Il y a quatre ans, un article intitulé *La fin des bases de données relationnelles*[1] nous expliquait que les serveurs de SGBDR étaient le point faible des architectures à plusieurs niveaux, et devaient donc disparaître dans un futur proche pour laisser la place aux serveurs d'applications. L'auteur[2] qualifiait même d'"*arnaque intellectuelle*" le concept de bases de données...
Force est de constater qu'il n'en est rien, mais que les technologies bien mûres des bases de données ont donné de beaux fruits, à condition de savoir les apprécier, ce qui semble encore loin d'être le cas. Explication d'un concept dont l'avenir est très prometteur.

Darwinisme et informatique : les ORM et les frameworks survivront-ils au concept de développement en base de données épaisse ?

Le concept de base de données épaisse n'est pas nouveau. Il est simplement mature, puissant, efficace et très économique... Alors pourquoi est-il largement sous employé ?
Le principal frein de cette façon de concevoir l'écriture d'application est un manque de culture et de réflexion sur ce que sont les fondements même de l'informatique. La question est simple : partant de la dichotomie données et programmes, lequel des deux éléments s'avère le plus critique pour l'entreprise ? Or combien de temps accordez-vous à la modélisation de données comme à la gestion de la qualité des données ?
Le développement épais consiste à prendre le problème à la racine en s'assurant des fondations solides afin de minimiser la construction. Il en résulte deux avantages considérables : un gain important en termes de temps de développement et une optimisation naturelle.
Retour sur des expériences très concluantes...

par Frédéric Brouard

Dans l'histoire de l'informatique il y a un certain nombre de théories et concepts sur lesquels tous les professionnels s'accordent. Il en est ainsi de la théorie de l'information de Claude

[1] p. 26 du 01 Informatique N°1821 du 30 Juin 2005 (http://www.01net.com/article/283957.html). Cet article avait été violemment critiqué dans une enfilade du forum developpez.net, visible à : http://www.developpez.net/forums/d40171/bases-donnees/decisions-sgbd/avenir-bases-donnees-relationnelles/#post276833.

[2] Dans son exemple, l'auteur s'est lourdement trompé : il présente l'expérience des transactions boursières du monde bancaire comme un exemple d'application candidate au relationnel et sauvée par l'objet. Or de la même manière que l'enregistreur de vol d'un avion n'a pas besoin d'une base de données relationnelles *et heureusement*, l'enregistrement des ventes d'actions à la corbeille ne nécessite que de stocker ce qui se passe sans que l'on ait besoin d'interroger globalement le système. C'est d'ailleurs lui même qui le dit : "*...le SGBD ne sert qu'à des fins de consultations épisodiques pour régler des litiges*". Bref, cette application est une véritable application temps réel et il aurait été parfaitement incongrue de la faire devenir lourdement relationnelle !

Shannon, de la théorie des langages (Turing, Chomsky...) et de la théorie des bases de données relationnelles d'Edgar Codd. Sur le reste, les avis vont du consensus mou (par exemple *l'intérêt des langages orientés objets*) à la divergence la plus absolue (comme *où placer le code métier*)... C'est un concentré de tous ces sujets que nous allons aborder...

1 - Histoire des bases de données[3]

Si aujourd'hui on s'accorde de fait sur l'utilisation assez systématique des bases de données relationnelles, cela n'a pas été sans mal. Les ancêtres des SGBD relationnels organisaient les données dans des espaces physiques fortement redondants constitués par des bases de données hiérarchiques. Puis vinrent les bases de données dites "réseau[4]" dont la redondance était mieux maîtrisée mais dont l'organisation du stockage reposait toujours sur des fichiers, à raison d'un fichier par "article", c'est à dire par entité dont on souhaite conserver les informations.
Les inconvénients majeurs de ces différents systèmes, notamment celles tournant autours des aspects physique de l'organisation des données posaient de multiples problèmes : difficulté de modification de la structure des fichiers en cas d'évolution du modèle, repérage de l'information par le biais d'une référence physique à la ligne dans le fichier (enregistrement), typage insuffisant. Ces impasses conduisirent Edgar Codd à trouver une solution basée sur l'ajout d'une couche logique permettant de se rendre indépendant de toute problématique physique. Mais le trait de génie d'Edgar Codd fut de considérer les données comme ensemblistes, c'est à dire n'ayant aucune relation d'ordre interne. Chris Date[5] donna à juste titre le mot de "sac" (bag) aux *relations*[6] comme analogie, et c'est à l'évidence le mot le plus juste. Il n'est qu'à regarder le contenu du sac à main d'une femme[7] pour se rendre compte qu'il n'y a vraiment aucun ordre là dedans...
Dans les années 80, la mode des concepts objets, introduits par les travaux de Barbara Liskov et popularisés par les méthodes de Grady Booch, fait fureur. On voit arriver des langages purement objet comme SmallTalk. Plus réalistes les éditeurs comme les utilisateurs leurs préféreront des langages hybrides à base de programmation impérative ajoutant la puissance de l'objet. On en arrive aux langages orientés objets modernes que l'on connaît aujourd'hui (Java, C++, C#...).
Dans le même temps, des essais plus laborieux ont lieu afin de rendre les bases de données purement objet. Hélas, il n'existe presque plus aucune trace de ces outils dont les noms ont fortement résonné sur les bancs des universités : O², Orion, GBase, VBase, Iris, GemStone... Seul véritable survivant, l'éditeur Versant a réussi à en racheter quelques-uns tandis

[3] En fait le terme de base de données apparait tardivement. On parle d'abord de banque de données, c'est à dire un système dans lequel est déposé un capital d'information sur lequel on peut faire des retraits à la demande.
[4] Ce terme parfaitement adapté à l'époque est aujourd'hui inapproprié et source de confusion. Il désigne en fait une base de données dont les "fichiers" sont agencés à la manière d'un graphe au sens mathématique du terme. N'oublions pas qu'en 1971 date de naissance des premières bases de données réseau, Internet venait de naître dans les labos, et les réseaux informatiques en étaient au stade embryonnaire. D'où le terme de bases de données réseau qui signifie en fait que les liens entre les différents fichiers forment un réseau...
[5] Sans doute le plus brillant des logiciens de la base de données.
[6] Le terme *relation* désigne un objet mathématique porteur d'information et non comme beaucoup d'informaticiens le croient un lien entre deux tables. La confusion vient du fait que le terme *relationship* désigne l'association dans le monde du modèle entité association. Une base de données relationnelle est donc une base de données constituée de relations au sens mathématique du terme, c'est à dire en fait de tables au sens physique.
[7] Mon sexisme invétéré, doublé de ma misogynie naturelle me laisse à penser que je vais avoir quelques reproches de la part des ligues de vertus féministes. Je leur présente par avances mes excuses, mais c'est l'analogie la plus vraie que j'ai trouvée à ce jour. Cependant, si mesdames m'en trouvent une meilleure, je m'engage à rectifier le tir dans mes articles futurs, avec, bien entendu mention de l'autrice !

qu'Objectivity et Caché constituent des outsiders épars. En bref, la quasi disparition de ces dinosaures n'est pas due à un cataclysme, mais plutôt à la sélection naturelle :

- les bases de données relationnelles étaient déjà matures et performantes, ce qui était loin d'être le cas des bases de données objet;
- le volume des données déjà stockées en relationnel constituait une forte inertie pour passer aux bases de données objet.
- les aspects de complexité rajoutée et la faible utilisation en pratique des techniques purement objet inhérentes à ces types de bases rendaient dubitatifs les développeurs et couteux les développements;
- l'absence d'une théorie mathématique fiable sous-jacente aux bases de données objet leur interdisait toute interopérabilité naturelle.

Tant est si bien que, de la même façon que les langages purement objet comme Smalltalk ont opéré une mutation au profit de langages hybrides fortement teintés d'objet, les bases de données relationnelles faisaient la révolution opposée en introduisant des concepts objets en leur sein. Le monde des éditeurs de SGBDR s'accorda donc autour d'une norme (SQL:1999) et aujourd'hui les SGBDR modernes comme Oracle, IBM DB2 ou SQL Server permettent de manipuler des objets au sein des tables d'une manière simple et somme toute assez efficace, *pourvu que ce ne soit pas là l'essentiel des données stockées*.
Force est de conclure que *le monde des données est encore fortement relationnel, tandis que le monde des programmes est déjà fortement objet*...
De là naît une dichotomie qui a eu lieu depuis les premiers temps de l'informatique entre les données et les programmes. **Cette dichotomie induit depuis des décennies le vieux débat consacré au placement du code métier**... et c'est maintenant cet aspect-là des choses que nous allons aborder.

2 - Abord pragmatique

Lorsque, chaque année, je donne mon premier cours sur les techniques des bases de données relationnelles, je pose à mes étudiants la question suivante : "*vous êtes un PDG et je suis un terroriste. Je vous donne le choix suivant : soit je détruis toutes vos machines et logiciels et préserve vos données, soit je détruis toutes vos données et les sauvegardes qui vont avec, et vous laisse vos machines et logiciels. Que choisissez-vous ?*"
La plupart de mes élèves s'avèrent judicieux dans leurs choix en préservant les données. Mais sont-ils de bons PDG ? On peut en douter ! En effet, la lecture du bilan d'une entreprise ne laisse aucun doute à ce sujet : on y trouvera de nombreuses lignes comptables sur l'acquisition ou la location des machines et logiciels, comme sur les salaires des développeurs ou des administrateurs. Y voyez-vous une quelconque valorisation des données ? Pour autant, nous savons tous que le capital informatique d'une entreprise est pour l'essentiel constitué par les données. Or que faites-vous pour le préserver, l'améliorer ou l'enrichir ? Avez-vous déjà mis en place un programme de gestion de la qualité ? Avez-vous pensé à établir un plan préventif de gestion de la performance ? Où se trouve la documentation d'accès aux données ?
On constate donc souvent que ces concepts sont oubliés. Mais quelle en est la raison ?

C'est en revenant sur la dichotomie données/programme que l'on peut comprendre le problème : à l'évidence, la différence essentielle qui réside entre les problématiques de données et celles de codage sont les suivantes :

- un manque de connaissance des problématiques de données dans les programmes d'éducation et de formation à l'informatique;

- l'aspect non ludique des données alors que le codage des IHM consacre la cosmétique avec aujourd'hui force strass et paillettes (Html, Flash, 3D...);
- la difficulté de mesurer la production en matière de données[8] alors qu'avec le code c'est si simple (autrefois certains informaticiens étaient payés à la ligne de code !).

Tout ceci a conduit la très grande majorité des informaticiens à une fâcheuse inculture du monde des bases de données. Un petit test révélateur fera l'affaire. Posez autour de vous la question suivante : une vue SQL peut-elle être mise à jour à l'aide d'instructions INSERT, UPDATE ou DELETE ? Vous verrez que le taux de réponse est très largement en faveur du non, alors que la solution est oui !
Comment dès lors faire confiance à l'informaticien moyen pour qu'il distille les meilleurs conseils, s'il ne connaît – espérons le *bien*[9] – qu'une seule des facettes du monde auquel il a affaire ?
C'est pourquoi la majorité des informaticiens considèrent uniquement l'aspect stockage d'une base de données au détriment des aspects de codage que sont les fonctions, procédures et déclencheurs. Constat d'autant plus vrai que certains produits populaires ayant vu le jour assez récemment (à l'échelle de l'histoire de l'informatique) étaient originellement fonctionnellement si pauvres que les techniques les plus intéressantes n'y étaient pas implantées. Donnons à titre d'exemple MySQL[10], mais c'est toujours le cas de certains pseudos SGBDR comme SQL Lite[11]...

A la fin de mes premières sessions de cours sur les bases de données, et au vu de mes démonstrations, mes étudiants de cinquième année – ils seront ingénieur dans quelques mois – sont stupéfaits de constater ce que peut faire un SGBDR moderne. *On ne pensait pas qu'on pouvait faire tout cela avec un SGBDR !* Disent-ils en chœur... Que d'années perdues !

3 - Où placer le code métier ?

Maintenant que nous savons que la base de données est bien plus intelligente que nous le pensions, voyons comment aborder ce vieux débat où les avis divergent. Soyons encore une fois pragmatiques et n'hésitons pas à expérimenter la chose.
Considérons une application Web utilisant un SGBD relationnel de haut niveau permettant d'utiliser toutes les techniques modernes. Ajoutons autant de serveurs que nous voulons...
Dans une telle configuration, on peut répartir le code en au moins trois endroits : sur le client, sur le SGBDR et enfin sur tous les serveurs intermédiaires entre les données et le client. Qu'il y ait, un, deux ou trois serveurs intermédiaires – que nous appellerons *tiers* – n'apporte aucun changement majeur, dans le sens où nous espérons que chacun des serveurs s'est spécialisé dans une tâche et que le développeur a bien fait son travail (par exemple serveur d'objet "métier" COM, DCOM, CORBA, EJB, .net, serveur de présentation pour le rendu Web, serveur d'application à distance...).

[8] Combien de fois ai-je entendu dans mes audits la phrase suivante : mon chef de projet pense que je consacre trop de temps à mettre au point une requête SQL. Il préfère que je code !
[9] A la relecture de cet article, Stéphane Faroult, auteur de "The Art of SQL", m'a glissé cet acide commentaire : "*optimisme débridé*" !
[10] D'accord, MySQL possède aujourd'hui presque tout cela. Mais c'est sur le *presque* que les choses ne vont pas, comme vous pourrez le constater en lisant la suite !
[11] Comme me le suggère Stéphane Faroult on ne devrait pas leur donner le titre de SGBDR, mais de *gestionnaire de données*, même si la présence d'un ersatz de SQL en leur sein les fait passer pour le *Canada Dry* du relationnel...

Donc le problème revient à analyser l'importance du code à mettre en œuvre aux trois points du système : un client léger ou lourd, un serveur de bases de données épais ou fin et un serveur tiers chargé ou dépouillé.
C'est ce que vient de faire Toon Koppelaars dans un article fort intéressant visible à cette adresse : http://thehelsinkideclaration.blogspot.com/2009/03/jee-and-traditional-mvc-part-2.html.
Monsieur Koppelaars est un expert en bases de données relationnelles et plus précisément un spécialiste d'Oracle. Il a maintenant sa propre entreprise[12] et a publié en tant que coauteur en 2007 un ouvrage fort intéressant[13] sur le développement en matière de bases de données, avec une approche mathématique de la chose.

Dans cette étude il nous montre qu'en fait il n'existe que sept alternatives au placement du code que voici résumées dans le présent tableau :

	Client	Tiers	SGBDR
1	Lourd	Chargé	Épais
2	Lourd	Chargé	Fin
3	Lourd	Dépouillé	Épais
4	Lourd	Dépouillé	Fin
5	Léger	Chargé	Épais
6	Léger	Chargé	Fin
7	Léger	Dépouillé	Épais
8	Léger	Dépouillé	Fin

Nous avons bien dit sept car la dernière ligne (ligne 8) est irréaliste puisque dans ce cas nous supposons pratiquement qu'aucun codage n'est possible sur aucun des systèmes...

L'approche conventionnelle nous indique que la solution la plus souvent retenue est la ligne 6, celle dans laquelle le client est léger, la base n'assure que le stockage et l'interrogation des données et que toute la logique métier est exécutée par le ou les serveurs tiers.
Ce n'est bien entendu pas la réponse la plus intéressante en terme de performances ni de souplesse de développement. Je vous laisse lire les arguments de Toon Koopelars sur la solution préférée (ligne 7), c'est à dire le développement épais côté base de données avec des clients et serveurs tiers les plus fins possible (*voir figure 1*).

[12] RuleGen : http://www.rulegen.com/
[13] Applied Mathematics for Database Professionals. Apress 2007.

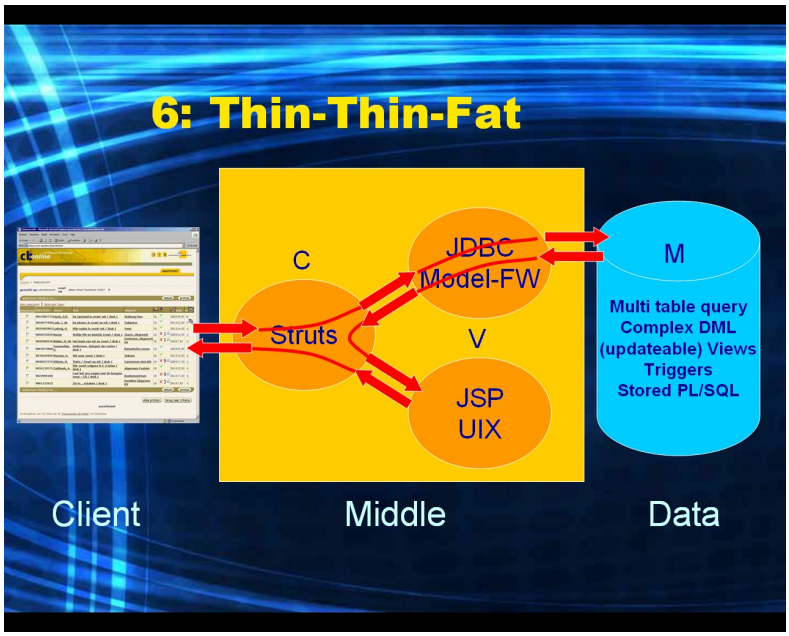

figure 1 : Approche thin-thin-fat, préconisée par Toon Koopelars (© Toon Koppelaars)

Dans un registre similaire, Paul Dorsey enfonce le clou pour nous faire comprendre en quoi l'architecture constituée par le développement bases de données épais est si intéressante.
Son article, lisible à :
http://www.dulcian.com/Articles/Thick%20Database%20Revisited_final.htm
commence par ces termes :
L'architecture de bases de données épaisse est une réponse aux défaillances catastrophiques que se posent souvent les équipes de développement qui suivent l'habitude conventionnelle de placer l'ensemble de la logique du système au niveau du tiers et utilisent la base de données comme endroit pour stocker la persistance des classes.
Et de poursuivre par
...ce n'est que récemment que l'ensemble des techniques nécessaires pour l'utiliser avec succès ont été réunies pour élaborer une architecture de développement complet.
Il s'ensuit une analyse des retours d'expériences menées par l'auteur en comparant les développements conçus par l'approche traditionnelle et celle du concept de développement épais côté bases de données.
Et le constat fait froid dans le dos...

- réduction par un facteur 3 à 4 des lignes de code client, donc réduction potentielle par ce même facteur des bugs non encore découverts;
- division par un facteur 10 à 100 des temps de réponse du système;
- réduction par un facteur 2 à 3 du temps global de développement.

Le seul inconvénient de cette approche réside dans le fait qu'il convient que les acteurs d'un tel projet maîtrisent les techniques des bases de données relationnelles, ce qui est rarement le cas !
On l'a compris, les avantages majeurs de cette approche sont la concision du code et l'absence d'aller et retour réseau entre le code métier et la base de données. Bref performance, qualité et économie...

Pour avoir été adepte de cette technique depuis des années, et cela à la manière de monsieur Jourdain, c'est-à-dire sans avoir mis un nom dessus, je peux me permettre d'apporter ma modeste conclusion : *ça pète le feu comme un foudre de guerre !*

4 - Les concepts et techniques du développement base de données épais

Tous les concepts et toutes les techniques du développement base de données épais existent depuis une dizaine d'années. C'est le temps qu'il aura fallu pour s'en rendre compte et commencer enfin à voir apparaître des compétences sur le sujet.
L'essentiel reposant sur les données, voici les différentes composantes à mettre en œuvre pour parvenir à ce résultat.

Le modèle doit être sans redondance : il faut s'assurer d'une modélisation relationnelle des données la plus consistante possible et basée sur les concepts du schéma entité association et dans le respect des différentes formes normales.

Il convient de **respecter les quatre niveaux de schéma** de la modélisation :
- conceptuel (transposition du monde réel en modèle conceptuel);
- logique (les relations au sens mathématique du terme);
- physique (les tables de la base de données);
- externes (les vues SQL qui seront utilisées par les interfaces).

L'impasse couramment commise sur le schéma externe est la pire des choses : elle interdit toute souplesse dans l'évolution de la base et conduit souvent le développeur à modéliser la base de façon médiocre, c'est-à-dire sans penser à la factorisation des entités, comme à leur découpage au plus fin.

Il faut s'interdire tout accès direct aux tables[14] et par conséquent ne passer que par des vues pour la lecture ou des fonctions tables qui permettent de faire des vues paramétrées. De la même manière on peut passer par des vues pour la mise à jour ou des procédures pour les mises à jour complexes. Bref, il est impératif de recourir systématiquement à la notion de schéma externe et de se reposer dessus pour tout accès depuis l'IHM...

Pour assurer la mise à jour des vues, sauf pour celles qui sont triviales, **on doit utiliser des déclencheurs (triggers) INSTEAD OF**[15] qui seront la plupart du temps couplées à des procédures stockées. A défaut, on peut convenir d'interdire par privilège toute mise à jour directe des tables et le faire par des procédures.

Il faut reporter la logique métier dans les procédures stockées ou les triggers. C'est le meilleur moyen de rendre ultra performants les allers et retours entre données et calculs et surtout le seul moyen s'assurer qu'une transaction soit parfaitement étanche et totalement intègre.
A ce sujet, il est navrant de constater que de nombreux architectes croient naïvement qu'un objet tiers transactionné constitue une transaction parfaite pour une base de données. Il n'en

[14] Mais il faut aussi éviter les vues de vues, de vues... C'est pourquoi personnellement je recommande de ne pas dépasser deux niveaux d'imbrication des vues : des vues de mise à plat des données et des vues orientées objets pouvant éventuellement utiliser les premières.
[15] Un trigger INSTEAD OF n'exécute pas la commande INSERT, UPDATE ou DELETE prévue, mais permet de faire croire à une vue quelconque qu'elle peut être mise à jour par ce biais. Ce sera au développeur dans le code du trigger que de répartir le traitement de mise à jour dans les différentes tables composant la vue.

est évidemment rien, et une transaction manipulant des objets n'a rien à voir avec une transaction de données, nécessitant notamment de piloter le niveau d'isolation en fonction des anomalies transactionnelles pouvant potentiellement découler du traitement. Et surtout, en utilisant une transaction dans une procédure stockée et non dans du code client, on minimise la durée de la pose des verrous sur les objets, puisqu'aucun temps mort relatif au transit par le réseau ne sera comptabilisé, ce qui contribue naturellement à une plus grande fluidité de l'application et par conséquent une meilleure montée en charge.

Les données pilotent les programmes : il faut s'assurer que l'enchaînement des traitements repose sur le résultat de données en sortie d'un précédent traitement. Pour cela tout doit être exprimé dans la base de données. Par exemple on doit faire en sorte qu'il n'existe aucune constante, autre que naturelle[16], dans les programmes.

Il faut factoriser le code autant que possible : les tables n'étant pas des objets, la façon de factoriser les appels aux données consiste à utiliser la généricité, par exemple en structurant des tables similaires avec une structure uniforme, on peut activer un code générique[17]. De manière similaire, le placement des objets composant une base de données dans différents schémas SQL y contribue fortement, tout comme l'ajout d'une norme de nommage sévère. Enfin l'introduction de la CTE[18] a permis d'économiser et de mieux structurer des requêtes complexes.

Tant qu'à faire du code lourd dans la base de données **autant y gérer la qualité des données en amont**. C'est ainsi qu'il convient de placer le maximum de contraintes ainsi que des procédures de correction automatique de saisie des données. L'avantage de contrôler la qualité des données est qu'elle contribue fortement aux performances des requêtes, en ce sens qu'extraire une donnée floue nécessite des requêtes souvent complexes, donc peu performantes et rarement optimisables[19].

Enfin démystifions à nouveau une idée reçue : le **langage de requête SQL est un langage complet**[20], c'est à dire qu'en théorie[21], partant d'un modèle de données nécessaire et suffisant au problème posé, et respectueux des règles de l'art de la modélisation, une seule requête suffit à effectuer tout traitement demandé quelque soit son niveau de complexité... reste que les développeurs sont peu habitués à passer trois jours à élaborer une requête alors que rien ne les gênent à développer pendant trois semaines la même procédure client qui fera le même traitement, mais de manière itérative, c'est à dire mille fois moins performante.

[16] Bien évidemment Pi, constituant une constante naturelle indépendante de tous les données applicatives, trouve parfaitement sa place dans la code.
[17] On me demande souvent s'il ne serait pas plus pertinent que ces tables soient fondues en une seule... Mais certaines tables aux structures identiques peuvent contenir des données quasi statiques (codes postaux par exemple) et d'autres des données fortement dynamiques (taux de change par exemple) qu'il est intéressant de placer sur différents espaces de stockage dont certains peuvent être mis en *lecture seulement*...
[18] CTE signifie Common Table Expression (expression de table en français) et permet de factoriser une partie de la requête comme étant utilisable à la manière d'une table dans la suite du code. Cette façon d'écrire qui s'apparente à une vue "*one shot*" permet de réaliser entre autres des requêtes récursives.
[19] sans parler de la lourdeur du code client réalisant de tels contrôles alors qu'avec SQL, une simple contrainte cent fois plus synthétique opère sans faille !
[20] Un langage complet au sens de Turing doit admettre la récursivité. Pourvu de tous les mécanismes ordinaires d'un langage (le traitement des boucles étant le SELECT, les tests conditionnels se faisant à l'aide du CASE), SQL permet de calculer la solution de tout problème soluble au sens de la machine de Turing.
[21] Ce que le malicieux Chris Date traduisait par la formule "*the gap between theory and practice is wider in practice than in theory*" dixit Stéphane Faroult.

Dans tous les cas, le SGBD relationnel[22] utilisé doit permettre :
- de créer des vues;
- de coder des procédures stockées, des déclencheurs et des fonctions SQL;
- la mise à jour des vues simples, directement, et celles plus complexes par le biais de triggers INSTEAD OF;
- des requêtes récursives, pas exemple à l'aide des Common Table Expression (CTE);

Il devrait en outre être capable :
- de factoriser le code SQL par le biais de la CTE;
- de mettre en œuvre les schémas SQL.

Tout le reste n'est que culture et formation.

5 - jusqu'où charger le baudet ?

Au tout début de notre article, nous avons parlé de la notion ensembliste qui constitue le fondement de la théorie de Codd. En quoi cet aspect-là des choses influence-t-il le choix d'architecture ? Il faut simplement penser qu'en développant du code itératif[23], – c'est-à-dire dans le cadre d'une programmation conventionnelle, même opérant sur des objets – l'exécution de ce dernier est strictement statique, alors que dans le monde des bases de données, du fait de la nature ensembliste des données, c'est le SGBDR qui choisit au dernier moment la meilleure façon de traiter la demande. En effet, la particularité du code SQL et des requêtes qui en sont l'essence, le traitement final sera celui que l'optimiseur aura décidé en fonction des index posés sur les données et des statistiques de distribution des valeurs dont il dispose. Mieux même... l'adaptativité du SQL et des moteurs actuels est qu'ils savent pertinemment tirer le meilleur parti du hardware et que rajouter de la RAM, du CPU ou du disque reconditionne les plans d'exécution des requêtes pour en tirer immédiatement bénéfice... Bien entendu une telle démarche est impraticable dans l'univers du code itératif, car elle supposerait que le développeur code son traitement avec tous les algorithmes connus et y rajoute une surcouche capable de décider en fonction d'un certain nombre de critères, y compris hardware, vers quel aiguillage de code lancer le traitement.
Nous devons donc conclure qu'en définitif un code ensembliste portant sur un jeu de données, même relativement faible, aura toutes les chances de battre systématiquement tout code itératif même le plus moderne qui soit. Bien entendu les différences de performances seront d'autant plus grandes que le volume des données sera plus important.

Nous devons aussi prendre en compte un autre aspect des choses. Les bases de données sont faites pour retrouver, traiter, manipuler des données. Pas pour faire des calculs. Nous devons donc nous donner une seconde règle qui consiste à peser le poids des calculs en regard de celui de l'accès aux données. Dès que le calcul va devenir fort complexe et porter sur une faible quantité de données, le code ensembliste risque de devenir moins performant que le code itératif.

De ces remarques vont naître les règles qui vont nous dicter où placer au mieux le code en termes de performance :

[22] Voici pourquoi MySQL est inutilisable dans ce contexte : pas de vues supportant les mises à jour, pas de déclencheurs INSTEAD OF, pas de CTE, pas de requêtes récursives, pas de gestion des schéma SQL...
[23] J'appelle code itératif par opposition au code ensembliste, tout code émanent d'un langage ne permettant de traiter qu'une seule donnée ou ligne de table à la fois, comme c'est le cas de la plupart des langages à objet.

Code métier :

	logique	calculs	données	localisation du code métier
1	simple	simples	unitaire	client ou tiers
2	simple	simples	plurielles	SGBDR
3	simple	complexes	unitaire	tiers
4	simple	complexes	plurielles	SGBDRO
5	complexe	simples	unitaire	SGBDR
6	complexe	simples	plurielles	SGBDR
7	complexe	complexes	unitaire	SGBDRO
8	complexe	complexes	plurielles	SGBDRO

Dans ce tableau, nous voyons apparaître la notion de SGBDRO, c'est à dire relationnel objet... Quel en est donc l'intérêt ?
Nous avons vu que les SGBDR récents et modernes étaient devenus orientés objet. C'est le sens du O qui a été ajouté à l'acronyme SGBDR. Mais l'astuce de certains éditeurs est d'avoir confié cette couche objets à un langage itératif performant, tel que Java pour Oracle ou .net pour SQL Server. Bref un code spécialement performant pour les calculs complexes (mathématiques, traitements de chaînes...). D'où la possibilité d'utiliser des procédures, fonctions ou déclencheurs codés avec un langage itératif et directement utilisable dans le code SQL. De ce fait il n'y a, à nouveau, plus les temps mort du réseau pour véhiculer les données entre le serveur de bases de données et le serveur tiers avec son code métier, puisque tout le code métier peut dorénavant tourner sur le serveur de bases de données, soit en ensembliste, c'est-à-dire avec des requêtes SQL, soit en itératif, c'est-à-dire avec le langage client supporté par le SGBDR !
Il faut cependant être parcimonieux avec cette approche car elle a son revers. Une fois de plus nous avons assisté à des développeurs qui se frottaient les mains, en pensant naïvement pourvoir écrire toutes les procédures stockées en Java ou Visual Basic ! Le O de SGBDRO n'est qu'un complément, une béquille intéressante la ou le pur transactionnel s'avère moins à l'aise et moins performant. Mais d'autres inconvénients sont présents comme le défaut d'impédance[24] ou la nécessité des changements de contexte induits par le passage du code relationnel à l'objet itératif.

Vous venez donc de comprendre que presque toute la logique métier peut être placée dans le SGBDR et que cela ne présente que des avantages.

Sur ce plan, je donne d'ailleurs une petite avance à SQL Server... En effet la particularité du code objet devant tourner au sein du serveur SQL est que pour SQL Server, ce dernier s'exécute dans une machine virtuelle sous le contrôle du SGBDR (SQL CLR) alors que dans le monde Oracle par exemple, les appels à la machine Java, comme l'exécution du code, échappent au contrôle du SGBDR. Il peut donc en résulter une instabilité du système découlant du pompage des ressources par l'un ou l'autre. Tandis que du côté de Microsoft, les exécutions de code .net sont contrôlées par le SGBDR afin qu'en cas de consommation anormalement élevée des ressources, on puisse décider de tuer le processus[25]. C'est le concept

[24] Un certain nombre de fonctions propre aux traitements relationnels n'existent pas dans l'univers itératif. C'est le défaut d'impédance. Par exemple le traitement natif des marqueurs NULLs comme la notion de *collations*.
[25] Ceci se faisant par utilisateur SQL et non globalement.

de code administrable[26] spécifique à la CLR. C'est pourquoi SQL Server intègre une machine CLR nommée SQL CLR, située à l'intérieur de l'environnement d'exécution du SGBDR, pour laquelle le serveur de bases de données exerce un contrôle total se permettant au pire des cas de virer le processus fautif dans un bac à sable !

6 – Pourquoi pas un ORM ?

Bonne idée, mais mauvaise pioche… Les outils actuels de *mapping*[27] relationnel objet, d'Hibernate à iBatis en passant par l'inabouti Entity Framework, possèdent tous les mêmes défauts. Malgré tous les caches[28] qu'on leur met, les performances sont divisées par un facteur important par rapport à une série d'insertions faites directement depuis le code client[29]. Pire encore lorsqu'on compare ce que fait l'ORM par rapport à une procédure stockée. Quant à la qualité des requêtes SQL les connaisseurs apprécieront : du fait qu'il faut être compatible avec tout un tas de SGBDR, le nivellement se fait par le bas. Mieux : pour les plus mauvais ORM, tous les arguments sont passés à titre de chaîne de caractères ce qui rend les index de la base inopérant ! Enfin, les transactions durent plus longtemps car il faut se payer les temps de communications réseau entre les serveurs, ce qui augmente la durée des verrous sur les tables et diminue donc la capacité globale du système à absorber la charge, alors que c'est un des principaux arguments de vente (cherchez l'erreur !). Enfin, peu de gens savent qu'une transaction applicative[30] n'est pas l'équivalent d'une transaction de données. Mais c'est après tout très logique... Christian Soutou, professant à l'université de Toulouse le Mirail, me montrait un tutorial de Serge Tahé[31] pour apprendre la persistance dans Java : près de 305 pages pour décrire le fonctionnement du système et les problèmes potentiels... Vous avez dit RAD ? Il est vrai qu'en SQL on ne dispose que de quelques instructions[32] pour faire la même chose avec la rapidité et la performance en plus...

En fait, à cause de ce genre d'outils et de l'unique pensée objet, un grand nombre de développeurs n'ont plus aucune vision de ce qu'est un modèle de données. Pour preuve, certains demandent sur les forums quels sont les *design pattern* pour modéliser les données d'un client... tandis que d'autre piégés par leur ORM et dans l'incapacité de savoir ce qui s'y

[26] On trouve souvent l'expression "code managé" pour désigner l'écriture de programme dans le framework .net, qui ne veut rien dire d'autre en français que code administré.
[27] Nous devrions dire *cartographiage*… mais que c'est laid !
[28] N'oublions pas qu'un SGBDR possède des caches et des stratégies de gestion de caches spécialisés dans les données et bien plus perfectionnés que ceux que l'on trouve dans les actuels ORM (cache des données, cache des procédures, cache des plans d'exécution, cache des privilèges…).
[29] Voir par exemple le test effectué par Yoran (http://www.arnumeral.fr/node/44) et l'intéressante discussion qui suivit (http://artisan.karma-lab.net/node/1568). Il y a aussi la plus sérieuse étude de Peter Van Zyl, disponible à l'url : http://researchspace.csir.co.za/dspace/bitstream/10204/984/1/van%20zyl_2006.pdf
[30] Il y a d'ailleurs souvent confusion dans l'esprit des informaticiens sur la notion de transaction qui ne s'applique qu'aux données. Selon les travaux de Gray et Bernstein une transaction doit posséder quatre propriétés : Atomicité, Cohérence, Isolation et Durabilité (ACID) et font passer un ensemble de données d'un état de cohérence à un autre en interdisant pendant la phase transitoire que les données momentanément incohérentes, puissent être utilisées d'une quelconque manière que ce soit. Trop de développeurs confondent les transactions informatiques avec les workflow de process, comme par exemple l'attente de la certitude du paiement par carte bancaire pour l'expédition d'une marchandise dans le cadre d'un site web marchand…
[31] Persistance Java 5, par la pratique de Serge Tahé sur developpez.com : ftp://ftp-developpez.com/tahe/fichiers-archive/jpa.pdf
[32] En dehors de la gestion des exceptions, le pilotage des transactions dans SQL ne repose que sur cinq commandes : BEGIN WORK, COMMIT, ROLLBACK, SET TRANSACTION ISOLATION LEVEL ..., SET CONSTRAINTS ... La persistance étant gérée automatiquement à l'aide du journal des transactions dont l'algorithme ARIES est commun à la plupart des SGBDR.

passe derrière, vont droit au mur, comme cet internaute[33] qui cherche désespérément à augmenter les ressources de PostGreSQL, afin que la requête générée par Hibernate puisse contenir plus de 1664 colonnes !
Enfonçons le clou un peu plus avec l'immédiat avenir de l'informatique que je prédis fort peu propice aux ORM et aux frameworks... Constatons qu'en matière de ressources, les ordinateurs ne peuvent plus croître sur la simple fréquence du processeur. Aujourd'hui il n'est pas physiquement possible d'aller au delà de quelques gigahertz pour les cadencer. Ceci induit que le seul moyen de faire croître la puissance de calcul est d'augmenter le nombre des CPU. Or les ORM ne disposent d'aucun mécanisme transparent pour exploiter pleinement le multi processing. Ainsi l'exécution d'un traitement complexe découlant d'une règle métier sera difficile à paralléliser. Alors que dans un SGBDR les moteurs en jeu tirent partit du maximum de CPU à disposition, et cela sans écrire la moindre ligne de code pour les en prier ! C'est la grande force de la nature ensembliste des traitements en bases de données...

Finalement les seuls arguments positifs en faveur ce ces outils sont les suivants : pour les éditeurs, cela permet de vendre de la boîte (*noire*…), pour les geeks de s'affirmer, et pour les développeurs de rester dans le concept objet sans jamais aller voir du côté de la base de données, les enfonçant ainsi encore un peu plus dans leur ignorance.
Quant à l'argument qui consiste à dire que la maintenabilité du code d'un langage client est bien plus simple que du SQL, j'ose dire qu'elle est d'une évidente stupidité : avoir 3 à 4 fois moins de lignes de code en matière relationnelle qu'avec du code itératif induit mathématiquement le fait qu'il y aura proportionnellement moins d'intervention à y faire. De plus la maintenance d'un service se fait à froid, alors que dans la base c'est nativement à chaud. Bref l'argument de facilité de maintenance, largement répandu, montre encore une fois l'étendue du désastre culturel : pour avoir, par manque de formation, rendu les développeurs inaptes au codage dans un SGBDR, pour avoir parfois choisi les mauvais outils, on a contourné le problème en y ajoutant des couches superflues, alambiquées et coûteuses, tant en licence, qu'en temps de développement ou en administration.

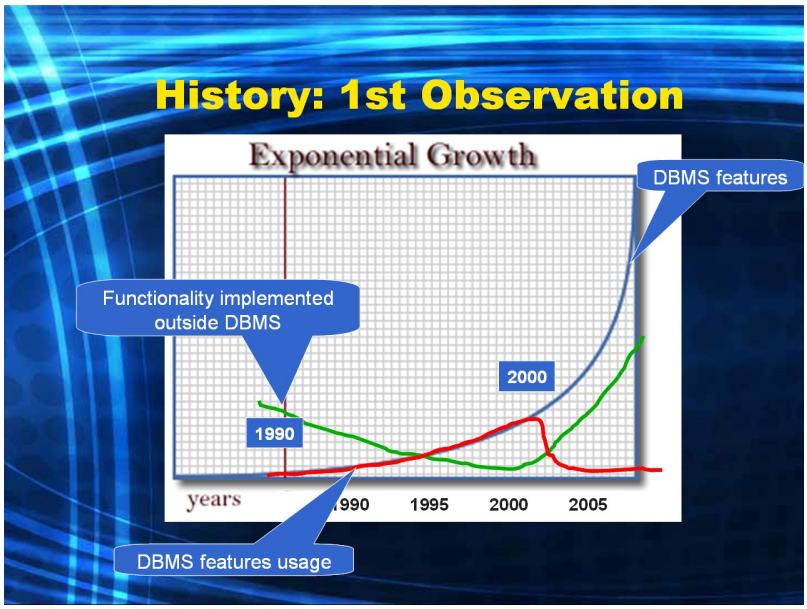

Figure 2 : historique de la montée en puissance des SGBDR et leur contre utilisation (© Toon Koppelaars)

[33] A lire : http://www.developpez.net/forums/d739003/bases-donnees/postgresql/requetes/probleme-jointures/

Toon Koopelars nous donne ce graphique si pertinent (*figure 2*). On y voit les possibilités des SGBDR croîtrent de façon exponentielle, tandis que leur utilisation commence à décroître brusquement avec la mode d'utilisation de certains outils comme les ORM ou l'arrivée de SGBD pseudo relationnels…

Cependant, je crois fermement en l'avenir de l'ORM. Pas de ceux qui existent de nos jours. Mais de celui qui sera capable de tirer parti des possibilités des SGBDR. Je rêve d'un ORM capable de créer automatiquement les vues correspondant aux objets modélisés, les procédures stockées de synthèse pour la mise à jour globale des tables relatives au mapping de l'objet au modèle relationnel, et finalement d'implémenter les déclencheurs INSTEAD OF adéquats pour assurer la correspondance entre vues et procédures … Ce n'est certes pas pour bientôt, mais c'est sans aucun doute vers cet avenir meilleur qu'il faut tendre. En attendant, c'est encore à la main que les choses se passent.

Quand aux frameworks et autres environnements de développement, constatons simplement que ces derniers changent au gré de la mode. Sitôt appris à l'école, sitôt démodés. Leur multiplicité, comme le nombre ahurissant des langages pour les piloter est la preuve évidente de leur immaturité[34]. Tout comme leurs technologies qui apparaissent tous les 6 à 9 mois et disparaissent en moins de deux ans. Bref pas mieux que les ORM, mais il faut bien hélas, une couche cliente de code...

En conclusion

Avec l'explosion des langages, ORM et frameworks, on doit se poser la question du ratio de l'investissement de formation nécessaire par rapport au code effectivement produit. Si l'on continue sur cette voie il faudra bientôt que les développeurs passent la moitié de leur temps de travail en formation, à moins finalement que l'on ne décide de les employer que quelques années juste après leur sortie de l'école et qu'on ne les jette après usage comme un kleenex...
Plus grave, cette position folle du code nouveau va à l'encontre de toute solution de pérennité pour les applications développées : il y a fort à parier que de telles applications seront de moins en moins maintenables dans le temps[35]. Mais il est vrai que la mode est au trash logiciel...
Cela serait-il dû à la prolétarisation de l'informatique comme le dit si bien Christian Fauré dans son article[36] "*la prolétarisation dans les sociétés informatiques*" ?

Les différents métiers que j'ai pratiqués m'ont appris quelque chose : on en revient toujours aux fondamentaux... Telle est la leçon qui devrait être enseignée à tout informaticien. Certes les choses évoluent, et en informatique plus vite que dans tout autre métier. Mais c'est justement pour cela qu'il est souhaitable de ne jamais perdre de vue les fondements mêmes de l'édifice informatique. Or les basiques en la matière reposent essentiellement sur les données :

[34] Citons-en quelques-uns encore en vogue : .Net, Actionscript, ADF-BC, Adobe/Flex, Ajax, Apex (html-DB), AspectJ, Blend, Clojure, Cocoa, Cocoon2, Drupal, Echo, Eclipse, EJB, Entity Framework, Erlang, F#, Grails, Groovy, GWT, GXT, Glassfish, Haskell, Hibernate, Hivemind, Html/XML, iBatis, ITmill, J2ME, Jakarta, Java, JavaFX, Javascript, Javaserver Faces, Javaserver Pages, JBoss, JDBC, JHeadstart, JPA, Laszlo, LCDS, Lift, LINQ, Maven, Maverick, Netbeans, PHP, Python, Ruby on Rails, Scala, Scheme, SilverLight, SmartGWT, Spring, Stripes, Struts, Swing, Tapestry, Toplink, UIX, WCF, Webforms, Webwork, Wicket, Xquery, ZK...
[35] Un de mes clients en a fait la cruelle expérience : l'éditeur, pour faire évoluer l'application, avait laissé faire les développeurs successifs qui ont choisi chacun l'outil le plus moderne du moment. Lorsqu'au bout de cinq ans il a fallu faire des correctifs majeurs, aucun des employés de cet éditeur n'avait plus aucune compétence pour assurer la maintenance du code...
[36] http://www.christian-faure.net/2009/03/14/la-proletarisation-dans-les-societes-informatiques/

après tout ce sont bien les données qui sont le moteur informatique de l'entreprise et non l'inverse. Une entreprise sans programmes mais avec données peut encore s'en tirer, mais sans données que saurait-elle faire de ses programmes ?
Il convient donc de tout faire autour des données, ce que certains ont appelé l'approche *data centric*, bref concevoir des modèles de données fiables et gérer la qualité des données. En y rajoutant les concepts de développement en base de données épaisse nous avons réuni toutes les garanties de pérennité, performance et fiabilité. Mais le SGBDR peut-il aujourd'hui tout faire ? En matière d'informatique de gestion, la réponse est assurément oui. Mais combien savent ce qu'est capable de faire un tel outil ? Qui sait que les SGBDR les plus modernes incorporent aujourd'hui le XML nativement, un SIG performant, un outil d'indexation textuel ou encore des services web ?
À lire certains internautes on doute encore que la connaissance de ce qu'est et ce que fait un bon SGBDR soit réellement répandue. À titre d'exemple je citerai ce post[37] : un ingénieur en qualité pratiquant l'audit d'un ERP en cours de finition s'étonnait de ne pas trouver de modèle de données ni aucune contrainte d'intégrité dans la base. Le responsable du projet de se défendre ainsi : *les bases de données énormes d'ERP n'ont généralement pas de contraintes de clef étrangère afin de faciliter les tâches de maintenance et de pouvoir adapter l'application rapidement en cas de modification des pré-requis*[38]. Mais alors pourquoi ne pas en revenir à de bon vieux fichiers style Cobol ?

Sérieusement, ne voyez-vous pas dans l'approche du tout base de données qu'il n'existe que des avantages et aucun inconvénient ? Que dire d'un langage comme SQL auquel tous les éditeurs de SGBDR sans exception ont adhéré ? Malgré cela il est étrange de constater que la pérennité du SQL va de pair avec sa méconnaissance. Sans doute n'est-il pas assez moderne d'utiliser massivement un langage efficace, parce qu'il est vieux de près de trente ans...

À un spécialiste des frameworks et ORM que j'ai réussi récemment à convaincre d'adhérer à la cause du développement en base de données épaisse pour un projet d'envergure, je posais la question suivante : quel argument vous a le plus convaincu ? Je pensais aux performances, à la pérennité du code, au temps gagné en développement, à la facilité de maintenance... Mais il m'a cloué le bec en me disant que c'était à cause de la reprise des données... Je n'y avais même pas pensé. Mais il est vrai qu'en fournissant tous les mécanismes de validation et de traitement des données côté base de données et à travers des vues, il ne suffisait plus que d'utiliser de simples jeux de requêtes pour importer les données de l'ancienne base vers la nouvelle. Bref, rien à redévelopper ! Mauvaise nouvelle pour les prestataires...

Le seul argument qui pourrait aller à l'encontre du tout SGBDR est celui de 'l'escalation'[39]. En effet, de part sa nature unique, l'information ne peut exister qu'en un seul point du système d'information, sauf à induire une redondance préjudiciable et donc à prendre les SGBDR à rebrousse poil. Contrairement donc à des serveurs d'objet ou des serveurs web, il convient d'écluser le *scale up* (se dit d'un système dont l'évolution se fait par apport interne ; par exemple les systèmes de gestion de bases de données relationnelles sont plus conçus pour une montée en charge en scale up - ajout de CPU, RAM, disque...- qu'en scale out) avant de passer au *scale out* (se dit d'un système qui évolue par apport externe ; par exemple les serveurs web - Apache, IIS... - peuvent monter en charge par scale out, c'est-à-dire par ajout en parallèle de

[37] http://www.developpez.net/forums/d742042/bases-donnees/decisions-sgbd/base-donnees-volumineuse-relations-avantage/
[38] Pourtant de nombreux outils et techniques existent pour modifier la structure d'une base avec souplesse, rapidement et sans danger... C'est la technique du refactoring de bases de données, dont tout bon outil de modélisation est capable (Power AMC, Win Design, ER Win...)
[39] Désolé pour ce néologisme, mais c'est tout ce que j'ai trouvé pour le concept de *scalability*...

nouvelles machines physiques) en matière de serveur de bases de données relationnelles. Mais il existe une façon peu connue, quoique parfaitement maîtrisée par certains éditeurs pour réaliser cela : utiliser le concept de bases de données réparties c'est à dire une architecture collaborative de données. On peut prendre l'exemple de ce que fournit Microsoft avec Service Broker, inclus en standard dans la licence SQL Server et qui assure un service de messageries transactionné, sérialisé, sécurisé et asynchrone, spécialisés dans la transmission de données entre serveurs SQL. Bref, de quoi faire du *scale out* après avoir épuisé toute possibilité de *scale up* !

Pour terminer, si vous n'avez pas encore été convaincu et si la seule fibre qui joue en vous est celle de l'écologie, posez-vous la question de savoir quel style de développement sera le plus durable...

Frédéric Brouard *est spécialiste en bases de données relationnelles et expert MS SQL Server. Il travaille pour sa propre compagnie de nom SQL spot. Il enseigne aux Arts & Métiers (CNAM PACA) et à l'école d'ingénieur ISEN de Toulon et donne des conférences à l'université de Toulouse le Mirail . Il a écrit le site français le plus visité en matière de SGBDR et SQL (http://sqlpro.developpez.com) et quelques-uns des ouvrages les plus lus sur le langage SQL. Il enseigne notamment SQL Server et la modélisation de données à Orsys.*

Travaux actuels :

Modélisation de l'ERP de Santé Service (Puteaux - 700 employés) : association d'hospitalisation à domicile. Équivalent en nombre de patient (1200) d'un CHU.

Architecte de données et modélisation de la base pour le renouvellement du SI de surveillances des crues du grand delta du Rhône (DDE du Gard).

Modélisation et développement du moteur de text mining (infométrie décisionnelle) de la société Intellixir à Manosque.