la base de ces moteurs, comme l'optimisation de requêtes ou les index en *B-tree* (arbre balancé). Comme Sequel était une marque déposée par une compagnie britannique, le nom du langage fut changé en SQL. En même temps, à l'université de Berkeley, Eugene Wong et Michael Stonebraker s'inspirèrent des travaux de Codd pour bâtir un système nommé Ingres, qui aboutira plus tard à PostgreSQL, Sybase et Informix. De son côté, vers la fin des années 1970, Larry Ellison s'inspira aussi des travaux de Codd et de System R pour développer son moteur de bases de données, Oracle, dont la version 2 fut en 1979 le premier moteur relationnel commercial. IBM transforma System R en un vrai moteur commercial, nommé SQL/DS, qui devint par la suite DB2.

Le système dominant : le modèle relationnel d'Edgar Frank Codd

À partir des années 1980, le modèle relationnel supplanta donc toutes les autres formes de structuration de données. System R avait prouvé qu'un langage déclaratif pouvait s'avérer être un excellent choix pour une interrogation performante des données. Oracle, DB2 et les descendants d'Ingres furent les implémentations qui rendirent les SGBDR populaires auprès des entreprises et des universités.

Le modèle relationnel a développé notre vision de ce qu'est ou doit être une base de données, pour des décennies : séparation logique et physique, langage déclaratif, structuration forte des données, représentation tabulaire, contraintes définies au niveau du moteur, cohérence transactionnelle forte, etc. Ces caractéristiques ont été gravées dans le marbre par Edgar Codd grâce à ce qu'on appelle « les douze règles de Codd ».

Les douze règles de Codd

Ce qu'on appelle les douze règles de Codd sont en fait numérotées de 0 à 12, il y en a donc treize. Une erreur classique de base zéro.

Elles parurent en octobre 1985 dans le magazine *ComputerWorld* dans deux articles maintenant célèbres : *« Is Your DBMS Really Relational? »* et *« Does Your DBMS Run By the Rules? »*.

Les règles de Codd

Vous trouverez ci-après celles qui nous intéressent le plus pour la suite de ce livre.

- Règle 0 – Toutes les fonctionnalités du SGBDR doivent être disponibles à travers le modèle relationnel et le langage d'interrogation.

- Règle 1 – Toutes les données sont représentées par des valeurs présentes dans des colonnes et des lignes de tables.
- Règle 3 – Une cellule peut ne pas contenir de valeur, ou exprimer que la valeur est inconnue, à l'aide du marqueur `NULL`. Il s'agit d'un indicateur spécial, distinct de toute valeur et traité de façon particulière.
- Règle 5 – Le SGBDR doit implémenter un langage relationnel qui supporte des fonctionnalités de manipulation des données et des métadonnées, de définition de contraintes de sécurité et la gestion des transactions.
- Règle 10 – Indépendance d'intégrité : les contraintes d'intégrité doivent être indépendantes des programmes clients et doivent être stockées dans le catalogue du SGBDR. On doit pouvoir modifier ces contraintes sans affecter les programmes clients.
- Règle 11 – Indépendance de distribution : la distribution ou le partitionnement des données ne doivent avoir aucun impact sur les programmes clients.
- Règle 12 – Règle de non-subversion : aucune interface de bas niveau ne doit permettre de contourner les règles édictées. Dans les faits, cette règle implique qu'il n'est possible d'interroger et de manipuler le SGBDR qu'à travers son langage relationnel.

L'ensemble de ces règles indique la voie à suivre pour les systèmes de gestion de bases de données relationnelles. Elles concernent le modèle sous-jacent des bases dites SQL et ne sont jamais totalement implémentées, à cause des difficultés techniques que cela représente.

De OLTP à OLAP

À partir des années 1980, les moteurs relationnels ont donc pris le pas sur les autres systèmes pour les besoins de tous types de données, d'abord pour les systèmes d'entreprises ou d'académies, puis auprès de développeurs indépendants pour des initiatives libres ou personnelles, comme des logiciels *shareware*, des sites web, etc. Même pour des petits besoins, des moteurs embarqués ou locaux comme SQLite (http://www.sqlite.org/) sont largement utilisés.

Assez rapidement, pourtant, un besoin différent s'est manifesté : le modèle relationnel est performant pour une utilisation purement transactionnelle, ce qu'on appelle OLTP *(Online Transactional Processing)*. Une base de données de gestion, par exemple, qu'on utilise dans les PGI (Progiciels de gestion intégrée, ERP en anglais pour *Enterprise Resource Planning*), présente une activité permanente de mises à jour et de lectures de jeux de résultats réduits. On interroge la table des factures filtrées pour un client, ce qui retourne une dizaine de lignes, on requête la table des paiements pour vérifier que ce client est bien solvable, si c'est le cas, on ajoute une facture comportant des lignes de factures, et pour chaque produit ajouté, on décrémente son stock dans la table des produits. Toutes ces opérations ont une envergure limitée dans des tables dont la cardinalité (le nombre de lignes) peut être par ailleurs importante. Mais, par les vertus d'une bonne modélisation des données et grâce à la présence d'index, chacune de ces opérations est optimisée.

Mais qu'en est-il des besoins statistiques ? Comment répondre aux demandes de tableaux de bord, d'analyses historiques voire prédictives ? Dans un PGI, cela peut être une analyse complète des tendances de ventes par catégories de produits, par succursales, par rayons, par mois, par type de clients, sur les cinq dernières années, en calculant les évolutions pour déterminer quelles

catégories de produits évoluent dans quelle région et pour quelle clientèle, etc. Dans ce genre de requête, qu'on appelle OLAP *(Online Analytical Processing)*, qui doit parcourir une grande partie des données pour calculer des agrégats, le modèle relationnel, les optimiseurs de requête des SGBDR et l'indexation ne permettent pas de répondre de manière satisfaisante au besoin.

Le schéma en étoile

Un modèle différent a alors émergé, avec une structuration des données adaptée à de larges parcours de données volumineuses. Ce modèle est orienté autour du schéma en étoile ou en flocon. Faisons la différence entre un modèle normalisé très simple, comme celui reproduit sur la figure 1-1 et un modèle OLAP en étoile, tel que celui représenté sur la figure 1-2.

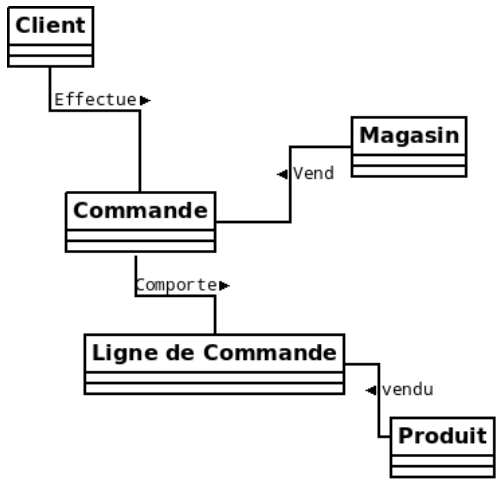


Figure 1-1
Modèle normalisé OLTP

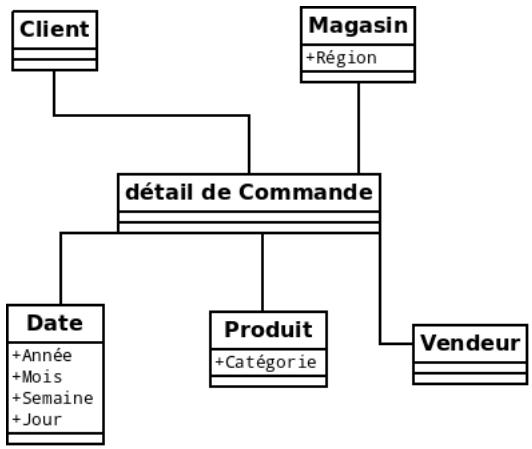


Figure 1-2
Modèle OLAP en étoile

Nous verrons plus loin en quoi consiste la normalisation. Pour l'instant, nous souhaitons simplement montrer comment les structures de données se sont modifiées suite à l'évolution des besoins des entreprises, laquelle dépend aussi de l'augmentation du volume des données manipulées par le système d'information. Le modèle OLAP a vu le jour en raison de l'augmentation du stockage de données agrégées et historiques concernant le fonctionnement et le métier des entreprises, et des besoins de requêtes globales sur ces grands volumes pour des besoins analytiques. C'est ce qu'on appelle le décisionnel ou la *Business Intelligence* : l'utilisation des données pour l'analyse et la connaissance du métier, des caractéristiques commerciales, marketing et comptables de l'entreprise.

Ce modèle, qui a aussi été formalisé par Codd et son équipe, préfigure ce qu'on appelle aujourd'hui le *Big Data*. À la base, l'interrogation du modèle OLAP se fait à travers une vision dite « cube » ou « hypercube », intégrée dans des outils d'analyse, et qui permet de manipuler les données agrégées selon un nombre théoriquement infini de dimensions, qui sont justement les axes d'analyse dont nous parlions. Nous verrons que l'augmentation exponentielle du volume des données aboutit aujourd'hui à chercher de nouvelles façons – ou en tout cas des façons complémentaires – de traiter et d'analyser les données décisionnelles et même celles en temps réel.

L'émergence du Big Data et des bases NoSQL

Les évolutions logicielles suivent assez naturellement les évolutions matérielles. Les premiers SGBD étaient construits autour de mainframes et dépendaient des capacités de stockage de l'époque. Le succès du modèle relationnel est dû non seulement aux qualités du modèle lui-même mais aussi aux optimisations de stockage que permet la réduction de la redondance des données. Avec la généralisation des interconnexions de réseaux, l'augmentation de la bande passante sur Internet et la diminution du coût de machines moyennement puissantes, de nouvelles possibilités ont vu le jour, dans le domaine de l'informatique distribuée et de la virtualisation, par exemple.

Le passage au XXI^e^ siècle a vu les volumes de données manipulées par certaines entreprises ou organismes, notamment ceux en rapport avec Internet, augmenter considérablement. Données scientifiques, réseaux sociaux, opérateurs téléphoniques, bases de données médicales, agences nationales de défense du territoire, indicateurs économiques et sociaux, etc., l'informatisation croissante des traitements en tout genre implique une multiplication exponentielle de ce volume de données qui se compte maintenant en pétaoctets (100 000 téraoctets). C'est ce que les Anglo-Saxons ont appelé le Big Data. La gestion et le traitement de ces volumes de données sont considérés comme un nouveau défi de l'informatique, et les moteurs de bases de données relationnelles traditionnels, hautement transactionnels, semblent totalement dépassés.

La solution Google

Google est probablement la société la plus concernée par la manipulation de grands volumes de données, c'est pourquoi elle a cherché et développé une solution visant à relever ces défis. Google doit non seulement gérer un volume extrêmement important de données afin d'alimenter son moteur de recherche, mais aussi ses différentes offres, comme Google Maps, YouTube, Google Groupes et bien sûr son offre d'outils comme Gmail ou Google Drive. Cela nécessite

non seulement un stockage de données très important, mais aussi des besoins de traitements sur ces volumes.

Afin de permettre le stockage de ces grands volumes de données, Google a développé il y a plus de dix ans un système de fichiers distribué nommé GoogleFS, ou GFS, système propriétaire utilisé seulement chez Google. Fin 2003, au symposium de l'ACM *(Association for Computing Machinery)* sur les principes de systèmes d'exploitation à Lake George, des ingénieurs de Google (Sanjay Ghemawat, Howard Gobioff et Shun-Tak Leung) firent une présentation de Google FS, intitulée *The Google File System* (http://research.google.com/archive/gfs.html). En 2003 déjà, le système de fichiers distribué de Google avait fait ses preuves et était intensivement utilisé en production, ce qui prouvait la viabilité de la solution technique. Le but de GFS était d'offrir un environnement de stockage redondant et résilient fonctionnant sur un *cluster* constitué d'un grand nombre de machines de moyenne puissance, « jetables » (le terme anglo-saxon est *commodity hardware*). Quelques mois plus tard, en 2004, à l'occasion du sixième symposium OSDI *(Operating System Design and Implementation)* à San Francisco, Jeffrey Dean et le même Sanjay Ghemawat de Google présentèrent un autre pilier de leur solution technique. Le titre de la présentation était *MapReduce: Simplified Data Processing on Large Clusters* (http://research.google.com/archive/mapreduce.html). Comme nous le verrons plus en détail, il s'agissait, en plus de stocker les données sur GFS, de pouvoir effectuer des traitements sur ces données de façon également distribuée et de pouvoir en restituer les résultats. Pour ce faire, les ingénieurs de Google se sont inspirés des langages fonctionnels et en ont extrait deux primitives, les fonctions `Map` et `Reduce`. À la base, `Map` permet d'effectuer un traitement sur une liste de valeurs. En le réalisant sur GFS et en regroupant les résultats grâce à la fonction `Reduce`, Google avait réussi à bâtir un environnement de traitement distribué qui lui a permis de résoudre un grand nombre de problèmes.

Hadoop, une implémentation de MapReduce

Doug Cutting, le développeur du moteur de recherche en plein texte Lucene, cherchait un moyen de distribuer le traitement de Lucene pour bâtir le moteur d'indexation web libre Nutch. S'inspirant de la publication sur GFS, il créa avec d'autres contributeurs du projet une implémentation libre en Java nommée d'abord NDFS *(Nutch Distributed File System)*. Afin de permettre un traitement distribué des données accumulées par Nutch, Doug Cutting entama alors une implémentation libre de MapReduce en Java, qu'il appela Hadoop, du nom de l'éléphant doudou de son fils. Nous en reparlerons plus loin ainsi que de sa mise en œuvre. NDFS fut ensuite renommé HDFS *(Hadoop Distributed FileSystem)*. D'abord financé par Yahoo! où Doug Cutting était à l'époque employé, donné à la fondation Apache ensuite, Hadoop est devenu un projet d'une grande popularité, implémenté par de plus en plus de sociétés, jusqu'à Microsoft qui a annoncé début 2012 son soutien au projet, et la disponibilité d'Hadoop sur son service de cloud, Windows Azure, ainsi que sur Windows Server. Au Hadoop Summit 2012, Yahoo! a indiqué que 42 000 serveurs tournaient avec Hadoop, et que leur plus gros cluster Hadoop était composé de 4 000 machines, bientôt 10 000 pour la sortie de Hadoop 2.0. Par ailleurs, en juin 2012, Facebook a annoncé sur son site que leur installation d'HDFS atteignait le volume physique de 100 pétaoctets (http://www.facebook.com/notes/facebook-engineering/under-the-hood-hadoop-distributed-filesystem-reliability-with-namenode-and-avata/10150888759153920).

BigTable, encore Google !

En 2004, Google commença à bâtir un système de gestion de données basé sur GFS : BigTable. En 2006, encore à l'occasion du symposium OSDI, des ingénieurs de Google firent une présentation nommée *BigTable: A Distributed Storage System for Structured Data*. BigTable ressemble à une gigantesque table de hachage distribuée qui incorpore des mécanismes permettant de gérer la cohérence et la distribution des données sur GFS. Une fois de plus, cette présentation de Google inspira la création d'implémentations libres, dont la plus notable est HBase. Cette dernière a été développée à l'origine par la société Powerset, active dans le domaine de la recherche en langage naturel. Leur besoin était de traiter de larges volumes d'informations pour développer leur moteur de recherche. Se basant sur Hadoop et HDFS, ils ajoutèrent la couche correspondant à BigTable. À partir de 2006, deux développeurs de Powerset, Michael Stack et Jim Kellerman, également membres du comité de gestion d'Hadoop, furent dédiés à plein temps au développement de Hbase, projet libre qui commença comme une contribution à Hadoop et devint un projet indépendant de la fondation Apache en janvier 2008. En juillet 2008, Microsoft racheta Powerset pour intégrer leur technologie dans son moteur de recherche Bing. Après deux mois d'inactivité, Microsoft permit aux développeurs de Powerset de continuer leur contribution à plein temps au projet libre.

HBase est devenu depuis un des projets phares du monde NoSQL. Il s'agit d'un système de gestion de données orienté colonnes (voir page 21) destiné à manipuler de très larges volumes de données sur une architecture totalement distribuée. Il est utilisé par des acteurs majeurs du Web, comme Ebay, Yahoo! et Twitter, de même que par des sociétés telles qu'Adobe. Pour consulter une liste non exhaustive des sociétés utilisant Hbase, rendez-vous à l'adresse suivante : http://wiki.apache.org/hadoop/Hbase/PoweredBy.

L'influence d'Amazon

En octobre 2007, Werner Vogels, *Chief Technical Officer* d'Amazon annonce sur son blog, *All Things Distributed* (http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html), qu'Amazon va présenter un papier au 21e symposium sur les principes des systèmes d'exploitation *(Symposium on Operating Systems Principles)* organisé par l'ACM. Cette présentation concerne une technologie qu'Amazon a nommé Dynamo (http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html) et comme celle de Google, sa lecture est très intéressante. Dynamo consiste en un entrepôt de paires clé-valeur destiné à être lui aussi totalement distribué, dans une architecture sans maître. Dynamo utilise un certain nombre de technologies typiques du monde NoSQL que nous détaillerons au chapitre 3 consacré aux choix techniques du NoSQL. Ceci dit, si les moteurs NoSQL implémentent ces technologies, c'est aussi grâce au papier d'Amazon. Cette communication a eu une grande influence dans le monde NoSQL et plusieurs projets se sont basés sur les technologies présentées pour bâtir leur solution libre. Dynamo, comme BigTable de Google, est resté la propriété d'Amazon et le code n'est donc pas disponible.

Un mouvement pragmatique

Voici la traduction d'un passage intéressant de l'article de Werner Vogels :

« Nous avons beaucoup de chance que le papier ait été sélectionné pour publication par le SOSP ; très peu de systèmes réellement utilisés en production ont été présentés durant ces conférences, et de ce point de vue, il s'agit d'une reconnaissance du travail fourni pour créer un système de stockage capable réellement de monter en charge de façon incrémentielle, et dans lequel les propriétés les plus importantes peuvent être configurées de façon appropriée. »

Dans les nouvelles tendances de l'informatique, il y a toujours une part de réponse à des besoins réels de développement, d'invention, de sophistication technique, et une part de mode, d'investissement de grands acteurs du domaine pour développer ou renforcer leur présence, et évidemment pour ne pas se laisser distancer en manquant un rendez-vous avec le futur. Le NoSQL fait partie de ces phénomènes, mais il présente aussi l'avantage d'être un mouvement très pragmatique. Comme nous le verrons, nombre de moteurs NoSQL sont développés par les entreprises elles-mêmes, qui les utilisent en production afin de répondre à leurs propres besoins.

Apache Cassandra

Avinash Lakshman, un des développeurs de Dynamo, fut recruté par Facebook pour créer un entrepôt de données destiné à mettre en place la fonctionnalité de recherche dans les boîtes aux lettres des comptes Facebook, qui permet aux utilisateurs d'effectuer des recherches parmi tous les messages reçus. En collaboration avec un autre ingénieur, Prashant Malik, précédemment Senior Engineer chez Microsoft, il développa un entrepôt orienté colonnes totalement décentralisé, mélangeant donc les technologies présentées par Google et Amazon. Le résultat, Cassandra, fut publié en projet libre sur Google Code en 2008, puis devint un projet Apache en mars 2009. Au fil des versions, Apache Cassandra est devenu un produit très intéressant et représentatif de ce que le mouvement NoSQL peut offrir de meilleur. Sur son site (http://cassandra.apache.org/), on apprend que Cassandra est utilisé par des acteurs comme Twitter, Reddit et Cisco, et que les installations de production Apple représentent 75 000 nœuds et 10 pétaoctets de données, et celles de Netflix, 2 500 nœuds, 420 téraoctets de données et un billion[2] d'opérations par jour.

Ironiquement, lorsque Facebook évalua une nouvelle méthode de stockage pour ses fonctionnalités de messagerie, les tests effectués avec un cluster de machines MySQL, un cluster Cassandra et un cluster HBase les décidèrent à choisir HBase, principalement pour des raisons de facilité de montée en charge et de modèle de cohérence de données (http://www.facebook.com/notes/facebook-engineering/the-underlying-technology-of-messages/454991608919). Mais ne vous méprenez pas, ce choix a été fait par Facebook en 2010. Depuis, Cassandra a vécu de très importants changements, aussi bien en termes de performance que d'architecture. Il n'est pas sûr que le même test aujourd'hui produirait les mêmes résultats.

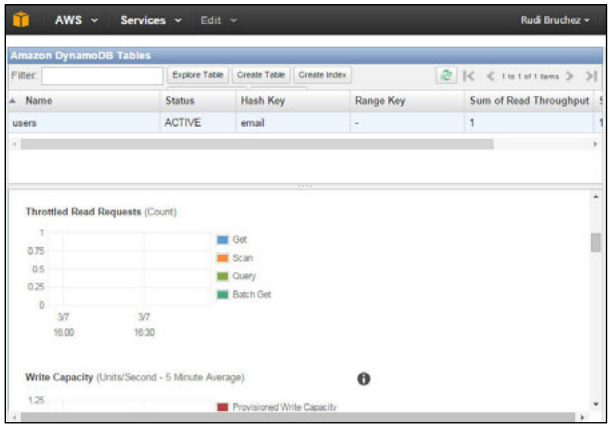
HBase et Cassandra sont des outils à la fois proches et très différents comme nous le verrons plus en détail dans ce livre.

2. Soit un trillion en anglais.

DynamoDB

Amazon Dynamo, nous l'avons vu, a inspiré la création de plusieurs moteurs NoSQL comme Cassandra, Riak ou le projet Voldemort de LinkedIn. Lorsqu'il fut présenté en 2007, Dynamo ne pouvait pas être utilisé directement par d'autres entreprises qu'Amazon. Il était mis en œuvre dans les offres de services web d'Amazon comme S3 *(Simple Storage Service)*, ce qui le rendait indirectement disponible pour les utilisateurs de ces services. Toutefois, en janvier 2012, Amazon a décidé de rendre disponible son moteur à travers une offre cloud nommée DynamoDB (http://aws.amazon.com/fr/dynamodb/), qui n'est pas exactement le même moteur que Dynamo, mais offre une technologie semblable. Un exemple de table géré à travers le service web d'Amazon est représenté sur la figure suivante.

Figure 1-3
Interface d'administration de DynamoDB



Dans un article publié sur son blog en juin 2012 (http://www.allthingsdistributed.com/2012/06/amazon-dynamodb-growth.html), Werner Vogels annonce que de tous les services d'Amazon, DynamoDB est l'offre ayant eu la croissance la plus rapide, et qu'en juin 2012 elle avait déjà dépassé les prévisions d'Amazon pour l'année 2012. DynamoDB permet de gérer des données très rapidement et de monter en charge à travers le cloud d'Amazon sans s'occuper des détails techniques ou matériels. Il met à disposition une API pour accéder aux données à l'aide de langages comme Java, .NET ou PHP. C'est une tendance du NoSQL particulière : l'offre cloud qui représente les avantages de la simplicité et de l'oubli des contraintes matérielles, mais comporte ses désavantages, comme le stockage des données sur des serveurs appartenant à une société tierce, qui plus est américaine, et la dépendance à des services qui ne sont pas exempts de risques de défaillance générale, comme on a pu l'expérimenter quelques fois, même auprès des fournisseurs les plus sérieux. On se souvient notamment qu'en février 2013, Microsoft Azure fut indisponible durant deux jours à cause de l'expiration d'un certificat SSL…

Les évolutions du Big Data

Hadoop a eu un succès fulgurant et il est toujours aujourd'hui la bannière du Big Data. Mais ce n'est qu'une étape, déjà marquée historiquement par la sortie d'une version réarchitecturée

d'Hadoop, appelée Hadoop Yarn ou MapReduce 2 (MRv2). Nous reviendrons sur les détails techniques de ce changement important, l'essentiel étant que la nouvelle architecture devient le socle pour un grand nombre de traitements distribués. Là où la génération précédente d'Hadoop était limitée à un traitement de type Map Reduce en mode batch, Hadoop Yarn permet de diversifier les types de traitement sur de très larges volumes de données, ce qui se révèle notamment très utile de nos jours pour développer des systèmes de traitement en quasi temps réel sur des architectures Big Data. Les outils émergents du Big Data d'aujourd'hui comme Apache Storm ou Apache Spark tournent sur Hadoop Yarn. Nous allons bien sûr parler dans cet ouvrage des importantes évolutions de ces technologies.

Le mouvement NoSQL est-il une avancée ?

On l'a vu dans notre bref historique, l'idée de bases de données non relationnelles n'est pas nouvelle. En fait, le modèle relationnel peut être considéré comme une innovation de rupture qui a supplanté les anciens modèles. Est-ce que pour autant le mouvement NoSQL représente un retour en arrière, comme certains peuvent le dire ? La victoire du relationnel provient de son intelligence et de ses capacités à répondre aux défis du moment, en permettant d'optimiser le stockage et le traitement des données à une époque où les capacités matérielles étaient limitées. Une base de données relationnelle est idéale pour assurer d'excellentes performances et l'intégrité des données sur un serveur dont les ressources sont par définition limitées. Pour réaliser cela, un moteur relationnel offre la capacité de construire un modèle de données, de lui adjoindre des structures physiques comme des index, et d'utiliser un langage d'extraction, le SQL. Cela requiert beaucoup d'efforts et de technicité dans la conception. Les SGBDR sont nés à une époque où les outils devaient être compris et bien utilisés, et où le rythme de développement était plus lent. Cette technicité se perd, les principes et l'implémentation des SGBDR ne s'enseignent pas assez dans les écoles d'informatique, et les développeurs qui s'attaquent à la gestion des données n'ont pas suffisamment conscience des enjeux et des méthodologies spécifiques à ces outils.

De plus, l'avènement de l'informatique distribuée a changé la donne : il ne s'agit plus de faire au mieux par rapport au matériel, mais d'adapter les contraintes logicielles aux possibilités d'extension offertes par la multiplication des machines. Et cela provoque un effet d'entraînement. La distribution permet l'augmentation du volume des données, et l'augmentation du volume des données entraîne un besoin accru de distribution. On sait tous que 90 % du volume de données actuelles a été créé durant les deux dernières années. Une étude d'EMC (http://france.emc.com/index.htm, l'une des grandes entreprises du stockage et du cloud) en 2011 a prévu une multiplication par cinquante du volume de données informatiques entre 2011 et 2020. On n'est évidemment plus du tout dans le domaine d'action des bases de données relationnelles, en tout cas pas des SGBDR traditionnels du marché actuel.

L'une des expressions les plus souvent prononcées par les acteurs du NoSQL est *Web scale*. Un moteur NoSQL est conçu pour répondre à des charges de la taille des besoins du Web. Les besoins ont changé, les types de données manipulées ont aussi évolué dans certains cas, de même que les ressources matérielles.

Les besoins se sont modifiés lorsque les sites web dynamiques ont commencé à servir des millions d'utilisateurs. Par exemple, les moteurs de recherche doivent indexer des centaines de millions de

pages web et offrir des capacités de recherche plein texte sophistiquées et rapides. Ces données sont semi-structurées, et on ne voit pas bien comment on pourrait utiliser un moteur relationnel, qui impose des types de données strictes et atomiques, pour réaliser de telles recherches.

À l'arrivée d'une nouvelle technologie, la tendance humaine est de résister au changement si la technologie actuelle est fermement ancrée, ou au contraire de s'enthousiasmer au-delà du raisonnable si on n'est pas familier avec l'existante, ou si on ne l'aime pas. Le mouvement NoSQL constitue une avancée dès lors que l'on sait en profiter, c'est-à-dire l'utiliser à bon escient, en comprenant sa nature et ses possibilités. C'est justement le but de cet ouvrage. Les capacités de stockage, les besoins de volumétrie, le développement rapide, les approches orientées service, ou encore l'analyse de données volumineuses sont autant de paramètres qui conduisent à l'émergence d'une nouvelle façon, complémentaire, de gérer les données.

La nébuleuse NoSQL

Aujourd'hui, le mot NoSQL est connu de presque tous les informaticiens. Ce mouvement a en effet vécu une nette progression au cours de la dernière décennie. À la parution de la première édition de ce livre, son utilisation était encore réservée à quelques entreprises innovantes. Désormais, le mouvement est bien enclenché, et quelques sociétés de services françaises se spécialisent déjà dans le NoSQL. De nombreuses grandes sociétés, notamment celles présentes sur le Web ou dans le domaine des télécommunications, ou encore certains services publics, s'interrogent et recherchent des techniques leur permettant de monter en charge ou de trouver des solutions à leurs problèmes de traitement de données volumineuses. Nous en verrons un exemple dans le chapitre 16 présentant une étude de cas. Mais, sous la bannière NoSQL, les choix techniques et les acteurs sont divers. Ce n'est pas du tout la même chose que de chercher à traiter en un temps raisonnable des centaines de téraoctets de données, ou de vouloir optimiser un traitement de données en mémoire. Les outils sont donc parfois assez différents. Ils restent néanmoins gouvernés par quelques principes communs, ce qui permet de les regrouper sous le terme de NoSQL.

Tentatives de classement

Avant de voir ce qui regroupe ces outils, essayons d'abord de les distinguer. Quels sont les grands groupes de moteurs NoSQL, et comment les classer ?

Classement par usage

Nous pouvons d'abord tenter de les différencier suivant leur usage, c'est-à-dire selon le type de problématique auquel ils répondent. Nous en avons identifié quatre, que voici.

- **Amélioration des performances** : certains moteurs NoSQL ont pour but d'augmenter au maximum les performances de la manipulation des données, soit pour offrir un espace de cache en mémoire intermédiaire lors du requêtage de SGBDR, soit en tant que SGBD à part entière, qu'il soit distribué ou non. Cet objectif est généralement atteint par trois mécanismes : l'utilisation de la RAM – et nous parlons donc de bases de données en mémoire, la simplification du modèle de données en paires clé-valeur et la distribution du traitement sur les nœuds d'un cluster.

- **Assouplissement de la structure** : pour s'affranchir de la rigidité du modèle relationnel, les moteurs NoSQL simplifient la plupart du temps la structure des données (utilisations de schémas souples comme le JSON, relâchement des contraintes, pas d'intégrité référentielle entre des tables, pas de schéma explicite au niveau du serveur).
- **Structures spécifiques** : certains moteurs NoSQL sont dédiés à des besoins spécifiques, et implémentent donc une structure et des fonctionnalités focalisées sur un cas d'utilisation. Citons par exemple les moteurs orientés graphes, ou les moteurs de recherche plein texte qui offrent également des fonctionnalités proches d'un SGBD, comme Apache Solr ou ElasticSearch.
- **Volumétrie** : l'un des aspects importants des moteurs NoSQL est leur capacité à monter en charge. C'est sans doute même la raison première de la création du mouvement NoSQL. Supporter des volumétries importantes passe par une distribution du stockage et du traitement. C'est une distinction importante que nous approfondirons par la suite. Hadoop est un système de distribution du traitement colocalisé avec un système de distribution de stockage. La distribution du traitement est très importante dans un contexte analytique et dans la plupart des applications Big Data. Le stockage distribué est soit réalisé par des fichiers plats sur un système de fichiers distribués, soit par un moteur de base de données distribué comme Cassandra ou Hbase, conçu pour fonctionner sur un large cluster de machines.

  La distribution peut être centralisée ou décentralisée. Les systèmes centralisés, qui s'inspirent souvent de Google, comportent un maître qui coordonne le stockage et l'aspect transactionnel. Les systèmes décentralisés mettent chaque nœud au même niveau, ce qui implique que chaque nœud comporte un gestionnaire de données et un gestionnaire de cluster, de façon à pouvoir assurer la gestion des données bien sûr, mais aussi pour connaître et maintenir la topologie du cluster, et pouvoir répondre aux requêtes des clients.

La figure suivante représente une carte de ces différents types d'usage.

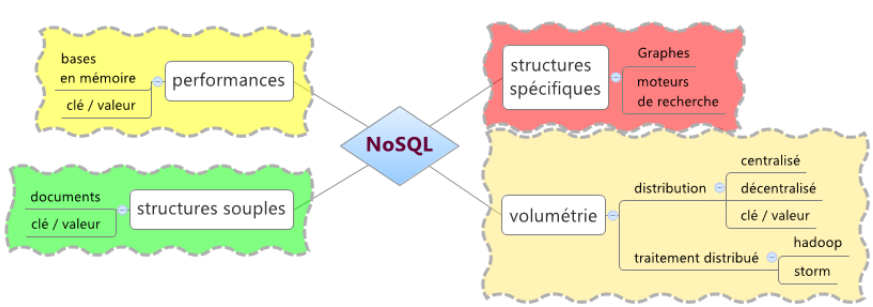


Figure 1-4
Les types d'usage des moteurs NoSQL

Classement par schéma de données

On peut aussi répertorier les moteurs SQL par le schéma ou la structure des données qu'ils manipulent. On distingue dans ce cas cinq modèles.

- **Paires clé-valeur** : les moteurs NoSQL les plus simples manipulent des paires clés valeur, ou des tableaux de hachage, dont l'accès se fait exclusivement par la clé. Il s'agit de moteurs en mémoire comme Redis ou de systèmes distribués inspirés de Dynamo, comme Riak. Ces moteurs offrent des fonctionnalités simplifiées, souvent une moins grande richesse fonctionnelle, en termes de requêtes par exemple, et d'excellentes performances grâce à leur modèle d'accès simplifié. Un système de paires clé-valeur est relativement facile à implémenter : le pattern d'accès par la clé ne nécessite pas de moteur de requête complexe, et ce point d'entrée unique permet de soigner les performances.
- **Les moteurs orientés documents** : le format de sérialisation et d'échange de données le plus populaire est aujourd'hui le JSON *(JavaScript Object Notation)*. Il permet d'exprimer un document structuré qui comporte des types de données simples, mais aussi des listes et des paires clé-valeur, sous une forme hiérarchique. Il est idéal pour représenter des données structurées ou semi-structurées. Il est donc logique de voir apparaître des moteurs dont le format natif de stockage est le JSON. C'est le cas du très populaire MongoDB, mais aussi de CouchDB ou Couchbase Server. La nuance entre moteurs paires clé-valeur et moteurs orientés documents a tendance à s'estomper, parce qu'au fur et à mesure de leur évolution, les premiers intègrent souvent un support du JSON. C'est notamment le cas avec Riak qui depuis sa version 2 supporte l'indexation secondaire, donc sur des documents JSON dans la partie valeur de ses données.
- **Les moteurs orientés colonnes** : inspirés par Google BigTable, plusieurs moteurs NoSQL implémentent une structure proche de la table, dont nous avons déjà parlé et que nous avons définie comme une table de hachage distribuée. Contrairement aux moteurs orientés documents, les données sont ici clairement représentées en lignes et séparées par colonnes. Chaque ligne est identifiée uniquement par une clé, ce qu'on appelle dans le modèle relationnel une clé primaire, et les données de la ligne sont découpées dans des colonnes, ce qui représente un niveau de structuration plus fort que dans les deux modèles précédents. Plus utilisés pour des volumétries importantes, Cassandra ou Hbase appartiennent à cette catégotrie de moteurs.
- **Index inversé** : un index inversé est une correspondance entre un terme, ou du contenu, et sa position dans un ensemble de données, par exemple un document ou une page web. Google utilise un index inversé pour répondre aux recherches sur son moteur. Il existe des moteurs de recherche sur document qu'on apparente au mouvement NoSQL, comme ElasticSearch ou Solr. Ils sont tous deux basés sur Lucene, le moteur d'indexation en index inversé libre. Pourquoi les considérer comme des moteurs de bases de données ? Parce que par-dessus Lucene, ils permettent de manipuler une structure JSON, de la chercher et de la restituer. C'est pratique, parce que cela permet à l'utilisateur de travailler avec une structure de données semblable à un moteur orienté documents, et de profiter d'excellentes capacités de requêtage grâce au moteur de recherche.
- **Structures particulières** : il nous reste juste à regrouper les moteurs NoSQL qui ne rentrent pas dans les catégories précédentes, comme les moteurs orientés graphe, qui utilisent des représentations de données particulières et des méthodes de stockage natives.

Peut-on trouver des points communs entre les moteurs NoSQL ?

Avec l'évolution des moteurs NoSQL, il devient pratiquement impossible de trouver des points communs qui réunissent tous ces moteurs et qui donnent un sens taxinomique au terme NoSQL lui-même. Certaines caractéristiques sont présentes dans presque tous les moteurs mais il demeure toujours des exceptions. Voici quelques-uns de ces points communs.

- **Le schéma implicite**. Alors que dans les moteurs relationnels, le schéma des données doit être prédéfini explicitement au niveau du serveur, dans la quasi-totalité des moteurs NoSQL, le moteur ne contrôle pas le schéma des données et c'est à l'application cliente de structurer correctement ce qu'elle veut manipuler dans la base de données. Nous verrons cela plus en détail dans la suite de l'ouvrage, mais il y a aujourd'hui un contre-exemple de taille : Cassandra qui, à partir de sa version 2 sortie en 2014, impose un schéma fort, prédéfini au niveau du serveur.
- **L'absence de relation**. Que ce soient des tableaux ou des collections, les ensembles de données stockées dans les moteurs NoSQL n'ont pas de relations les unes avec les autres, à l'inverse d'un un moteur relationnel comme son nom l'indique. Vous ne pouvez donc pas créer de référence au niveau du serveur entre un élément d'une collection et plusieurs éléments d'une autre collection. Aucun mécanisme ne le permet. Chaque collection est donc indépendante et c'est un critère important pour assurer une distribution facilitée des données. On peut trouver quelques contre-exemples structurels ou fonctionnels : ainsi, les moteurs orientés graphe implémentent nécessairement des relations entre les nœuds, c'est l'exemple structurel ; et des moteurs de requête pour le Big Data, comme Hive, permettent d'exprimer dans la requête des clauses de jointure, qu'ils vont résoudre derrière le rideau en requêtant les différents ensembles de données, c'est l'exemple fonctionnel.
- **Le langage SQL**. Malheureusement pour le nom NoSQL lui-même, l'absence de langage SQL n'est pas du tout un critère de reconnaissance des moteurs NoSQL. De nombreux moteurs SQL implémentent maintenant un langage déclaratif proche du SQL pour la manipulation des données : CQL en Cassandra, N1QL dans Couchbase Server, par exemple.
- **Le logiciel libre**. Nous l'avons vu, beaucoup de bases NoSQL sont des logiciels libres, et participent au mouvement du libre. Le logiciel libre a maintenant prouvé ses qualités. Une grande partie des serveurs présents sur Internet tournent sur des environnements totalement libres, Linux bien sûr, mais aussi Apache, Tomcat, nginx, MySQL et tant d'autres. Le libre n'est plus synonyme d'amateurisme, et même Microsoft supporte de plus en plus le libre, à travers des développements comme les pilotes PHP et ODBC sur Linux pour SQL Server, le support de Linux sur le cloud Azure, etc. Certaines entreprises comme Facebook, ou pour rester français, Skyrock (voir l'étude de cas chapitre 16), ont la culture du libre et mettent à disposition de la communauté leurs développements internes, sans parler des contributions qu'ils font, financièrement ou en temps de développement aux grands projets libres. Le mouvement NoSQL est donc naturellement un mouvement du logiciel libre. Les éditeurs de SGBD traditionnels, poussés par l'engouement NoSQL, ont des attitudes diverses sur le sujet. Oracle, par exemple, propose un entrepôt de paires clé-valeur distribué nommé simplement Oracle NoSQL Database (http://www.oracle.com/technetwork/products/nosqldb/overview/index.html), avec une version communautaire librement téléchargeable et une version Entreprise qui comportera des améliorations payantes. C'est aussi le choix d'acteurs du monde NoSQL comme Basho, qui développe Riak, ou 10gen avec MongoDB. Des versions Entreprise ou des services additionnels sont payants.

D'autres moteurs sont totalement libres, comme HBase ou Cassandra. Un certain nombre de sociétés se sont créées autour de ces produits pour vendre des services, de l'hébergement ou de la formation, telles que Datastax (http://www.datastax.com/) pour Cassandra ou Cloudera pour la pile Hadoop, HDFS et HBase. Microsoft a annoncé vouloir supporter Hadoop sur Windows et Azure début 2012, notamment en relation avec les fonctionnalités de décisionnel de leur moteur SQL Server. Le service s'appelle maintenant HDInsight (http://www.microsoft.com/sqlserver/en/us/solutions-technologies/business-intelligence/big-data.aspx). On a l'habitude de ces noms très marketing de la part de Microsoft.

Pour toutes ces raisons et pour leur excellente intégration dans les environnements de développement libre, les moteurs NoSQL sont maintenant un choix intéressant de gestion des données pour des projets de développement basés sur des outils libres, dans des langages comme PHP ou Java, et dans tous types d'environnements, également sur les clouds de Microsoft (par exemple, une installation de MongoDB est disponible pour Windows Azure) ou d'Amazon.

2

NoSQL versus SQL : quelles différences ?

S'il y a du NoSQL – qui signifie aujourd'hui, après quelques hésitations, diplomatiquement *Not Only SQL* –, cela veut dire qu'il y a donc aussi du SQL. Quelle est alors la différence entre les deux approches ? Sont-elles si éloignées l'une de l'autre et réellement ennemies ? En quoi le mouvement NoSQL s'écarte-t-il du modèle relationnel et quels sont les choix techniques qui le caractérisent ? Voici les questions auxquelles nous allons tenter de répondre dans ce chapitre.

Les principes du relationnel en regard du NoSQL

Dans le chapitre précédent, nous avons vu d'où venait historiquement le modèle relationnel, et le rôle important qu'a joué Edgar Codd, son créateur, dans l'histoire des bases de données. Voyons maintenant brièvement quelles sont les caractéristiques techniques des SGBDR, ce qui nous permettra de mieux comprendre la différence apportée par le mouvement NoSQL.

Tout d'abord, le modèle relationnel est basé sur le présupposé qu'un modèle mathématique peut servir de base à une représentation des données dans un système informatique. Cela se retrouve dans la conception même des entités que sont les tables des bases de données relationnelles. Elles sont représentées comme des ensembles de données et représentées sous forme de table. Il y a une séparation totale entre l'organisation logique des données et l'implémentation physique du moteur. Les données sont très structurées : on travaille dans le modèle relationnel avec des attributs fortement typés et on modélise la réalité pour la faire entrer dans ce modèle relativement contraignant. Ce système offre l'avantage de pouvoir ensuite appliquer des opérations algébriques

et logiques sur ces données, mais en contrepartie force des données parfois plus complexes à se plier à cette structure prédéfinie.

Cette approche, qui prend sa source dans des modèles théoriques et s'occupe ensuite de l'implémentation physique, est totalement différente du mouvement NoSQL qui se base sur des expérimentations pratiques pour répondre à des besoins concrets. Nous avons vu l'histoire de Dynamo d'Amazon, où le développement a été orienté vers la résolution d'une problématique particulière à l'aide d'un certain nombre de technologies déjà existantes et dont le mariage n'allait pas de soi de prime abord. Beaucoup de moteurs NoSQL utilisent maintenant les technologies de Dynamo parce qu'Amazon a démontré que ces solutions étaient viables pour la simple et bonne raison que cet outil est utilisé avec succès sur le panier d'achat du site Amazon. Il s'agit donc ici plus d'un travail d'ingénieur que d'un travail de chercheur ou de mathématicien.

Les structures de données

Entre le relationnel et le NoSQL, il y a des conceptions fondamentalement différentes de la structure des données. En fait, dans le NoSQL lui-même, il existe différentes façons de considérer la donnée.

Le modèle relationnel se base sur une structuration importante des données. Les métadonnées sont fixées au préalable, les attributs sont fortement typés, et selon les principes édictés par Codd, la normalisation correcte des structures implique qu'il n'y ait aucune redondance des données. Les entités ont entre elles des relations, ce qui permet d'avoir un modèle totalement interdépendant.

Dans le modèle relationnel, la métaphore de la structure, c'est le tableau : une suite de lignes et de colonnes avec sur la première ligne, le nom des colonnes en en-tête. Cela signifie que les métadonnées font partie de l'en-tête et ne sont bien sûr pas répétées ligne par ligne. Cela veut dire également que chaque donnée atomique est stockée dans une cellule, et qu'on peut la retrouver en indiquant le nom de la colonne et l'identifiant de la ligne.

Nous avons vu dans le chapitre précédent les différentes structures de données utilisées dans le monde NoSQL, comme les paires clé-valeur ou les documents JSON.

L'agrégat

Parlons un instant de l'idée d'agrégation. Nous reprenons ici les idées de Pramod J. Sadalage et Martin Fowler dans leur livre *NoSQL Distilled*[3], qui permettent bien de comprendre les différences essentielles de structuration des données entre les bases de données relationnelles et le NoSQL.

Les auteurs soulignent justement que c'est un concept intéressant pour comprendre les structures des bases NoSQL. L'idée provient d'Éric Evans et de son livre *Domain-Driven Design*[4] sur la conception pilotée par le domaine. L'un des concepts de cet ouvrage est l'agrégat, à savoir une collection d'objets liés par une entité racine, nommée la racine de l'agrégat. L'agrégat permet de créer une unité d'information complexe qui est traitée, stockée, échangée de façon atomique. De

3. Pramod J. Sadalage et Martin Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, Addison Wesley, 2012.

4. Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison Wesley, 2003.

l'extérieur, on ne peut faire référence à l'agrégat que par sa racine. En termes de programmation, c'est un concept intéressant parce qu'il permet d'assurer l'unité et l'intégrité d'un bloc d'information en identifiant une racine, on pourrait dire une clé, et en permettant la référence uniquement sur cette clé. C'est une notion qui permet également de modéliser des éléments qu'on trouve dans la réalité. Par exemple, un magazine peut être vu comme une unité, qui contient des pages, des articles, des illustrations, un contenu riche et complexe, mais qui n'existe pas indépendamment du magazine lui-même. Pour y faire référence, vous vous référez au magazine. Pour lire un article, vous devez prendre le magazine et l'ouvrir. La racine représente donc le point d'entrée à ce bloc d'information.

Une base de données orientée documents stocke des données sous forme d'agrégats. Le document JSON contient un bloc d'information, une clé qui permet de l'identifier uniquement dans la collection des documents, et la manipulation du contenu du document se fait à la base en extrayant le document du moteur pour le manipuler dans son intégralité et le stocker à nouveau dans sa totalité.

De ce point de vue, une base de données relationnelle ne structure pas du tout ses données en agrégats. Toute la structure des tables et des relations est à plat, et lorsque nous effectuons une requête SQL, comme celle-ci :

```
SELECT
  c.ContactId, c.Nom, c.Prenom, SUM(f.MontantHT) as TotalFactureHT,
  COUNT(i.InscriptionId) as NbInscriptions
FROM Contact.Contact c
JOIN Inscription.Inscription i ON c.ContactId = i.ContactId
JOIN Inscription.InscriptionFacture inf ON i.InscriptionId = inf.InscriptionId
JOIN Inscription.Facture f ON inf.FactureCd = f.FactureCd
JOIN Stage.Session s ON i.SessionId = s.SessionId
GROUP BY c.ContactId, c.Nom, c.Prenom;
```

nous exprimons dans la requête un agrégat que nous formons à partir des données des tables. Nous prenons la décision de partir d'une table spécifique (dans ce cas, le contact) et nous exprimons par les jointures l'agrégat dont cette table est la racine (ici, on récupère ses inscriptions et le montant de ses factures). On peut donc, à partir d'un modèle relationnel, créer dynamiquement les agrégats souhaités à partir de n'importe quelle racine, de n'importe quel point d'entrée. La requête suivante retourne également le nombre d'inscriptions et le montant des factures, mais cette fois-ci par rapport à une session de formation, donc en partant d'une racine différente.

```
SELECT s.SessionId, s.DateDebut, SUM(f.MontantHT) as TotalFactureHT,
  COUNT(i.InscriptionId) as NbInscriptions
FROM Stage.Session s
JOIN Inscription.Inscription i ON s.SessionId = i.SessionId
JOIN Inscription.InscriptionFacture inf ON i.InscriptionId = inf.InscriptionId
JOIN Inscription.Facture f ON inf.FactureCd = f.FactureCd;
```

L'agrégat est une unité d'information qui va être manipulée par un programme. C'est quelque chose qui sera exprimé par exemple en un objet ou une collection d'objets dans un langage orienté objet.

La centralité de la donnée

Cette souplesse du SGBDR lui permet de servir plusieurs besoins applicatifs. Comme chaque application a besoin d'un agrégat différent, la structure plate d'une base de données relationnelle permet de reconstituer les agrégats demandés par chaque application. Par exemple, une base de données de gestion d'entreprise va permettre de satisfaire les applications des différents services : la logistique, la comptabilité, la relation client, ou la gestion des stocks, qui voudront manipuler les données par des agrégats sur les commandes, les factures, l'identité du client, le produit.

En revanche, si l'agrégat est déjà formé dans la structure de la base de données, comme c'est le cas dans un moteur orienté documents, il sera plus enclin à satisfaire un besoin applicatif, mais pas tous.

Il y a donc dans les bases de données relationnelles une centralité plus forte de la donnée par rapport aux applications, et une seule base de données pourra être le point de convergence de plusieurs processus d'application de l'entreprise. Martin Fowler l'explique de la façon suivante : les bases de données relationnelles sont des outils d'intégration et ils ont été utilisés en développement comme tels. Cette capacité de centralisation a été exploitée par les projets informatiques pour faire converger les diverses applications client vers un seul point.

Est-ce une bonne ou une mauvaise chose ? Un moteur de base de données est-il un outil d'intégration ? En tout cas, on a pris l'habitude de le considérer ainsi, et lorsqu'on entreprend de choisir un moteur de base de données, on pense généralement à un seul moteur pour répondre à tous les besoins.

Si on abandonne l'idée du moteur de base de données unique qui centralise les besoins de toutes les applications, on peut alors choisir différents moteurs selon ses besoins, mais on doit alors considérer la duplication des données, l'échange et la transformation de ces données, bref une couche supplémentaire d'intégration, qu'il faut concevoir, développer, maintenir, avec les contraintes d'architecture, de développement, de performances et bien sûr d'intégrité. On se rapproche de la conception orientée services.

Les structures de données des moteurs NoSQL

La plupart des moteurs NoSQL manipulent des agrégats. Il y a bien sûr des exceptions, et les choses évoluent. Autant vous le dire tout de suite : le monde NoSQL est un terrain en pleine évolution, soyez prêt à tout. Un contre-exemple évident est le groupe des moteurs de bases de données orientés graphe, structurés en nœuds et en relations entre les nœuds. Il s'agit d'outils comme Neo4J. Ils ne sont pas du tout architecturés à partir d'une clé, de la racine d'un agrégat, et ils se rapprochent de ce point de vue des moteurs relationnels. Ils sont également difficiles à distribuer. Cet exemple montre qu'il est difficile de généraliser des caractéristiques dans les moteurs réunis sous la bannière NoSQL.

Nous avons déjà vu deux cas de structure : la structure clé-valeur et la structure orientée documents. Il existe une autre grande catégorie, appelée moteurs orientés colonne. Nous en parlerons plus loin, mais pour schématiser, ces moteurs se basent sur la structure de Google BigTable, et reproduisent une structure en apparence assez proche du modèle relationnel. En tout cas, ils reprennent la métaphore de la ligne et des colonnes. Il y a donc une structure plus formelle, plus contraignante que les autres moteurs. Mais dans les faits, ce que vous stockez dans une colonne

de ces bases de données n'est pas typé, ni limité. Vous avez beaucoup moins de contrainte. Et, concrètement, vous êtes beaucoup plus proche d'un agrégat que d'une table, parce que la clé revêt une plus grande importance. Nous n'en disons pas plus ici, nous entrerons dans les détails au chapitre 4.

Le modèle et la réalité

Pour penser et schématiser la réalité, nous avons besoin de modèles : pour penser la matière, nous avons bâti un modèle des atomes qui correspond à de petits systèmes stellaires. Pour penser le découpage le plus élémentaire de la matière, il existe aujourd'hui une théorie des cordes, qui représente ces éléments sous forme de filaments. Cela ne correspond pas à la réalité telle qu'on pourrait l'observer objectivement, c'est simplement un modèle qui va nous permettre, grâce à une simplification et à une adaptation, à quelque chose que nous pouvons saisir cognitivement et représenter. Mais évidemment il ne faut pas confondre réalité et représentation. Et parfois les représentations peuvent changer.

Nous avons déjà signalé qu'un changement de technologie provoque une tension tirant dans deux directions : vers le conservatisme, et vers l'attrait irréfléchi du nouveau. Il y a conservatisme en base de données parce qu'on a pris l'habitude d'un modèle dominant. Le modèle relationnel est juste un modèle, ce n'est pas une vérité objective et naturelle. Il permet beaucoup de choses, mais il n'est pas la vérité révélée et il n'est pas infaillible. Il est facile d'en dénigrer d'autres juste parce qu'on a pris l'habitude d'un modèle. D'autre part, le modèle relationnel possède de nombreuses qualités, et il serait idiot de le remplacer aveuglément par autre chose pour simplement satisfaire un goût de la nouveauté. En fait, le mouvement NoSQL n'est pas une révolution au sens politique du terme : le remplacement d'un ancien régime par un nouveau. On sait en général que ce type de radicalisme s'accompagne d'une forme d'aveuglement. C'est une révolution plutôt copernicienne, dans le sens où à la place d'un modèle hégémonique, qui nous oblige à le choisir en toutes circonstances, on obtient quelque chose de plus riche : le choix.

La modélisation

L'une des caractéristiques les plus importantes du modèle relationnel est justement… les relations, c'est-à-dire le fait de modéliser les données sous forme de tables reliées entre elles par des relations sur des clés. Cela implique une réflexion préliminaire qu'on appelle justement modélisation et dont dépend largement la qualité du système qui sera bâti autour de la base de données. Créer le modèle d'une base de données relationnelle est la première étape du développement, lequel sera d'autant plus facile et efficace que cette étape est réussie. En d'autres termes, pour réussir dans le monde relationnel, il est conseillé de veiller à bien conduire l'étape d'analyse préliminaire. Le modèle relationnel s'intègre bien dans un processus de développement séquentiel tel que le modèle de la chute d'eau *(Waterfall Model)*, illustré sur la figure 2-1.

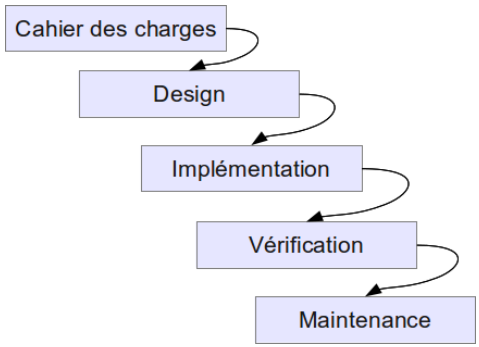


Figure 2-1
Le modèle Waterfall

Ce modèle bien connu implique que chaque étape du développement logiciel soit maîtrisée et terminée avant de passer à l'étape suivante. C'est pour cette raison que le terme *waterfall* est employé : l'eau coule vers le bas, mais il n'y a jamais de retour aux étapes précédentes. Il est assez clair que ce modèle a montré ses nombreuses limites et qu'il est de plus en plus abandonné de nos jours pour des méthodes dites « agiles », qui considèrent le développement comme un processus itératif, où chaque étape essaie d'être suffisamment courte et souple pour pouvoir corriger rapidement les choix malheureux et être beaucoup plus flexible. L'ironie de ce modèle, utilisé pendant des décennies pour conduire les projets informatiques, est que sa première description formelle peut être trouvée dans un article de Winston Royce intitulé « *Managing the Development of Large Software Systems* » (« Gérer le développement de systèmes logiciels de grande taille »), rédigé pour une conférence ayant eu lieu en 1970 (le papier original est disponible à l'adresse suivante : http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf). Le Waterfall est présenté dans cet article comme un exemple de modèle qui ne fonctionne pas. Quarante ans plus tard, un grand nombre de projets informatiques continuent à être basés sur ce type de procédures.

L'un des reproches faits aux SGBDR par le mouvement NoSQL est donc la rigidité du modèle et les difficultés que pose sa réingénierie. Cela est dû également à la structuration forte des données. Dans un SGBDR, vous devez bâtir autant que possible toute la structure avant de pouvoir travailler avec le contenu. Lorsque le contenu est vivant, il devient ensuite difficile de modifier la structure. Pour cette raison, la plupart des moteurs de bases de données NoSQL appliquent un modèle sans schéma *(schema-less)*, où le schéma des données n'a pas vraiment besoin d'être fixé à l'avance. Nous verrons ce que cela signifie dans le chapitre 3 consacré aux choix techniques des moteurs NoSQL.

La rigidité du modèle relationnel

La modélisation est certes une étape importante. Mais le procès en rigidité qui est fait au modèle relationnel est exagéré. De nos jours, ajouter une colonne dans un moteur relationnel n'est pas très difficile, et l'impact sur l'application cliente est nul si le développeur a bien fait son travail. La rigidité du modèle relationnel est donc relative, et les moteurs NoSQL ne sont pas miraculeusement préservés de ce type de problématique. D'une certaine façon, ce n'est pas la faute du modèle, mais de ses implémentations. La modification d'un modèle relationnel est coûteuse, mais changer des données stockées dans une base NoSQL ne l'est pas forcément moins. Si vous avez stocké des

centaines de milliers de documents JSON qui mentionnent un code de produit, et que la structure de ce code change – on passe par exemple d'un EAN 10 à un EAN 13, comme cela s'est produit sur les ISBN *(International Standard Book Number)* pour le marché du livre –, vous serez bien entendu obligé de modifier vos données comme vous auriez à le faire dans un moteur relationnel. Vous pouvez même avoir plus de travail à faire que dans un moteur relationnel. Pour deux raisons :

- beaucoup de moteurs NoSQL sont plutôt des systèmes de dépôts de données qui n'ont pas de concept de langage de manipulation côté serveur et de traitement efficace des opérations sur des collections de données. Le traitement est souvent unitaire ;
- il n'y a en général pas de mécanisme d'abstraction des structures de données semblables aux vues.

Les choses évoluent et continuent à évoluer. Les moteurs NoSQL sont à la base des outils simplifiés, qui peu à peu s'enrichissent fonctionnellement. Mais pour l'instant le modèle relationnel est plus élégant et plus riche fonctionnellement sur bien des points.

Le partitionnement de données

L'un des problèmes de la normalisation dans un SGBDR concerne la distribution des données et du traitement. Si vous stockez des données qui ont un rapport entre elles, comme des clients, des commandes, des factures, des lignes de facture, etc., dans des tables différentes, vous rencontrerez un problème lorsque vous souhaiterez partitionner ces données, les *sharder*[5]. Vous devez alors vous assurer que les données en rapport les unes avec les autres se trouvent sur le même serveur, en tout cas si vous voulez réaliser des jointures du côté du serveur. Dans le cas contraire, vous devrez effectuer des échanges entre serveurs pour manipuler vos données. De ce point de vue, un stockage orienté documents, qui contient dans un document l'intégralité des données correspondant à la requête à effectuer, quitte à générer une base de données finalement très volumineuse à cause d'un grand nombre de doublons de valeurs, présente un certain nombre d'avantages. Dans un modèle où les données sont découpées, il reste possible de s'assurer que les différentes tables se partitionnent de façon à localiser leurs relations, mais cela implique une gestion manuelle des clés primaires et du *sharding*, ainsi que des techniques spécifiques pour distribuer le plus également possible des données entre nœuds. Ce n'est donc pas une affaire simple et elle est difficile à automatiser.

Toutefois, on ne peut argumenter de façon schématique que le modèle relationnel est non-partitionnable et qu'il faut systématiquement partir sur une solution NoSQL basée sur une agrégation. Les choses évoluent et l'avenir verra sans doute une plus grande tolérance au partitionnement du modèle relationnel. Lorsque nous parlerons du théorème CAP (voir page 43), nous verrons qu'il est impossible d'avoir en même temps une disponibilité satisfaisante, une tolérance au partitionnement et une bonne cohérence des données. La cohérence est une chose, les capacités relationnelles en sont une autre. Il y a aujourd'hui de nombreuses initiatives pour distribuer des moteurs relationnels : plusieurs initiatives tournant autour de MySQL ou de MariaDB comme MySQL Cluster, des bases de données dites NewSQL comme VoltDB ou NuoDB, et le plus

5. Le terme *shard* est souvent utilisé dans les moteurs NoSQL pour exprimer un partitionnement géré automatiquement par le système. Nous en parlerons au chapitre 3.

impressionnant et mystérieux de tous, le système relationnel et transactionnel de Google amené à remplacer son moteur orienté colonnes, Google Spanner.

Google F1

Depuis plusieurs années, Google publie régulièrement des articles sur le développement interne d'un moteur de base de données relationnelle complètement distribuée. Notamment, en 2012, des ingénieurs de l'entreprise ont diffusé un document intitulé « F1 - The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business » (http://research.google.com/pubs/pub38125.html) au SIGMOD, le Special Interest Group on Management of Data de l'ACM. Plusieurs systèmes importants de Google étant encore gérés sur MySQL, le géant américain a développé en interne un moteur complètement distribué pour le remplacer. Celui-ci supporte toutes les fonctionnalités importantes d'un SGBDR, avec un moteur de requête parallélisé et la gestion des transactions, tout en s'appuyant sur un stockage complètement distribué. Les mises à jour sont transactionnellement cohérentes, ce qui entraîne une latence plus importante dans les mises à jour, comparé à leur précédent système sur MySQL, mais le document explique quelle a été leur stratégie pour contourner ce problème. Google présente F1 comme le mariage réussi entre les forces du NoSQL (la distribution) et les qualités, notamment transactionnelles, du SGBDR.

En bref, il existe maintenant des implémentations aussi bien que des recherches menées pour permettre le partitionnement d'un modèle basé sur des relations. Aujourd'hui, il demeure beaucoup plus simple de distribuer des données non liées, mais cette situation évoluera peut-être dans un avenir proche.

Le langage SQL

Le langage SQL est un langage d'extraction de données, certes conçu autour du modèle relationnel, mais il n'est pas le modèle relationnel lui-même. SQL est un langage principalement déclaratif, destiné à la manipulation de données et relativement proche du langage naturel. Les bases de données du mouvement NoSQL n'ont pas vocation à s'opposer ou s'éloigner du langage SQL, mais du modèle relationnel. Ce sont les implémentations courantes du modèle relationnel qui comportent les limitations que les bases NoSQL cherchent à dépasser. On devrait donc appeler ce mouvement « NoRelational » plutôt que NoSQL.

Les qualités du langage SQL

Nous ne résisterons pas à citer une conclusion de Philip Greenspun, l'un des pionniers du développement de sites web pilotés par les données, dans un article de blog publié en 2005 et illustrant une requête SQL (http://blogs.law.harvard.edu/philg/2005/03/07/).

« Regardez comme il est amusant de programmer du SQL. Trois lignes de code et vous obtenez une réponse intéressante […]. Comparez cela à du Java et du C, où taper sur votre clavier jusqu'à ce que vos doigts se détachent ne donne en général que peu de résultats. SQL, Lisp et Haskell sont les seuls langages de programmation que j'ai vus où on dépense plus de temps à réfléchir qu'à taper. »

Pourquoi alors vouloir s'éloigner du langage SQL si son expressivité offre un haut niveau d'abstraction et de puissance ? Sans doute pour deux raisons : exprimer une requête correcte en SQL requiert justement plus de réflexion que de programmation. On peut construire une requête SQL

de manière itérative, bloc après bloc, mais ces blocs ne sont pas des opérations, ce sont des éléments de la demande. Bien travailler en SQL demande de ne jamais perdre de vue la question qui est posée, et d'en examiner scrupuleusement chaque mot. C'est réellement une autre façon de penser que celle qui est à l'œuvre dans l'écriture de code impératif. La seconde raison concerne ce qu'on appelle le défaut d'impédance objet-relationnel.

Le requêtage déclaratif en NoSQL

Il y a des évolutions rapides dans le NoSQL, et aussi du point de vue du langage de requête. Depuis la première édition de ce livre, plusieurs changements sont survenus vers l'intégration de langages de requête déclaratifs dans les moteurs NoSQL. Deux exemples : Cassandra, passé en version 2, a remplacé officiellement le langage de l'API Thrift par CQL *(Cassandra Query Language)*, qui reproduit exactement la syntaxe du langage SQL. De son côté, Couchbase Server a introduit dans sa version 3 un langage déclaratif nommé N1QL (prononcer « nickel ») qui utilise une syntaxe `SELECT` très proche du SQL et permet même d'approcher du concept de la jointure avec des instructions comme `NEST` et `UNNEST`.

Autre exemple plus poussé, dans l'analytique interactive avec Hadoop, des outils comme Hive ou Spark proposent une syntaxe SQL qui tente de se rapprocher de la norme SQL. HiveQL inclut ainsi une large part de la syntaxe des jointures définie dans la norme, avec jointures internes, externes, les `CROSS JOIN` et même une syntaxe `SEMI JOIN` pour implémenter dans la clause de jointure la sémantique du mot-clé `EXISTS`. L'une des restrictions de HiveQL est de se limiter à une équijointure : on ne peut écrire des clauses de jointure qui reposent sur autre chose qu'une égalité. Mais c'est de toute façon la grande majorité des cas d'utilisation.

Le terme NoSQL devient donc de moins en moins valide, et dans peu de temps, le seul critère commun subsistant pour essayer de donner une définition générale des moteurs NoSQL, à savoir qu'ils n'utilisaient pas de paradigme relationnel, risque de devenir obsolète.

Méfiez-vous des comparaisons

Une chose assez évidente, mais qui est hélas rarement mise en avant dans les comparaisons faites par les défenseurs du NoSQL avec les moteurs relationnels, c'est que MySQL est souvent utilisé comme point de comparaison. La plupart des entreprises qui ont développé des moteurs NoSQL ont un esprit open source : Google, Facebook, Twitter, LinkedIn utilisent tous intensivement MySQL ou MariaDB. Twitter par exemple a développé en 2011 une couche de sharding sur MySQL nommée Gizzard.

Le problème, c'est que MySQL n'est pas le meilleur moteur relationnel. Il souffre d'un certain nombre de problématiques qui rendent son utilisation intensive et ses performances délicates, et on ne peut pas le prendre comme modèle d'une critique objective de tous les moteurs relationnels. En voici quelques preuves.

Beaucoup de défenseurs du mouvement NoSQL avancent que l'une des forces de ces moteurs est l'absence de jointures, et que ce sont les jointures qui provoquent de forts ralentissements dans les moteurs relationnels. À la base, la jointure n'est pas un problème dans un bon SGBDR. Elle se fait en général entre la clé primaire et une clé étrangère, et si toutes les deux sont indexées, cela équivaut à une recherche de correspondance à travers deux index. Si la cardinalité du résultat

est faible, c'est-à-dire si la correspondance touche un petit nombre de lignes, les algorithmes de jointure des moteurs relationnels offrent d'excellentes performances. En fait, les performances sont meilleures que celles d'un moteur NoSQL, parce que ce qui coûte le plus cher dans un moteur de base de données ce sont les entrées-sorties. À partir du moment où le modèle relationnel élimine la redondance, le volume de données à parcourir pour résoudre la requête est moindre que dans un moteur NoSQL, et donc les performances sont meilleures.

Mais, historiquement, MySQL gère très mal les jointures. Jusqu'à la version 5.6 de MySQL et les versions 5.3 et 5.5 de MariaDB, le seul algorithme disponible est la boucle imbriquée. C'est un algorithme valable sur des petits volumes et qui devient totalement contre-performant sur des volumétries plus importantes. Les moteurs relationnels commerciaux implémentent deux autres algorithmes qui sont la jointure de fusion (merge join ou sort-merge join) et la jointure de hachage (hash join). Notamment, la jointure de hachage est beaucoup plus performante sur des gros volumes.

Donc MySQL n'est vraiment pas un bon point de comparaison pour les jointures volumineuses, car d'autres moteurs relationnels s'en sortent beaucoup mieux. Mais il est clair que lorsqu'on veut effectuer des jointures sur des volumes importants pour des requêtes analytiques où le nombre de correspondances est important, un moteur relationnel va montrer ses limites et le NoSQL arrive à la rescousse.

Autre exemple, on a abordé le sujet de la rigidité du modèle. Si vous utilisez MySQL, vous pouvez facilement avoir l'impression qu'il est très pénible de modifier le schéma de vos tables. Jusqu'à la version 5.6 de MySQL, un certain nombre de modifications de tables étaient effectuées « hors ligne ». Beaucoup de commandes `ALTER TABLE`, au lieu d'effectuer simplement la modification dans la structure de la table, déclenchaient derrière le rideau la création d'une nouvelle table, suivie de la copie de toutes les données de l'ancienne table dans la nouvelle, puis de la suppression de l'ancienne table et enfin du renommage de la nouvelle. Vous imaginez l'effet sur des tables volumineuses dans un environnement fortement multi-utilisateurs. Or, bien entendu, les SGBDR ne fonctionnent pas tous ainsi.

On voit également souvent des données chiffrées de volumétrie. Par exemple, on lit qu'une table comportant un million de lignes deviendrait impossible à gérer dans un moteur relationnel. En réalité, un moteur SQL est capable de gérer correctement des tables qui comportent des centaines de millions de lignes, voire des milliards de lignes, à partir du moment où les recherches qu'on effectue sur ces tables sont sélectives et utilise de bons index. Dans le terme Big Data, Big veut dire vraiment Big.

Le défaut d'impédance

L'une des rugosités du langage SQL est ce que les Anglo-Saxons appellent le défaut d'impédance *(impedance mismatch)* objet-relationnel.

Ce terme est un emprunt à l'électricité. L'impédance est, pour simplifier, une opposition faite par un circuit au passage d'un courant électrique. Optimiser l'impédance consiste à diminuer cette résistance et donc à transférer une énergie de façon optimale. Par défaut d'impédance, les créateurs du terme voulaient signifier que le passage du relationnel à l'objet s'effectue avec une

perte d'énergie et une résistance trop forte. Prenons l'exemple d'un code PHP qui appelle une requête SQL pour faire une recherche dans MySQL :

```
<?php

mysql_connect('localhost','root','********');
mysql_select_db('********');

$nom    = $_REQUEST['nom'];
$prenom = $_REQUEST['prenom'];
$ville  = $_REQUEST['ville'];
$pays   = $_REQUEST['pays'];
$sexe   = $_REQUEST['sexe'];
$age    = $_REQUEST['age'];

$where = "";
$where .= isset($nom)   ? (isset($where)? " OR ": " WHERE " )."( nom    = '$nom') " :
")"
$where .= isset($prenom)? (isset($where)? " OR ": " WHERE " )."( prenom =
'$prenom') ": ")"
$where .= isset($ville) ? (isset($where)? " OR ": " WHERE " )."( ville  = '$ville')
": ")"
$where .= isset($pays)  ? (isset($where)? " OR ": " WHERE " )."( pays   = '$pays')
 ": ")"
$where .= isset($sexe)  ? (isset($where)? " OR ": " WHERE " )."( sexe   = '$sexe')
": ")"
$where .= isset($age)   ? (isset($where)? " OR ": " WHERE " )."( age    = '$age') " :
")"

$sql = "SELECT nom, prenom, ville, pays, sexe, age
FROM Contact
$where
ORDER BY nom, prenom  DESC ";

$req = mysql_query($sql) or die('Erreur SQL!<br />'.$sql.'<br />'.mysql_error());

$data = mysql_fetch_array($req);

mysql_free_result ($req);?>
```

Ici, nous devons construire une chaîne pour bâtir la requête SQL qui sera envoyée telle quelle au serveur. Afin de bâtir la meilleure chaîne possible, nous testons la valeur de chaque paramètre pour éviter d'introduire dans la requête des critères de recherche trop nombreux, ce qui diminuerait les performances de la requête. Le code PHP est reconnu par l'interpréteur, c'est du vrai code. La chaîne SQL, quant à elle, n'est rien d'autre à ce niveau qu'une chaîne. Elle ne sera reconnue que du côté du serveur. Il n'y a pas de réelle interaction entre le code PHP et le code SQL.

Il y a effectivement quelque chose de frustrant que l'on se place d'un côté ou de l'autre de la lorgnette. Si on est un développeur objet, on souffre de devoir sortir du modèle objet pour interroger la base de données. Du point de vue du défenseur du langage SQL, l'élégance de sa syntaxe lui

permet d'exprimer la complexité d'une façon très sobre, et résume des opérations complexes en peu de mots. Les langages objet restent tributaires d'une logique procédurale, où chaque opération doit être exprimée. L'exécution d'un code procédural est aveugle. Pas de raisonnement nécessaire, le code s'exécute comme demandé. Un langage déclaratif comme le SQL est soutenu par un moteur d'optimisation qui décide, en vrai spécialiste, de la meilleure méthodologie pour obtenir les résultats désirés. Il s'agit de deux approches très différentes qui naturellement s'expriment par des langages qui ne fonctionnent pas au même niveau. Cela conduit à rassembler tant bien que mal deux types d'opérations très différentes dans le même code.

Pour traiter ce défaut d'impédance, on peut maquiller l'appel au SQL en l'encapsulant dans des objets qui vont faire office de générateur de code SQL. Les frameworks de mapping objet-relationnel (*Object-Relational Mapping*, ORM) dissimulent ce travail en automatisant la création de classes pour les tables de la base de données, les lignes étant représentées par des instances de ces classes. La théorie des générateurs de code est intéressante, mais en pratique le langage SQL est riche et complexe, et il est impossible d'automatiser élégamment l'écriture d'une requête qui dépasse les besoins élémentaires d'extraction et de modification. Les requêtes générées par des outils comme Hibernate ou Entity Framework sont par la force des choses génériques et grossières, et ne permettent pas de développer toute la puissance du SGBDR. En fait, les requêtes générées sont très souvent bancales et disgracieuses et posent des problèmes de performance. De plus, les ORM déplacent une partie du traitement des données vers le client et se privent ainsi des grandes capacités des moteurs relationnels à manipuler et extraire des ensembles.

Pas de défaut d'impédance en NoSQL ?

Est-ce que les moteurs NoSQL corrigent cette problématique, si tant est qu'elle en soit une ? Oui, d'une certaine façon. Même si certains moteurs NoSQL offrent un langage de requête déclaratif, il est encapsulé dans un pilote adapté à chaque langage client. Voici un exemple de requête vers MongoDB à partir d'un code PHP. Nous essayons de reproduire quelque chose de semblable à notre exemple MySQL.

```
<?
$mongo = new Mongo('mongodb://localhost:27017', array("timeout" =>
2000));
$db = $mongo->selectDB('CRM');

$db->Contact->find(
array('$or' => array(
    array('nom'    => $nom),
    array('prenom' => $prenom),
    array('ville'  => $ville),
    array('pays'   => $pays),
    array('sexe'   => $sexe),
    array('age'    => $age)
)),
array ('nom', 'prenom', 'ville', 'pays', 'sexe', 'age')
);
?>
```

Ici, bien sûr, la requête utilise des objets propres au pilote MongoDB pour PHP développé par MongoDB Inc. Les critères sont envoyés sous forme de double tableau, c'est ensuite à MongoDB de se débrouiller avec ça, par exemple en utilisant un index secondaire s'il est présent. Au passage, la stratégie d'exécution est visible dans MongoDB en utilisant la méthode `explain`, comme ceci :

```
$db->Contact->find(
array('$or' => array(

)),
array ('nom', 'prenom', 'ville', 'pays', 'sexe', 'age')
)->explain();
```

D'autres moteurs, comme CouchDB, offrent une interface REST (*Representational State Transfer*, voir le chapitre 3 consacré aux choix techniques du NoSQL). Envoyer une requête REST et envoyer une requête SQL sont deux actions assez similaires : il faut générer une URI avec des paramètres. La différence est qu'une URI est plus facile à générer qu'une requête SQL, et de nombreuses bibliothèques existent pour tous les langages clients qui permettent d'exprimer les paramètres sous forme de table de hachage ou de tableau, par exemple. Voici un exemple de génération d'une requête REST en Python :

```
import urllib
import urllib2

url = 'http://localhost:5984'
params = urllib.urlencode({
   'nom': nom,
   'prenom': prenom,
   'ville': ville,
   'pays': pays,
   'sexe': sexe,
   'age': age)
})
response = urllib2.urlopen(url, params).read()
```

D'autres moteurs supportent une interface Thrift, qui est une API d'accès que nous détaillerons dans le chapitre 3.

Les ORM pour NoSQL

Les instructions d'extraction des moteurs NoSQL sont simplifiées : on est loin de la complexité du langage SQL. De plus, les performances ne dépendent pas la plupart du temps de la syntaxe des requêtes, mais de la distribution du traitement. Il est donc beaucoup plus facile de concevoir des générateurs de code. Même si ce n'est pas un besoin qui s'est fait sentir avec force, c'est une demande de beaucoup d'équipes de développement habituées à cette manière de travailler, et il existe des bibliothèques pour les moteurs NoSQL les plus importants, par exemple avec MongoDB, mongoose pour Node.JS ou mongoengine pour Python.

Les objets et le relationnel

Une autre réflexion fréquemment rencontrée dans le monde NoSQL comparé au modèle relationnel est le rapport à la « réalité » des données, en tout cas au modèle tel qu'on le bâtit dans l'application cliente, en général sous forme d'objets. L'application cliente organise ses données sous forme d'objets et se trouve obligée de décomposer ces objets pour les stocker dans des tables différentes. Mais les deux modèles sont-ils si différents ? Une bonne organisation des classes permet presque de respecter les trois premières formes normales : chaque objet est une entité distincte dont les propriétés représentent des attributs qui se rapportent strictement à la classe, et les classes entre elles entretiennent des relations avec différentes cardinalités, à travers des collections d'objets. Sérialiser en un document une classe et ses collections d'objets liés dans un document JSON n'est donc pas plus logique que de le faire dans des tables distinctes. Au contraire, l'un des avantages du modèle relationnel est de permettre une grande économie de stockage et une représentation très logique des données, de façon à offrir tous les traitements et toutes les utilisations possibles de ces données. De plus, il n'y a pas de justification logique à conserver les données telles qu'elles sont manipulées par un langage client.

C'est comme si quelqu'un vous disait que les dictionnaires devraient être organisés en phrases complètes, par ordre du sujet, puisque c'est par le sujet que la phrase commence, et ceci parce que tout le monde parle ainsi. Le dictionnaire est organisé par ordre alphabétique de mots, parce que le but d'un dictionnaire est d'y puiser l'un après l'autre les vocables qui composeront les phrases, lesquelles sont de la responsabilité du locuteur, et pas du dictionnaire.

Les NULL

Une autre critique portée aux SGBDR concerne les `NULL`. Du fait de la prédéfinition d'un schéma rigide sous forme de table, ou chaque colonne est prédéfinie, un moteur relationnel doit indiquer d'une façon ou d'une autre l'absence de valeur d'une cellule. Ceci est réalisé grâce au marqueur `NULL`, qui est un animal un peu spécial dans le monde du SGBDR. Au lieu de laisser la cellule vide, on y pose un marqueur, `NULL`, qui signifie valeur inconnue. Le raisonnement est le suivant : laisser une chaîne vide dans la cellule indique une valeur, à savoir « vide ». Le marqueur `NULL` indique clairement l'absence de valeur, ce n'est donc pas une valeur. Ceci dit, selon les implémentations, ce marqueur `NULL` coûte un petit peu en stockage. Une autre problématique du `NULL` est sa gestion compliquée dans le langage SQL. Dans la mesure où il ne s'agit pas d'une valeur, le marqueur `NULL` ne peut pas être comparé, par exemple dans un critère de filtrage (clause `WHERE`) qui est évalué à vrai ou faux. Il impose la gestion d'une logique à trois états : vrai, faux et inconnu. Ainsi, la requête suivante ne retournera pas les lignes dont le titre est `NULL` :

```
SELECT *
FROM Contact
WHERE Titre = 'Mme' OR Titre <> 'Mme';
```

car le `NULL` ne pouvant se comparer, il n'est ni égal à « Mme », ni différent, il est simplement inconnu. Il faut donc chaque fois penser à gérer une logique à trois états de la façon suivante :

```
SELECT *
FROM Contact
WHERE Titre = 'Mme' OR Titre <> 'Mme' OR Titre IS NULL;
```

Les moteurs NoSQL amènent de la souplesse dans cette conception. Dans les moteurs orientés documents ou paires clé-valeur, il n'y a aucune notion de schéma préétabli et de validation d'une structure. Il n'est donc jamais nécessaire d'indiquer un attribut pour dire qu'il est inconnu ! Il suffit de ne pas déclarer l'attribut dans le document. Par exemple, lorsque vous créez un index secondaire sur des éléments du document, cela suppose pour MongoDB que les valeurs des éléments valent `NULL` si elles ne sont pas trouvées dans le document. Il n'y a donc plus besoin de gérer ces valeurs lorsqu'elles n'ont pas de sens dans le document. Cette forme de représentation des données est ainsi beaucoup plus dense que ne l'est celle des SGBDR. Dans les moteurs orientés colonne, les métadonnées des colonnes ne sont pas prédéfinies à la création de la table. En fait, la colonne doit être exprimée à chaque création d'une ligne. Elle peut s'y trouver ou non. On est donc dans le même cas de figure : il n'y a pas réellement de schéma explicite, et les colonnes peuvent être différentes d'une ligne à l'autre. Au lieu de devoir poser un `NULL` en cas d'absence de valeur, il suffit de ne pas exprimer la colonne. Pour cette raison, les SGBD orientés colonnes n'intègrent pas le concept de `NULL`, et on les appelle des entrepôts de données « maigres » (le terme anglais *sparse* est assez difficile à traduire dans ce contexte. On trouve parfois la traduction littérale « éparse », ce qui ne veut pas dire grand-chose).

Problème résolu ?

Donc les moteurs NoSQL résolvent le problème des `NULL` ? Oui, mais il reste deux choses à signaler. D'abord, les `NULL` sont-ils un réel problème ? Ils gênent les requêtes par leur logique à trois états. Dans ce cas, problème résolu. En ce qui concerne l'occupation d'espace des `NULL`, il est minime dans les SGBDR, et on ne peut pas vraiment mettre l'économie d'espace de stockage au crédit des moteurs NoSQL. L'absence de relation entre les tables entraîne en général beaucoup de redondance de données et donc, `NULL` ou pas `NULL`, les moteurs NoSQL occupent beaucoup plus d'espace de stockage qu'une SGBDR au modèle correctement normalisé. Il s'agit donc plus d'un gain en clarté pour la programmation qu'en stockage.

Ironiquement, il nous est arrivé de voir des `NULL` dans les données des moteurs NoSQL, notamment en MongoDB. Non pas que MongoDB le demande, mais simplement parce qu'il s'agissait d'un import des données d'un SGBDR, et que la routine développée pour cet import ne se préoccupait pas de tester si le contenu des colonnes était ou non `NULL` pour générer l'élément dans le document JSON. Le script d'import générait des documents qui comportaient chaque colonne et écrivait le mot « `NULL` » comme valeur de l'élément, le cas échéant.

Le transactionnel et la cohérence des données

Dans le monde informatique, une transaction est une unité d'action qui doit respecter quatre critères, résumés par l'acronyme ACID, et qu'on nomme donc l'acidité de la transaction. Ces critères sont présentés dans le tableau 2-1.

Tableau 2-1. Critères ACID de la transaction

Critère	Définition
Atomique	Une transaction représente une unité de travail qui est validée intégralement ou totalement annulée. C'est tout ou rien.
Cohérente	La transaction doit maintenir le système en cohérence par rapport à ses règles fonctionnelles. Durant l'exécution de la transaction, le système peut être temporairement incohérent, mais lorsque la transaction se termine, il doit être cohérent, soit dans un nouvel état si la transaction est validée, soit dans l'état cohérent antérieur si la transaction est annulée.
Isolée	Comme la transaction met temporairement les données qu'elle manipule dans un état incohérent, elle isole ces données des autres transactions de façon à ce qu'elle ne puisse pas lire des données en cours de modification.
Durable	Lorsque la transaction est validée, le nouvel état est durablement inscrit dans le système.

Cohérence peut aussi se dire consistance. La notion de cohérence est importante, et c'est l'un des éléments les plus sensibles entre le monde du relationnel et le monde NoSQL. Les SGBDR imposent une règle stricte : d'un point de vue transactionnel, les lectures de données se feront toujours (dans les niveaux d'isolation par défaut des moteurs) sur des données cohérentes. La base de données est visible en permanence sur un état cohérent. Les modifications en cours sont donc cachées, un peu comme sur la scène d'un théâtre : chaque acte d'une pièce de théâtre se déroule intégralement sous l'œil des spectateurs. Si le décor doit changer, le rideau se baisse, les spectateurs doivent attendre, les changements s'effectuent derrière le rideau et lorsque le rideau se lève, la scène est de nouveau dans un état totalement cohérent. Les spectateurs n'ont jamais eu accès à l'état intermédiaire.

Mais cet état intermédiaire peut durer longtemps, pour deux raisons : plusieurs instructions de modifications de données peuvent être regroupées dans une unité transactionnelle, qui peut donc être complexe. Ensuite, un SGBDR fonctionnant de façon ensembliste, une seule instruction de mise à jour, qui est naturellement et automatiquement transactionnelle, peut très bien déclencher la mise à jour de milliers de lignes de table. Cette modification en masse conservera des verrous d'écriture sur les ressources et écrira dans le journal de transaction ligne par ligne. Tout ceci prend bien évidemment du temps.

Relâchement de la rigueur transactionnelle

Une transaction, c'est donc un mécanisme d'isolation d'une opération simple ou complexe. Elle est importante dans le cadre d'un SGBDR, car du fait du modèle relationnel, une donnée complexe va se décomposer dans plusieurs tables et devra donc faire l'objet de plusieurs opérations d'écriture. Si je considère une donnée comme une commande client, qui comporte les informations du client, des produits commandés et de la facture, il faudra exécuter plusieurs commandes SQL d'insertion, sur les tables client, commande et facture. Le langage SQL ne permet de cibler qu'une table à la fois dans la commande INSERT. L'opération logique de l'insertion d'une donnée complexe devra se traduire par plusieurs opérations physiques d'insertion, et devra être regroupée dans une transaction explicite pour s'assurer qu'il n'est pas possible de se retrouver dans la base de données avec des données partiellement écrites.

On comprend bien qu'un moteur NoSQL, basé sur le principe de l'agrégat, comme nous en avons discuté, ne présente pas ce genre de problème. Dans MongoDB par exemple, on écrit en une seule fois un document JSON qui agrège toutes les informations de la commande. : plus besoin par conséquent d'une transaction explicite. De fait, les critères de la transaction changent de sens : l'écriture est naturellement atomique car le document est écrit en une fois, et elle est naturellement isolée car le document est verrouillé durant l'écriture. Le besoin d'une gestion explicite d'une transaction complexe disparaît, car l'atomicité n'est plus le fait du langage, mais d'une structuration de la donnée décidée par l'architecte ou le développeur.

Il faut donc repenser le concept des critères transactionnels à la lumière d'un système qui gère différemment ses données, et dont les possibilités et les exigences sont autres.

Les systèmes NoSQL agrègent les données, mais quand ils sont distribués, ils les dupliquent. Cela repose le problème transactionnel dans le cadre de la réplication. Si on écrit une donnée plusieurs fois, sur plusieurs nœuds d'un cluster, cette écriture redondante sera-t-elle atomique, cohérente, isolée et durable ? C'est dans ce contexte qu'il faut comprendre les discussions autour de la cohérence *(consistency)* des écritures dans un moteur NoSQL. Nous allons étudier cet aspect plus en détail.

Distribution synchrone ou asynchrone

Nous entrons donc ici brièvement dans des problématiques d'informatique distribuée. La distribution de données peut se concevoir de deux façons : synchrone ou asynchrone. Une distribution synchrone signifie que le processus de réplication des données garantit que les réplicas soient tous alimentés avant de considérer que l'écriture est terminée. Une distribution asynchrone va autoriser la réplication à se faire hors du contrôle du processus de mise à jour. Il n'y aura donc pas de garantie stricte que la réplication s'effectue avec succès.

Le mode de réplication synchrone va permettre de garantir la cohérence des données dans un système distribué, tel que nous venons de le voir. Dans ce cadre, la cohérence est assurée par consensus. Le consensus en informatique, c'est simplement l'accord de plusieurs processus, locaux ou distants, sur une valeur commune. C'est un concept essentiel en calcul et en stockage distribué. Dans l'idéal, tous les processus faisant partie d'un traitement devraient pouvoir participer au consensus, mais il est nécessaire de prendre en compte les risques de défaillance, et donc, un bon protocole permettant d'atteindre un consensus doit être tolérant aux défaillances d'un ou de plusieurs nœuds, pour arriver à former un quorum (une décision à la majorité) plutôt qu'une décision à l'unanimité.

Assurer une réplication à base de consensus en tolérant les pannes n'est possible que sur un réseau synchrone. Cette affirmation a été prouvée dans un article de Fischer, Lynch et Paterson titré « *Impossibility of distributed consensus with one faulty process* »[6]. Cela signifie une chose : si vous voulez assurer une cohérence distribuée, vous devez travailler de façon synchrone. Si vous souhaitez favoriser les performances et travailler de façon asynchrone, vous ne pourrez pas garantir la cohérence de votre réplication. Dans les systèmes asynchrones, on parle donc souvent de cohérence finale *(eventual consistency)* pour exprimer le fait que la cohérence n'est pas garantie

6. Journal of the ACM, vol. 32, n° 2, avril 1985

immédiatement mais que des mécanismes vont permettre d'assurer la mise à jour des réplicas de façon asynchrone avec un délai.

La transaction distribuée

Les moteurs relationnels n'ont pas le choix : ils doivent garantir l'acidité de la transaction, donc la cohérence des écritures. Si vous voulez distribuer une base de données relationnelle, vous devez donc être dans un réseau synchrone et utiliser un protocole spécifique qui assure de façon distribuée la cohérence de cette transaction. Il s'agit de ce qu'on appelle une transaction distribuée.

Comme son nom l'indique, une transaction distribuée s'applique sur plusieurs machines. Son implémentation est relativement complexe. Elle s'appuie sur la présence sur chaque machine d'un gestionnaire de ressources (*ressource manager*, RM) qui supporte en général la spécification X/Open XA rédigée par l'*Open Group for Distributed Transaction Processing*. Les machines sont supervisées par un gestionnaire centralisé appelé le *Transaction Processing Monitor* (TPM). Au moment de la validation *(commit)* de la transaction, le TPM organise une validation en deux phases (*two-phase commit*, 2PC), représentée sur la figure suivante.

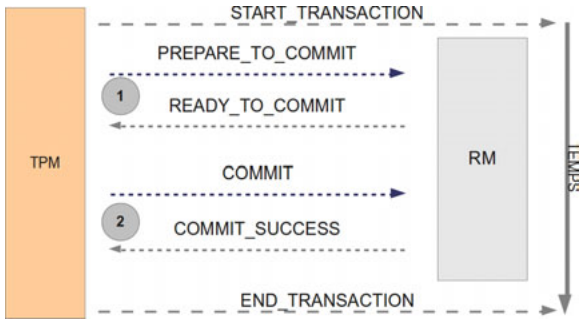


Figure 2-2
Validation à deux phases dans une transaction distribuée

Lorsque la transaction est prête à être validée, une première phase prépare la validation sur tous les sous-systèmes impliqués, jusqu'à ce que le TPM reçoive un `READY_TO_COMMIT` de la part de tous les RM. Si c'est le cas, la seconde phase de validation réelle peut commencer. Le TPM envoie des ordres de `COMMIT` aux RM et reçoit de leur part des messages de `COMMIT_SUCCESS`. La procédure est donc assez lourde et peut être source de ralentissement dans les systèmes transactionnels qui l'utilisent intensément.

Autre remarque : la validation à deux phases permet de vérifier qu'une transaction peut s'exécuter de façon atomique sur plusieurs serveurs. Il s'agit en fait d'une version spécialisée des protocoles de consensus des systèmes distribués. La première phase de la validation sert à s'assurer que les participants sont prêts à valider, c'est une phase de vote. Si tout le monde est d'accord, la seconde phase réalise la validation. On peut se demander ce qui se passerait si l'un des sous-systèmes venait à crasher après la phase de vote, la seconde phase de validation n'ayant pas encore été réalisée. Dans certains rares cas, cela peut provoquer une erreur de la transaction. Mais dans la plupart des cas, si la machine en erreur redémarre et restaure l'état du moteur transactionnel, ce dernier finira par valider la transaction. Vous voyez peut-être où se situe également le problème : 2PC est un protocole bloquant. La transaction reste ouverte jusqu'à ce que tous les participants aient

envoyé le `COMMIT_SUCCESS`. Si cela dure longtemps, des verrous vont rester posés sur les ressources locales et diminuer la concurrence d'accès.

Paxos

La transaction distribuée est principalement utilisée dans le monde des SGBDR pour la réplication. Elle souffre notamment d'un défaut rédhibitoire qui la rend inutilisable dans le monde NoSQL : l'absence de tolérance aux pannes. Dans la transaction distribuée, si un acteur ne répond pas, la transaction est annulée. Il s'agit en fait d'une forme de consensus à l'unanimité où tous les nœuds doivent être présents. L'objectif d'un moteur NoSQL est d'assurer une montée en charge sur un nombre théoriquement illimité de nœuds, sur du commodity hardware. À tout moment, un nœud peut donc tomber en panne. Il faut par conséquent appliquer un protocole de consensus à la majorité.

Le père des protocoles de consensus s'appelle Paxos. Paxos est un protocole proposé pour la première fois en 1989, notamment par Leslie Lamport, le bien connu créateur de LaTeX. Il existe donc depuis longtemps et a profité d'un certain nombre d'améliorations, notamment en termes de performances, pour répondre au mieux aux besoins dans un environnement distribué moderne. Avec Paxos, vous pouvez maintenir un état partagé et synchrone entre plusieurs nœuds. Pour qu'un tel protocole fonctionne dans le monde réel, il est important qu'il soit tolérant aux pannes. Il est maintenant admis qu'on ne peut bâtir un système informatique distribué sans une forme de résistance aux défaillances matérielles ou de réseaux. Nous verrons dans ce livre que les moteurs NoSQL distribués ont tous des mécanismes de tolérance de panne. C'est un élément nécessaire. Paxos inclut plusieurs mécanismes de tolérance de panne, notamment parce qu'il se base sur une forme de consensus à la majorité et qu'il peut donc continuer son travail même si un ou plusieurs nœuds sont tombés. Si une majorité des nœuds est présente et accepte les modifications, Paxos pourra répliquer les données.

Il n'y a pas d'implémentation officielle de Paxos, il s'agit de la description d'un protocole et les implémentations diverses peuvent varier dans les détails. Une implémentation très utilisée dans les outils qui tournent autour d'Hadoop s'appelle Zookeeper, un gestionnaire de consensus et d'état distribué.

Mais d'un point de vue général, la garantie de la cohérence sur des systèmes distribués est contraignante et entraîne des diminutions de performances en écriture, proportionnellement au nombre de nœuds, alors que l'objectif de la distribution, en tout cas dans le contexte de notre réflexion sur NoSQL, est d'augmenter les performances par une montée en charge horizontale. C'est pour cette raison que l'option de la cohérence finale est choisie dans plusieurs moteurs NoSQL. Le fait que certains moteurs NoSQL ne permettent pas d'assurer la cohérence des données est l'un des sujets majeurs de discorde entre le monde relationnel et le NoSQL. C'est aussi l'un des sujets chauds en raison d'un axiome bien connu dans le monde NoSQL, appelé le théorème CAP.

Le théorème CAP

En juillet 2000, lors du symposium sur les principes de l'informatique distribuée (*Principles of Distributed Computing*, PODC) organisé par l'ACM, le professeur Eric Brewer de l'université de Californie à Berkeley fit une présentation inaugurale intitulée *Towards Robust Distributed*

System. Eric Brewer est non seulement professeur à l'université, mais aussi cofondateur et *Chief Scientist* de la maintenant défunte société Inktomi, que les moins jeunes d'entre nous connaissent comme un des moteurs de recherche web importants de l'ère pré-Google.

Les transparents de cette présentation sont disponibles sur le site d'Eric Brewer : http://www.cs.berkeley.edu/~brewer/. Il s'agit d'une réflexion générale sur la conception de systèmes distribués, ce qui intéresse spécialement dans le monde du NoSQL. Nous pensons qu'il est utile de résumer très brièvement les réflexions d'Eric Brewer, ce qui nous fera nous intéresser au fameux théorème CAP. Classiquement, la focalisation de l'informatique distribuée est ou était faite sur le calcul et non sur les données. C'est pour des besoins de calcul qu'on créait des clusters ou des *grids* de travail, pour atteindre une puissance de calcul multipliée par le nombre de nœuds.

Cluster et grid

On fait en général la différence entre un cluster et un grid de machines de la façon suivante : un cluster est un groupe de machines situé dans le même espace géographique, c'est-à-dire dans le même data center. Un grid est un ensemble de machines distribuées dans plusieurs data centers.

L'exemple tout public de ceci est le projet SETI@home. Ce dernier utilise les machines des utilisateurs volontaires à travers le monde pour analyser les données du projet afin de détecter l'éventuelle présence d'une intelligence extraterrestre (http://setiathome.ssl.berkeley.edu/).

Mais, comme l'affirme Eric Brewer, le calcul distribué est relativement facile. Le paradigme Hadoop le démontre : un algorithme MapReduce, qui distribue le travail et agrège les résultats, n'est pas particulièrement difficile à concevoir et à mettre en œuvre. Ce qui est difficile, c'est la distribution des données. À ce sujet, Brewer énonce le théorème CAP, une abréviation de trois propriétés :

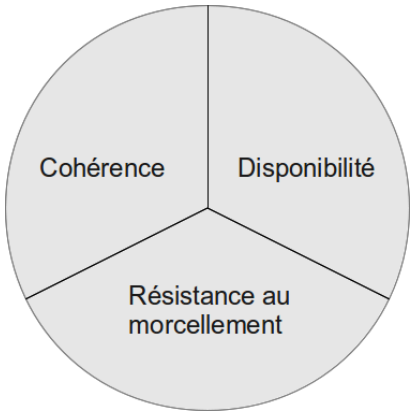
- *consistency* (cohérence) : tous les nœuds sont à jour sur les données au même moment ;
- *availability* (disponibilité) : la perte d'un nœud n'empêche pas le système de fonctionner et de servir l'intégralité des données ;
- *partition tolerance* (résistance au morcellement) : chaque nœud doit pouvoir fonctionner de manière autonome.

Le cas particulier de la résistance au morcellement

L'expression *partition tolerance* n'est pas toujours très claire. On entend par là qu'un système partitionné doit pouvoir survivre aux difficultés du réseau. La définition donnée par Seth Gilbert et Nancy Lynch est la suivante : « Le réseau peut perdre arbitrairement beaucoup de messages envoyés de l'un à l'autre. » Il s'agit ici d'une propriété du réseau. Cela ne signifie donc pas qu'un système soit partitionné, mais cela décrit sa capacité à survivre aux aléas de la perte de partitions ou de communication entre les partitions. Il ne s'agit jamais d'une garantie totale d'accès à partir d'un seul nœud – une situation idéale impossible à atteindre – mais bien d'une capacité à la survie et à la résilience aussi poussée que possible.

Les bases de données partagées ne peuvent satisfaire que deux de ces critères au plus.

Figure 2-3
Le théorème CAP



En 2002, Seth Gilbert et Nancy Lynch du MIT *(Massachusetts Institute of Technology)* ont publié un papier visant à apporter une démonstration de ce principe. Gilbert et Lynch analysent ce trio de contraintes sous l'angle des applications web et concluent qu'il n'est pas possible de réaliser un système qui soit à la fois ACID et distribué.

Si on regarde les bases de données relationnelles traditionnelles selon ce théorème, elles réussissent en matière de cohérence et de support du partitionnement et bien sûr de consistance à travers la garantie ACID des transactions, mais pas en matière de disponibilité.

Cohérence *vs* disponibilité

Si l'objectif d'un système distribué comme une base NoSQL est d'assurer une disponibilité maximale, cela signifie simplement que toute requête envoyée au système NoSQL doit retourner une réponse (le plus rapidement possible, mais cela n'est pas en soi une exigence). Il faut donc pour cela que la perte d'un ou de plusieurs nœuds n'ait pas d'incidence sur les réponses. Dans un système qui privilégie la consistance, on doit s'assurer que les données sont bien lues à partir du ou des nœuds qui contiennent les données les plus fraîches, ou que tous les nœuds sont à jour avant de pouvoir lire la donnée, ce qui pose des problèmes de disponibilité de l'information.

Nous aborderons les solutions techniques apportées par les moteurs NoSQL dans le chapitre suivant. Mais nous pouvons indiquer ici que les solutions des moteurs NoSQL sont assez variées et plus complexes que ce que vous avez peut-être déjà entendu dire. Il ne s'agit pas, pour les moteurs NoSQL asynchrones, d'un abandon pur et simple de la cohérence pour favoriser la disponibilité, les risques sont trop importants. Tout le monde n'est pas prêt à tout miser sur un SGBD distribué qui ne présente aucune garantie de maintien de cohérence de ses données, une sorte de niveau d'isolation chaos.

PACELC

Le théorème CAP est encore aujourd'hui sujet de débat. Certains en soulignent les imperfections, d'autres l'affinent en proposant des modèles plus complets, comme le modèle PACELC de Daniel Abadi du département de Computer Science de l'université Yale.

Le modèle PACELC est une sophistication du théorème CAP qui a l'avantage de proposer un cadre d'implémentation plus souple, avec plus de choix. Il nous a semblé intéressant d'en dire un mot pour montrer – nous le verrons en pratique dans la partie II présentant les moteurs NoSQL – que la cohérence n'est pas forcément un critère qu'on respecte scrupuleusement ou qu'on abandonne. Ce n'est pas blanc ou noir. Des moteurs comme Cassandra permettent de décider à chaque instruction du niveau de cohérence et de durabilité. Dans un moteur relationnel, le seul critère sur lequel il est possible d'avoir une action est l'isolation, comme nous l'avons vu. Un moteur comme Cassandra offre plus de souplesse dans la gestion transactionnelle. Nous en reparlerons dans le chapitre 11, dédié à Cassandra.

PACELC est donc un concept de Daniel Abadi. Sa présentation est disponible à l'adresse suivante : http://fr.slideshare.net/abadid/cap-pacelc-and-determinism. Le propos d'Abadi est que le théorème CAP est trop simple, voire simpliste, et qu'il est principalement utilisé comme excuse pour abandonner rapidement la cohérence. Il est facile d'affirmer « Le théorème CAP dit qu'on ne peut pas partitionner, conserver de bonnes performances et maintenir la cohérence des données, donc je laisse tomber la cohérence des données ». Mais, même si on reste dans l'optique du CAP, on peut avoir la disponibilité (A, *Availability*) et la cohérence si l'on n'a pas de partition. Un système peut très bien être partitionné ou non, et donc gérer différemment la cohérence dans ces deux cas. Le modèle PACELC part donc de ce principe et s'énonce comme une forme d'arbre de décisions, représenté sur la figure 2-4.

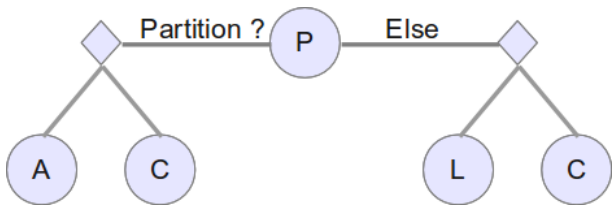


Figure 2-4
Le modèle PACELC

Le raisonnement est le suivant : si l'on est en présence d'une partition (P), le système va-t-il choisir entre la disponibilité (A) ou la cohérence (C) ? Sinon (E pour *Else*), le système va-t-il choisir entre la latence (L) ou la cohérence (C) ?

Un système comme Dynamo est de type PA/EL : en présence d'une partition, il privilégie la disponibilité au détriment de la cohérence, sinon il choisit la latence. En d'autres termes, il n'a jamais le souci de maintenir la cohérence. De l'autre côté du spectre, un moteur relationnel respectant l'ACIDité de la transaction sera PC/EC : la cohérence sera toujours privilégiée. On se dit donc qu'il manque le meilleur des choix, PA/EC, où la cohérence est privilégiée quand son coût est acceptable. Cette option n'est pas encore réellement mise en place dans les moteurs NoSQL.

Elle l'est dans des moteurs « hybrides » comme GenieDB (http://www.geniedb.com/), un moteur de stockage pour MySQL qui offre une couche de montée en charge horizontale.

Journalisation et durabilité de la transaction

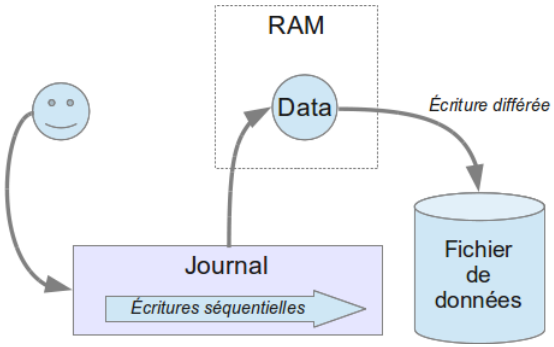
Nous avons vu dans les caractéristiques ACID de la transaction que la durabilité de cette transaction est un élément important. Il est vrai qu'il est plutôt important de ne pas perdre ses données. La durabilité semble a priori quelque chose de naturel : si la modification est écrite quelque part sur le disque, elle est donc durable, en tout cas dans les limites de la solidité du matériel. Pourquoi donc indiquer dans les caractéristiques d'une transaction qu'elle doit être durable ? En fait, dans beaucoup de SGBDR, tout comme dans ce qu'on appelle les bases de données en mémoire dans le monde SQL, les opérations de modification de données s'effectue en mémoire vive, dans un cache de données souvent appelé le *buffer*, et qui évite les écritures directement sur le disque. Les écritures dans des pages de données sont des écritures aléatoires, qui peuvent se produire à n'importe quelle position dans les fichiers de données, et ce type d'écriture est très pénalisant pour les performances.

Amplification d'écritures

Paradoxalement, les effets des écritures aléatoires sont encore plus néfastes sur les disques SSD, à cause d'un phénomène appelé l'amplification des écritures *(write amplification)*. La mémoire flash doit être effacée avant d'être réécrite. Les données sont écrites en unités nommées pages, mais effacées en plus grandes unités, les blocs. Écrire des données signifie donc : lire un bloc, écrire les données encore valides de ce bloc, puis les nouvelles données. Ce n'est pas très efficace. Les disques SSD implémentent souvent un ramasse-miettes pour diminuer ce problème, mais cela implique quand même du déplacement de données. Ensuite, selon les modèles de SSD, une distribution des écritures est réalisée pour éviter que des secteurs soient plus soumis aux écritures que les autres et atteignent prématurément leur limite de modification (une technique appelée *wear leveling*). L'effacement et l'écriture, ainsi que les déplacements éventuels dus aux mécanismes d'écriture de la mémoire flash font que la même écriture peut être physiquement multipliée. Les modèles de SSD affichent un taux d'amplification pour indiquer le nombre de fois où l'écriture est multipliée.

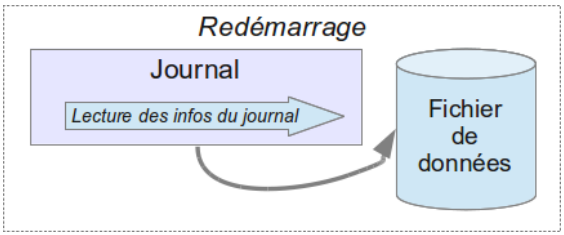
Si, pour éviter les écritures aléatoires, on reste en RAM, donc en mémoire volatile, il est nécessaire de prévoir un mécanisme d'écriture sur le disque pour éviter de perdre ses données en cas d'arrêt de la machine. Le mécanisme choisi pratiquement dans tous les cas est le système du journal. Encore une fois, la traduction française est difficile. Le mécanisme s'appelle *Write-Ahead Logging* (WAL) en anglais, qui pourrait se traduire littéralement par journalisation d'écritures anticipées. Sa représentation schématique est illustrée sur la figure 2-5.

Figure 2-5
Write-Ahead Logging



L'écriture est faite au préalable sur un support de stockage, mais sous forme journalisée, dans un fichier qui écrit séquentiellement les modifications sans se préoccuper de leur allocation finale dans un fichier de données. Il s'agit alors d'une écriture séquentielle, beaucoup moins coûteuse qu'une écriture aléatoire. La donnée est également inscrite dans un cache en mémoire de façon à pouvoir être directement manipulée par le SGBD. Admettons maintenant que la machine tombe. Lorsqu'elle redémarre, le journal est relu et les modifications appliquées dans les données, comme représenté sur la figure 2-6.

Figure 2-6
Phase de redémarrage du WAL



Cette relecture du journal au redémarrage de la base prend du temps, et empêche les données d'être immédiatement accessibles. Afin de limiter cette problématique, une synchronisation du buffer avec les données durables est effectuée à intervalles réguliers, concentrant donc les effets des écritures aléatoires sur une période de temps maîtrisée, souvent configurable.

La technologie du WAL est la plus efficace et a fait ses preuves. Par conséquent, elle est presque universellement utilisée, et les moteurs qui ne l'implémentaient pas et donc n'assuraient pas une réelle durabilité des écritures, l'ont souvent ajoutée au fil des versions. C'est par exemple le cas avec MongoDB dans sa version 1.8 sortie en mars 2011. Depuis la version 2, la journalisation est activée par défaut.

Certains moteurs comme HBase permettent d'écrire directement sur le fichier de données ou de gérer le cache dont nous venons de parler. Cette fonctionnalité s'appelle le *log flushing*, soit le nettoyage du journal en français. Prenons donc HBase comme exemple pour illustrer les différentes options de gestion du journal, qui sont assez communes aux moteurs NoSQL.

L'exemple de HBase

HBase peut écrire ou non dans le WAL. Une propriété booléenne de la classe `org.apache.hadoop.hbase.client.Mutation` dont dérivent les classes `Append` `Delete` et `Put` du client HBase pour Java est nommée `writeToWAL`. Si `writeToWAL` est à `false`, les écritures seront inscrites seulement en mémoire (dans l'équivalent d'un buffer, appelé le *memstore* en HBase). Cette option est à l'évidence une optimisation de performance, mais elle est très risquée. Aucune durabilité n'est assurée et des pertes de données peuvent donc se produire si la machine (le *RegionServer* en HBase) tombe.

En HBase, le comportement par défaut est d'écrire dans le journal, puis d'écrire physiquement la donnée juste après. Le vidage du journal peut aussi être configuré pour être différé. C'est configurable table par table en utilisant les propriétés de celles-ci dans la classe `HTableDescriptor`, qui contient une valeur `DEFERRED_LOG_FLUSH` qui peut être retournée à l'aide de la propriété `isDeferredLogFlush()` et modifiée par la méthode `setDeferredLogFlush(boolean isDeferredLogFlush)`. L'intervalle de vidage peut-être déterminé par l'option suivante du serveur de région : `hbase.regionserver.optionallogflushinterval`. Par défaut, elle est de 1 seconde (1 000 millisecondes).

Big Data et décisionnel

La logique de la croissance de l'informatisation et de la globalisation d'Internet aboutit à une augmentation formidable des volumes de données à stocker et à manipuler. Cette augmentation de volume est de plus imprévisible. Une entreprise fournissant des services ou des biens à des clients régionaux par les canaux de ventes traditionnels que sont par exemple la prospection ou les catalogues papier, peut estimer la croissance de son volume de ventes, et même un succès inespéré restera dans un volume gérable. En revanche, une entreprise fonctionnant sur Internet, qui ne délivre pas de bien manufacturé mais un service qui reste propre à l'environnement du réseau, peut connaître une croissance foudroyante et exponentielle de sa fréquentation. Elle doit être prête à soutenir une très forte augmentation de charge en un rien de temps. Pour cela, une seule solution est envisageable : une montée en charge horizontale, c'est-à-dire que la charge sera distribuée sur des serveurs disposables *(commodity servers)*, ajoutés à une architecture qui permet le plus simplement possible et de façon transparente de multiplier les nœuds.

L'exemple de Zynga

Zynga est une société californienne qui a beaucoup de succès. Elle produit des jeux sur Facebook comme CityVille et FarmVille, qui comptent des dizaines de millions d'utilisateurs. Lorsque Zynga a lancé FarmVille sur Facebook en juin 2009, la société estimait que si elle obtenait 200 000 utilisateurs en deux mois, son jeu serait un succès. Mais dès son lancement, le nombre d'utilisateurs augmenta d'un million par semaine, et ce pendant vingt-six semaines d'affilée.

Il est intéressant de noter que cette extraordinaire croissance a été possible même avec un moteur de bases de données relationnelles en stockage disque. L'architecture de Zynga, hébergée en cloud sur Amazon EC2, a pu passer de quelques dizaines de serveurs à plusieurs milliers. Le stockage est assuré dans une base MySQL, mais les données sont principalement cachées en mémoire par memcached, un système de mémoire cache distribué, et c'est là où nous retrouvons du NoSQL.

Dans le monde du Big Data, nous pouvons faire la même distinction que dans le monde des SGBDR concernant l'utilisation des données. Elle a été théorisée en 1993 par Edgar Frank Codd, entre un modèle OLTP et un modèle OLAP.

OLTP et OLAP

Ces modèles représentent des cas d'utilisation des bases de données. OLTP décrit une utilisation vivante et opérationnelle des données. Ces dernières sont en constante évolution, des ajouts, des modifications et des suppressions interviennent à tout moment, et les lectures sont hautement sélectives et retournent des jeux de résultats limités. Les bases de données constituent le socle des opérations quotidiennes des entreprises. Si on transpose le besoin dans le monde NoSQL, on peut dire que les traitements OLTP nécessitent une latence plus faible, donc de meilleurs temps de réponse, avec potentiellement un volume de données plus réduit. OLAP, ou traitement analytique en ligne, décrit une utilisation analytique, décisionnelle des données. Il s'agit de volumes importants de données historiques qui représentent toutes les données de l'entreprise, et qui sont requêtées afin d'obtenir des informations agrégées et statistiques de l'activité de l'entreprise (décisionnel, ou *Business Intelligence*), ou pour extraire des informations nouvelles de ces données existantes à l'aide d'algorithmes de traitement des données *(Data Mining)*. Dans le monde NoSQL, cette utilisation correspond à des besoins de manipulation de grands volumes de données, sans avoir nécessairement besoin de pouvoir les travailler en temps réel. En d'autres termes, une certaine latence est acceptable.

Ces deux modes d'utilisation des données ont des exigences et des défis différents. Un système de gestion de bases de données relationnelles est bien adapté à une utilisation OLTP. Lorsqu'une base relationnelle est bien modélisée, elle permet une grande efficacité des mises à jour grâce à l'absence de duplication des données, et une grande efficacité dans les lectures grâce à la diminution de la surface des données. Jusqu'à un volume même respectable, un bon SGBDR offre d'excellentes performances, si l'indexation est bien faite.

Les exigences de l'OLAP

En revanche, le modèle relationnel n'est pas adapté à une utilisation OLAP des données car dans ce cas de figure, toutes les opérations sont massives : écritures en masse par un outil d'ETL *(Extract, Transfer, Load)*, les lectures également massives pour obtenir des résultats agrégés.

Tableau 2-2. Les différences entre OLAP et OLTP

OLTP	OLAP
Mises à jour au fil de l'eau	Insertions massives
Recherches sélectives	Agrégation de valeurs sur de grands ensembles de données
Types de données divers	Principalement données chiffrées et dates
Utilisation partagée entre mises à jour et lectures	Utilisation en lecture seule la plupart du temps
Besoins temps réel	Besoins de calculs sur des données volumineuses, souvent sans exigence d'immédiateté

Les limitations d'OLAP

Le modèle OLAP est l'ancêtre du Big Data. Il a été conçu dans un contexte où le hardware restait cher et où les volumes étaient raisonnables. Il était logique de concentrer ses fonctionnalités sur une seule machine, ou un groupe limité de machines, et de construire un système intégré qui s'occupe du stockage et de l'interrogation des données à travers un modèle organisé en cubes.

Cette conception a pourtant un certain nombre de défauts quand on la ramène aux besoins d'aujourd'hui. Premièrement, elle suppose une modélisation des données préliminaire. Il faut bâtir un modèle en étoile, donc structurer ses données sous forme de table de fait et de table de dimensions. Créer un modèle préalable implique qu'on va structurer les données par rapport à des besoins prédéfinis. Si de nouveaux besoins émergent après la création du système, peut-être que la structure créée ne permettra pas d'y répondre.

Cette structuration suppose en plus qu'on doive utiliser des outils d'importation et de transformation des données (des outils d'ETL) pour préparer la donnée à l'analyse. Cela implique également qu'on ne garde pas forcément tout l'historique des dimensions par exemple, et qu'il faudra au besoin des stratégies de gestion des dimensions changeantes *(slowly changing dimensions).*

Comme on établit des relations entre une table de fait et des tables de dimensions, des liens subsistent entre les tables, ce qui rend leur distribution plus difficile. De plus, ce modèle fait et dimensions s'applique à une analyse multidimensionnelle des données, mais pas forcément à une recherche de type data mining ou machine learning. On est contraint à certains types de processus.

OLAP *vs* Big Data

Pour répondre à ces limitations, ce qu'on appelle le Big Data présente une conception différente de l'analytique. Une fois de plus, la donnée n'est pas très structurée, en tout cas il n'y a pas dans un système Big Data de schéma explicite de la donnée. L'objectif est de stocker l'intégralité des données, sans choix, sans transformation, sans remodélisation. On stocke la donnée brute, parce qu'on ne sait pas à l'avance ce qu'on va vouloir en faire, quelles sont les informations qu'on va vouloir en tirer. C'est fondamentalement le paradigme Hadoop qui nous a permis de penser de cette manière. Si on y réfléchit, Hadoop est un système qui permet de distribuer des fonctions de traitement sur un cluster de machines qui hébergent également des données.

Dans ce chapitre, nous explorons quelques différences entre le monde SQL et le monde NoSQL. Nous avons parlé de schéma explicite et du langage déclaratif qu'est SQL. Pour que ce système fonctionne, il faut prédéfinir une structure qui soit adaptée aux requêtes à travers ce langage normalisé qu'est SQL, qui travaille sur des ensembles de données structurées de façon précise. Inversement, un langage de programmation de moins haut niveau comme Java offre des usages beaucoup plus généraux, qui ne supposent pas une structuration prédéfinie de la matière avec laquelle il va travailler. Au lieu de définir une requête avec un langage de haut niveau adapté à une certaine structure de données, Hadoop est un framework où l'on développe des fonctions en Java (ou dans un autre langage du même type) qui vont donc s'adapter à la structure des données à disposition, et qui permettent d'envoyer ces fonctions directement plus proches de ces données pour effectuer un traitement. On envoie simplement des fonctions sur un cluster de machines.

Sur Hadoop, il est donc inutile de structurer la donnée. On peut très bien se contenter de fichiers plats, de type CSV par exemple, ou de tout type de structure de base de données ou de sérialisation

comme JSON ou Thrift. Il suffit de stocker au fil de l'eau les données telles qu'elles arrivent, dans toute leur richesse, et ensuite de les traiter au besoin à l'aide de fonctions, et donc de répondre à toutes les demandes futures quelles qu'elles soient.

La conception de Nathan Marz

L'un des acteurs influents du Big Data s'appelle Nathan Marz. Travaillant dans une entreprise rachetée par Twitter en 2011, il a créé une unité au sein de Twitter chargée de développer un environnement de traitement distribué, avant de quitter l'entreprise en 2013 pour fonder sa propre société. C'est le créateur d'Apache Storm, l'une des principales applications de traitement distribué en temps réel. Nous en reparlerons.

Il a mené une réflexion sur le Big Data qui a débouché sur un livre en fin de rédaction chez l'éditeur Manning (http://manning.com/marz/) et sur un concept d'architecture qu'il a appelé l'architecture Lambda. Nous y reviendrons également. Nous nous contenterons ici de parler des principes liés au stockage des données. Un environnement Big Data suivant l'architecture Lambda accueille des données immutables, selon un mode d'ajout uniquement (*append-only*). C'est l'un des fondements du système : la donnée est ajoutée dans toute sa richesse, et elle est conservée à tout jamais. Dans ce cas, la conception de la donnée ressemble à celle d'un journal (un log). C'est le seul moyen de pouvoir tracer un historique complet des données. L'absence de possibilité de mise à jour garantit également qu'il y a aucun risque de perte de données. Tous ceux qui ont déjà utilisé un moteur de base de données relationnelle ont vécu ces deux expériences : un jour, ils ont regretté de ne pas avoir structuré une table avec un historique et d'avoir perdu cette dimension au moment où on leur demandait des informations de ce type ; un autre jour, ils ont perdu des données à cause d'une mauvaise manipulation ou d'un bogue dans le programme client. Le principe de l'architecture Lambda est de ne plus jamais toucher aux données après leur insertion. Les outils de traitements batch de type Hadoop vont permettre de traiter ces grands volumes de données pour produire des résultats sous forme de vues matérialisées et donc de restructurer une copie de ces données brutes pour répondre aux besoins. Admettons qu'il y ait un bogue dans le programme de traitement qui génère ces vues matérialisées. Même si on détecte le bogue six mois après la mise en production, on sera capable de corriger le problème en supprimant les vues matérialisées, en corrigeant le bogue, et en relançant le traitement batch.

La structure des données n'est pas vraiment importante. Comme on envoie des fonctions pour traiter les données, elle peut s'adapter à tous types de format, du fichier plat au moteur de base de données. Donc on sépare vraiment ici le concept de moteur de traitement du concept de stockage. Il vaut mieux d'ailleurs stocker les données dans le format le plus universel possible, par exemple du JSON ou du CSV, de façon à être capable de le lire à partir de n'importe quel langage et dans un futur même lointain, à un moment où un éventuel format propriétaire aurait été déprécié. La structure propre à l'analyse par les outils de lecture et d'interrogation sera implémentée dans les vues matérialisées, qui peuvent donc être facilement recréées.

Décisionnel et NoSQL

Les environnements de traitement distribué de types batch, ou maintenant temps réel, vont peut-être remplacer à terme les outils OLAP du marché. En tout cas, on assiste pour l'instant à une convergence des deux approches, surtout grâce à la popularité de l'approche Big Data. Les éditeurs

de solutions décisionnelles ont tous intégré Hadoop ou un algorithme MapReduce distribué. Nous avons parlé de Microsoft qui supporte maintenant Hadoop avec une plate-forme nommée HDInsight, mais la distribution NoSQL d'Oracle peut également être utilisée avec Hadoop, avec un peu de bricolage (voir http://www.oracle.com/technetwork/articles/bigdata/nosql-hadoop-1654158.html). Un autre exemple est Vertica, un éditeur de solutions décisionnelles qui a été l'un des premiers à développer un connecteur Hadoop (http://www.vertica.com/the-analytics-platform/native-bi-etl-and-hadoop-mapreduce-integration/). Citons également Pentaho, la solution décisionnelle libre (http://www.pentahobigdata.com/ecosystem/platforms/hadoop) qui s'intègre totalement avec Hadoop, offrant un environnement graphique de création de travaux distribués. D'autres sociétés se spécialisent dans la mise à disposition d'environnements décisionnels basés sur Hadoop, telles que Datameer (http://www.datameer.com/), ou développent des interfaces pour Hadoop destinées au décisionnel. L'analytique est donc un des cas d'utilisation majeur de MapReduce.