Conception et exploitation d'une base de modèles : application aux data sciences

Cyrille Ponchateau

THESE

pour l'obtention du Grade de

DOCTEUR DE L'ÉCOLE NATIONALE SUPÉRIEURE DE MÉCANIQUE ET D'AÉROTECHNIQUE

(Diplôme National — Arrêté du 25 mai 2016)

Ecole Doctorale : Sciences et Ingénierie des Systèmes, Mathématiques, Informatique (SISMI)
Secteur de Recherche : INFORMATIQUE ET APPLICATIONS

Présentée par :

Cyrille PONCHATEAU

****

Conception et exploitation d'une base de modèles : application aux data sciences

****

Directeur de thèse : **Ladjel BELLATRECHE**
Co-encadrant : **Mickael BARON**

Soutenue le 12 octobre 2018
devant la Commission d'Examen

JURY

Rapporteurs :	**Djamal BENSLIMANE**	Professeur, Université de Lyon 1
	Abdelkader HAMEURLAIN	Professeur, Université Paul Sabatier, Toulouse 3
Examinateurs :	**Mirian HALFELD FERRARI ALVES**	Professeur, Université d'Orléans
	Carlos ORDONEZ	Associate professor, Université de Houston, USA
	Thierry POINOT	Professeur, Université de Poitiers
	Samira SI-SAID CHERFI	Professeur, CNAM, Paris
	Ladjel BELLATRECHE	Professeur, ISAE-ENSMA, Poitiers
	Mickael BARON	Ingénieur de recherche, ISAE-ENSMA, Poitiers

« *Il est grand temps de remplacer l'idéal du succès par celui du service.* »

Albert Einstein

Remerciements

À l'issue de ces longues trois années, qui, pourtant, passent bien plus vite qu'il n'y paraît, il y a des personnes que je souhaite remercier.

Je commencerais par remercier mon directeur de thèse **Ladjel BELLATRECHE** qui a été le professeur avec qui j'ai découvert les bases de données, leurs intérêts et leur complexité, qui m'a présenté un sujet de thèse d'intérêt et par la même occasion l'opportunité de réaliser cette thèse et qui m'a ensuite suivi, conseillé et guidé pendant ces trois années.

Ensuite, je voudrais remercier mon encadrant de thèse **Mickael BARON** pour l'accompagnement et les nombreux conseils techniques tout au long de mon travail, pour sa disponibilité, pour son soutient moral et sa bonne humeur.

Je remercie aussi **Carlos ORDONNEZ** pour l'intérêt qu'il a porté à mes travaux, pour le temps qu'il a consacré, malgré la distance et le décalage horaire, à l'écriture de plusieurs articles et les nombreux conseils données à ces occasions.

Je remercie également les membres de mon jury : **Djamal BENSLIMANE** et **Abdelkader HAMEURLAIN** pour leur rôle de rapporteurs, **Mirian HALFELD FERRARI ALVES** en tant que présidente ainsi que mes examinateurs **Samira SI-SAID CHERFI** et **Thierry POINOT**. Merci pour cet honneur et pour les remarques très encourageantes.

Je remercie aussi le Directeur adjoint du laboratoire **Emmanuel GROLLEAU**, pour l'accueil dans le laboratoire et pour les quelques mots de soutiens, qui aident à continuer.

Je remercie également **Claudine RAULT** sans qui j'aurais eu beaucoup de mal finaliser mon dossier d'inscription en temps et en heure.

Pour avoir guidé mes premiers pas dans l'enseignement, je remercie **Laurent GUITTET**, **Brice CHARDIN**, **Stéphane JEAN** et **Allel HADJALI**.

Je remercie aussi l'ensemble des permanents du laboratoire **Amin MESMOUDI**, **Yassine OUHAMMOU**, **Henry BAUER**, **Antoine BERTOUT** et **Bénédicte BOINOT**.

Je remercie également ceux avec qui j'ai pu partager les diverses contingences du quotidien :

- ceux qui sont déjà partis : **Guillaume**, **Lahcène**, **Thomas**, **Géraud**, **Ives**, **Selma**, **Nassima**, **Okba**, **Olga**, **Zouhir** et **Nadir** ;
- ceux qui sont encore là : **Anh Toan**, **Ibrahim**, **Abdallah**, **Jorge** et **Thanh Dat** ;
- et ceux qui voyagent : **Fayçal**, **Simon-Pierre** et **Sana**.

Je remercie l'ADIIS et ses membres, et en particuliers les membres du bureau, pour les évènements et les activités qu'ils ont organisés durant ces trois années.

Et enfin, à ma famille, pour tout leur soutient, pour m'avoir redonné de l'optimisme lorsque j'en manquais.

« *C'est le devoir de chaque homme de rendre au monde au moins autant qu'il en a reçu.* ».

Albert Einstein

Table des matières

Table des sigles et acronymes **xi**

1 Introduction générale **1**
1.1 Motivation et Objectifs 4
1.2 Contributions 6
1.3 Organisation de la thèse 7
1.4 Publications 9

I État de l'Art **11**

2 Évolution de la technologie de stockage de données **13**
2.1 Les bases de données relationnelles (BDR) 14
2.2 Les entrepôt de données 23
2.3 Remise en question du standard relationnel 29
2.4 Conclusion 36

3 Modèles Mathématiques et Séries Chronologiques dans le Monde des Bases de Données **39**
3.1 La nécessité d'avoir des gestionnaires spécifiques aux séries chronologiques . . 40
3.2 Stockage des séries chronologiques dans les bases de données traditionnelles, décisionnelles et NoSql 48
3.3 Stockage de modèles mathématiques 55
3.4 Formats d'échange de modèles mathématiques 58
3.5 Conclusion 63

II Nos propositions **65**

4 Modélisation Conceptuelle et Structures de Données Dédiées aux Modèles Mathématiques dans une Approche Entreposage **67**
4.1 Équations différentielles : définition et description 68
4.2 Structure de données et ses différents formats de représentation 71
4.3 Conclusion 88

5 Exploitation des Modèles Mathématiques **89**
5.1 Les processus de conversion 89
5.2 La génération de données théoriques 97
5.3 La comparaison de données théoriques et expérimentales (comparaison série-équation) 103
5.4 Définition d'une architecture fonctionnelle générique pour la base de modèles 106
5.5 Conclusion 108

6 Prototype et expérimentations **111**
6.1 Architecture logicielle et détails d'implémentation 111
6.2 Tests expérimentaux 118
6.3 Conclusion 127

III Conclusion et perspectives **129**

7 Conclusion & Perspectives **131**
7.1 Contributions 131
7.2 Perspectives 133

IV Annexes **135**

A Résolution d'une équation différentielle linéaire d'ordre 2 **137**

B XML Schema File **139**

C Code Source Java **145**
C.1 Définition de la classe `FunctionKey` 145
C.2 Implémentation des algorithmes de traitement numérique 146
C.3 Implémentation Java des algorithmes de conversions Arbre-XML 155
C.4 Implémentation Java des algorithmes de conversions XML-Flat 160
C.5 Implémentation des processus d'insertion et d'extractions (conversions Flat-Tuple) 167

Bibliographie **183**

Table des figures

1.1 Méthode de travail scientifique 6
1.2 Séquencement des chapitres de notre Thèse 8

2.1 Exemple d'arbre hiérarchique [Cha] 15
2.2 Exemple de diagramme (E/A) 19
2.3 Architecture d'un entrepôt de données, source :[Ell13] 25
2.4 Exemple de cube OLAP [BHS+98] 27
2.5 Schéma en étoile [BFG+06] 28
2.6 Exemple de schéma en flocon [BHS+98] 29

3.1 Exemple de CD, pour la fonction `log` 60

4.1 Exemple d'arbre de représentant les NPI de 4.5 et 4.6 74
4.2 Description de la structure d'arbre et exemple 76
4.3 Exemple d'arbre non-linéaire 76
4.4 Diagramme (E/A) représentant une équation différentielle 78
4.5 Représentation XML de l'équation 4.3 85
4.6 Schéma XML de la balise `differential-equation` 87
4.7 Schéma XML de la balise `binary-operator` 87
4.8 Restriction par le schéma XML des valeurs d'un attribut 87

5.1 Série chronologique correspondant à l'équation 4.3 102
5.2 Valeurs régénérées à l'aide de la méthode de Runge-Kutta d'ordre 4 102
5.3 Exemples de séries chronologiques, représentées graphiquement 103
5.4 Diagramme de l'algorithme de comparaison 105
5.5 Processus global de gestion des séries chronologiques (Comparaison/Génération) 107
5.6 Schéma générique du gestionnaire de modèles 107

6.1 Architecture des services 112
6.2 Aperçu de la fenêtre d'exploration/affichage 114
6.3 Ajout d'une équation dans la base de modèles 115
6.4 Aperçu de la fenêtre de préparation de requêtes de comparaison 116
6.5 Sélecteur de série chronologique 116
6.6 Aperçu de la fenêtre d'avancement des opérations de comparaison 117
6.7 Affichage des résultats d'une comparaison 118
6.8 Exemple 1 120
6.9 Exemple 2 121
6.10 Exemple 3 122
6.11 Résultat pour l'équation 1 123
6.12 Résultat pour l'équation 2 124
6.13 Résultat pour l'équation 3 124
6.14 Résultat du test sur cent modèles 125

6.15 Temps d'exécution pour cent modèles 127

Liste des tableaux

2.1 Exemple de modèle relationnel . . . 17
2.2 Table R . . . 21
2.3 Table S . . . 21
2.4 Produit cartésien : $R \times S$ . . . 21
2.5 Jointure naturelle entre R et S . . . 21

3.1 Stockage relationnel de série chronologiques [DF15] . . . 48

4.1 Modèle relationnel (Schéma logique) de la base de modèles . . . 80
4.2 `Differential_Equation` table pour l'équation 4.3 . . . 82
4.3 `Node_Content` table pour l'équation 4.3 . . . 82
4.4 `has_node` table pour l'équation 4.3 . . . 82
4.5 `Initial_Value` table pour l'équation 4.3 . . . 83
4.6 `Variable` table pour l'équation 4.3 . . . 83
4.7 `has_variable` table pour l'équation 4.3 . . . 83
4.8 `Input` table pour l'équation 4.3 . . . 83

5.1 Description de l'objet `Variable` . . . 92
5.2 Description de l'objet `InitialValue` . . . 92
5.3 Description de l'objet `Input` . . . 92
5.4 Description de l'objet `Node` . . . 92
5.5 Description de l'objet `FlatDifferentialEquation` . . . 93
5.6 Table formula à partir de l'arbre 4.2b . . . 93

Table des sigles et acronymes

BD	*Base de Données*
BDR	*Bases de Données Relationelles*
CD	*Content Dictionnaries*
CDF	*Clé De Fonction*
ED	*Entrepôt de Données*
NPI	*Notation Polonaise Inverse*
PMML	*Predictive Nodels Markups Language*
SGBD	*Système de Gestion de Bases de Données*
SGBDR	*Système de Gestion de Bases de Données Relationelles*
SGSC	*Système de Gestion de Séries Chronologiques*

Chapitre 1

Introduction générale

La technologie de stockage et de gestion de données est l'une des rares disciplines scientifiques qui a contribué aux succès de nombreuses entreprises productrices et consommatrices de solutions et outils (liés à la modélisation, le stockage, l'optimisation des accès, le réglage, le déploiement des masses de données de divers types) et à la récompense de *quatre chercheurs* par le prestigieux *prix de Tuning* pour leurs découvertes et leurs engagements. Ces chercheurs sont : *Charles William Bachman* (1973), pour son modèle Codasyl, *Edgar F. Codd* (1981), pour son modèle relationnel, *Jim Gray* (1998) pour ses travaux sur le traitement des transactions et *Michael Stonebraker* (2014), pour ses contributions autour des systèmes de stockage de données modernes et sa remise en question de l'hypothèse habituelle selon laquelle « one size fits all » lors de la mise en œuvre de systèmes de stockage relationnels. Cette technologie a bien évolué depuis sa création pour satisfaire les besoins continuels des industriels et utilisateurs en termes de stockage, de gestion efficace et de qualité de service. Actuellement, le marché de cette dernière couvre trois secteurs importants selon le type de base de données ($\mathcal{BD}$) utilisé : (1) les $\mathcal{BD}$ transactionnelles de type OLTP (On-line Transaction Processing), (2) les $\mathcal{BD}$ décisionnelles de type OLAP (On-line Analytical Processing) et (3) les $\mathcal{BD}$ décisionnelles en temps réel de type RTAP (Real-Time Analytics Processing).

En nous penchant sur le secteur impliquant les applications de type OLAP, nous constatons qu'il représente actuellement un marché très fructueux générant plusieurs milliards de dollars par an. Les entrepôts de données ($\mathcal{ED}$) sont au cœur de ce secteur qui contribuent à gérer un des capitaux les plus précieux de toute organisation/entreprise représentant ses données. D'après le cabinet Gartner, le marché de l'informatique décisionnelle croît de 8,4% par an et atteindra un chiffre d'affaires mondial de 27 milliards de dollars en 2019 [DM17]. Plusieurs définitions ont été données pour le concept d'$\mathcal{ED}$. Nous retenons la définition de William H. Inmon, considéré comme le père des $\mathcal{ED}$ qui a décrit en 1992 un $\mathcal{ED}$ comme « *une collection de données orientées sujet, intégrées, non volatiles et historisées, organisées pour supporter un processus d'aide à la décision* » [Inm02b]. Un $\mathcal{ED}$ offre une perception multidimensionnelle des données collectées à partir de sources de données. Cette perception offre aux décideurs une exploitation et une exploration intuitives de l'$\mathcal{ED}$ par des décideurs à l'aide des outils OLAP, de fouille de données et des techniques de personnalisation et de recommandation [AGM+14].

L'utilisation massive des $\mathcal{ED}$ a contribué positivement à sa maturité. Cette maturité s'est traduite par la naissance d'un cycle de vie de conception comprenant six phases principales [GR09] : (a) la définition des besoins des utilisateurs, (b) la modélisation conceptuelle, (c) la modélisation logique, (d) l'ETL (Extraction, Transformation, Chargement), (e) le déploie-

ment, (f) la modélisation physique et (g) l'exploitation. Les $\mathcal{ED}$ ont généré un *écosystème* croissant impliquant cinq acteurs principaux : **(1)** les consommateurs des $\mathcal{ED}$, **(2)** les producteurs des solutions et outils dédiés aux $\mathcal{ED}$, **(3)** les établissements de formation (universités, écoles, etc.) pour satisfaire le marché du travail, **(4)** les centres de recherche scientifique, et **(5)** les développeurs de logiciels libres pour les $\mathcal{ED}$.

1. Les $\mathcal{ED}$ ont largement contribué au « Success Stories » de plusieurs entreprises qui se sont appropriées cette technologie. Le domaine de prédilection initial des $\mathcal{ED}$ était la grande distribution (par exemple le cas de l'entreprise américaine Walmart et France Billet, filière de la Fnac).
2. Les producteurs qui ont fait un travail important en proposant des solutions et des outils couvrant l'ensemble de phases de cycle de vie de conception des entrepôts sont nombreux. Nous citons l'exemple de l'entreprise Teradata offrant des solutions pour la phase de déploiement. Ses solutions de stockage, d'exploitation et de passage à l'échelle ont été choisies par plus de 100 entreprises et compagnies (par ex. Air Canada, Carrefour), avec un chiffre d'affaires de 53 millions d'euros en 2016 dans sa filière française.
3. Pour la formation, un nombre important de programmes de masters et Ph.D. nationaux et internationaux a été créé pour satisfaire le marché du travail sur les problématiques liées aux $\mathcal{ED}$s (par exemple, le programme de master et PhD. Erasmus Mundus IT4BI).
4. Quant à la recherche scientifique, des conférences thématiques sur les $\mathcal{ED}$ et analyse en ligne à l'échelle nationale et internationale ont été créées. Nous pouvons citer l'exemple de la conférence nationale EDA [1], les deux conférences internationales : DAWAK [2] et DOLAP [3].
5. Enfin, des efforts considérables ont été déployés pour développer des suites logicielles libres pour concevoir des solutions d'entreposage. Nous pouvons citer les outils Talend Open Studio et Pentaho pour la phase d'ETL.

Cette technologie a pu s'adapter à l'arrivée de nouveaux types de sources de données : les sources traditionnelles (par exemple, le système WHIPS [HGW+95]), les sources XML (par exemple, le système Xyleme [ACFR02]), les sources sémantiques (par exemple, le système OntoDaWA [XBP08]), les séries temporelles (par exemple, le projet CADDY [4] [MCV+06]), les documents [RTTZ07], les Tweets [KFK+14], les modèles de besoins fonctionnels [DBB16], les données ouvertes liées tabulaires [RS16, IHPZ14], etc.

Parallèlement à l'évolution de la technologie de stockage et de gestion de données, la naissance de *Data Science* (Science des Données) a fait apparaître de nouveaux fournisseurs et consommateurs de données scientifiques à la conquête d'analyse afin d'augmenter leur pouvoir décisionnel et capter la valeur ajoutée [KKA+17]. Des conférences et journaux liés à la science de données et leur analyse ont été récemment lancées. Nous pouvons citer l'exemple de *IEEE International Conference on Data Science and Advanced Analytics et International Journal of Data Science and Analytics*. Ces données sont issues de plusieurs domaines : les sciences expérimentales [SZ96], la médecine (l'imagerie médicale [BRK+06]), l'électrocardiogramme

1. Entrepôts de données et Analyse en Ligne créée en 2005.
2. International Conference on Data Warehousing and Knowledge Discovery, lancée en 1999.
3. International Workshop on Data Warehousing and OLAP, crée en 1998.
4. Contrôle de l'Acquisition de Données temporelles massives, stockage et modèles DYnamiques.

[LMP10, Fu11], la surveillance médicale [EA12], la finance (total des ventes hebdomadaires), la bourse [LKL03], etc.

Notre présence dans le laboratoire LIAS, qui couvre l'informatique et l'automatique, comme disciplines d'études scientifiques, nous a permis d'observer de près nos collègues automaticiens lors de la conception des expériences pour étudier les systèmes physiques complexes. Ces expérimentations génèrent un nombre important de données, souvent analysées pour déterminer leurs modèles mathématiques associés. Pour illustrer nos propos, considérons le scénario, où un chercheur observe le comportement d'une tension appliquée aux bornes d'un moteur électrique. Le moteur commence à tourner et la vitesse de rotation dépend de la tension sus-mentionnée. Le chercheur doit décrire mathématiquement la dépendance entre la tension et la vitesse de rotation. Par la suite, le moteur est testé avec des tensions différentes et l'évolution de la vitesse de rotation est mesurée par un capteur. Les mesures sont prises à un rythme fixe, générant une *série de valeurs chronologiques*, connue sous le nom de *série chronologique* [JPT17, SDDM95, Nam15, Cas06]. L'étape suivante consiste à analyser ses séries à l'aide d'un logiciel numérique [5]. Enfin, l'analyse fournit un modèle mathématique (souvent décrit par une équation différentielle) décrivant le comportement de la vitesse de rotation du moteur en fonction des variations de tension. D'un point de vue pratique, lorsque les modèles sont produits, le chercheur n'a généralement pas de structure ou de format standard pour les stocker. Par conséquent, ils finissent par être stockés dans différents formats dans de nombreux fichiers de manière désorganisée. Par exemple, les modèles sont parfois intégrés dans des fichiers scripts Matlab, R ou Python. De plus, ils peuvent être stockés sur des fichiers texte, des feuilles de calcul, des fichiers de traitement de texte, etc. *Par conséquent, la recherche d'un modèle particulier nécessite une recherche longue et fastidieuse entre différents fichiers et répertoires. En discutant avec les collègues, nous avons identifié un souhait d'avoir des solutions compréhensives et intuitives permettant la recherche et la récupération manuelles de modèles.*

Le facteur commun des données scientifiques est qu'elles sont souvent représentées par des séries chronologiques dont leur intérêt était déjà connu bien avant l'invention de l'informatique. Un détour historique est nécessaire pour comprendre leur origine et leurs usages. En effet, Dunning et Friedman [DF15] donnent comme exemple historique, celui du capitaine Matthew Fontaine Maury. Ce dernier a vécu au 19-ième siècle. Il était capitaine de bateau, jusqu'à ce qu'une blessure à la jambe le force à abandonner sa carrière. À l'époque, les capitaines avaient pour habitude de tenir un journal de bord de toutes leurs traversées, ils notaient régulièrement, la date et (même souvent l'heure) des relevés de vitesse, la latitude, la longitude, les conditions météorologiques, l'état de la mer, les animaux observés... Après avoir mis fin à sa carrière de capitaine, Maury s'est tourné vers la recherche scientifique et a pressenti l'intérêt, que pouvaient avoir ces documents. En analysant ceux, qu'il avait à sa disposition, il a pu définir des itinéraires de traversées optimaux. Il a envoyé ses recommandations au capitaine Jackson, qui a été le premier à essayer un de ses itinéraires. Ce-dernier a pu gagner 17 jours de voyage, sur un itinéraire qui durait en moyenne 55 jours et ce uniquement sur le trajet aller. Il a pu en gagner encore d'avantage au retour. Le voyage global aura donc été un mois plus court que la moyenne et à son retour au port, la surprise fut telle, que la nouvelle s'est

5. Matlab, Octave, R, etc.

répandue comme une traînée de poudre. Très vite, de nombreux capitaines ont voulu profiter des conseils de Maury. Ce-dernier leur fournissait un journal de bord standard, dans lequel il définissait l'ensemble des données dont il avait besoin pour leurs analyses. Le dit journal est un exemple de collecte de données massives (pour l'époque), basé sur le « crowdsourcing » (ce sont les utilisateurs qui fournissent les données). Dans un sens, il s'agissait d'une anticipation de ce qu'est le « Big Data » aujourd'hui.

Un certain nombre de travaux se sont intéressés au stockage et l'exploitation des séries chronologiques à travers les différentes générations de $\mathcal{BD}$ (traditionnelles, décisionnelles et NoSql). Ces derniers seront discutés dans le Chapitre 3. En examinant ces travaux, nous avons identifié qu'ils se concentrent seulement sur les séries chronologiques, et ignorent leurs modèles mathématiques associés. En interrogeant les automaticiens, nous avons identifié leur souhait d'avoir des solutions stockant à la fois les séries ainsi que leurs modèles mathématiques. Ce constat représente la motivation principale de cette thèse.

1.1 Motivation et Objectifs

La Data Science a accentué l'intérêt de la *similarité des données* identifié dans les techniques de fouille de données. Ce phénomène correspond à la théorie de similarité décrit dans plusieurs ouvrages de science de l'ingénieur [Dur08]. L'importance de cette théorie repose sur le fait qu'il est possible d'obtenir de nouvelles informations importantes sur les flux à partir de la similitude des conditions et des processus sans avoir à rechercher des solutions directes aux problèmes posés. Franz Durst dans son livre *Fluid mechanisms* [Dur08] cite l'exemple de l'auteur d'une tour ou la largeur d'une rivière peut être trouvée au moyen de considérations de similarité, sans déterminer directement la hauteur ou la largeur. À partir de considérations de similarité dans le domaine de la géométrie, on sait que l'égalité des angles correspondants ou l'égalité des rapports des côtés correspondants est une condition suffisante pour la présence de triangles, quadrangles, etc. La notion de similarité est largement présente dans les expérimentations menées par des chercheurs et scientifiques. Afin d'illustrer cette théorie, considérons l'exemple suivant.

Exemple 1.1
Si nous réalisons un swing, avec une balle de golf, cette dernière s'élèvera dans les airs et retombera quelque part dans l'herbe. Si nous arrivons à réaliser exactement le même mouvement avec la même balle, placée exactement à la même position de départ, alors la balle aura exactement la même trajectoire et retombera dans l'herbe à la même position, que la première fois (on suppose aussi que le vent, l'état du terrain, la température et la pression de l'air... n'ont pas changé non plus). Si nous modifions la balle, en changeant sa taille ou sa masse, par exemple, la trajectoire sera légèrement modifiée, mais restera similaire à la trajectoire de la première balle et la loi qui régit le mouvement de la balle permet d'anticiper, très précisément, les changements de cette trajectoire.

En automatique, lesdites lois à l'origine de la régularité des séries sont généralement repré-

sentées par des équations différentielles, que les automaticiens obtiennent après analyse des résultats expérimentaux (séries de valeurs mesurées par un ou plusieurs capteurs ou dispositif de mesure) obtenus au cours d'une expérience. Ainsi, les séries obtenues peuvent être représentées par leur modèle mathématique. L'avantage d'un modèle mathématique est la mise en évidence des informations qui sont difficiles à obtenir à partir des données brutes (séries chronologiques).

Considérons l'équation différentielle ci-dessous, qui sera davantage détaillée dans le chapitre 4, décrivant un modèle mathématique.

$$Ty' + y = Ku \tag{1.1}$$

Dans cette équation : (i) y est la fonction inconnue de l'équation, d'un point de vue automatique, il s'agit de la sortie du modèle ; (ii) u est une fonction connue et est l'entrée du modèle ; (iii) T et K sont des paramètres réels appelés, respectivement, constantes de temps et gain (T s'exprime en seconde et K est un paramètre sans unité).

Nous nous concentrerons uniquement sur le paramètre T pour l'exemple qui suit. Ledit paramètre T indique (en quelque sorte) le temps de réaction d'un système à une stimulation extérieure (par exemple, le temps que mettra un four à atteindre la température souhaitée, une fois allumé, ou le temps que mettra un moteur à répondre à une commande d'accélération ou de décélération). Ce paramètre peut être calculé à partir de données numériques, mais l'équation permet de le lire directement dans la formule. Grâce au modèle, il est possible de chercher les équations qui ont un paramètre T, compris entre deux valeurs connues ($a < T < b$). De plus, le niveau d'abstraction offert par les modèles mathématiques permet de repérer rapidement des systèmes dont le comportement est similaire, car même si les systèmes peuvent être très différents (un four et moteur électrique), si leurs comportements se ressemblent (si leurs façons de réagir à un stimulus externe sont proches), leurs modèles seront similaires.

La figure 1.1 résume une des méthodes générales de travail, en sciences expérimentales. Lorsqu'un scientifique cherche à étudier un système physique, il émet généralement des hypothèses et conçoit des expériences pour éprouver ces hypothèses (c'est-à-dire, en général, tenter de démontrer qu'elles sont fausses, une hypothèse qui résiste à de multiples tentatives, commencent à être considérée vraie). Les différentes expériences permettent d'acquérir des données, qui pour notre cas d'étude, seront des séries chronologiques. Ces séries sont acquises et stockées pour être analysées. L'analyse, qui peut être un processus itératif, fait appel à divers outils d'analyse, généralement des fonctions d'analyse statistiques. Cela permet de dégager un modèle, dans notre cas, une équation différentielle, qui permet de décrire les données et offre une meilleure compréhension des caractéristiques physiques du système étudié (par exemple, l'équation différentielle qui régit le mouvement de la balle de l'exemple 1.1 fait intervenir la masse de la balle).

Actuellement, il existe de nombreuses technologies évoluées, qui proposent des dispositifs de stockage des séries chronologiques. On peut citer TokuDB [Nam15] ou encore Vertica

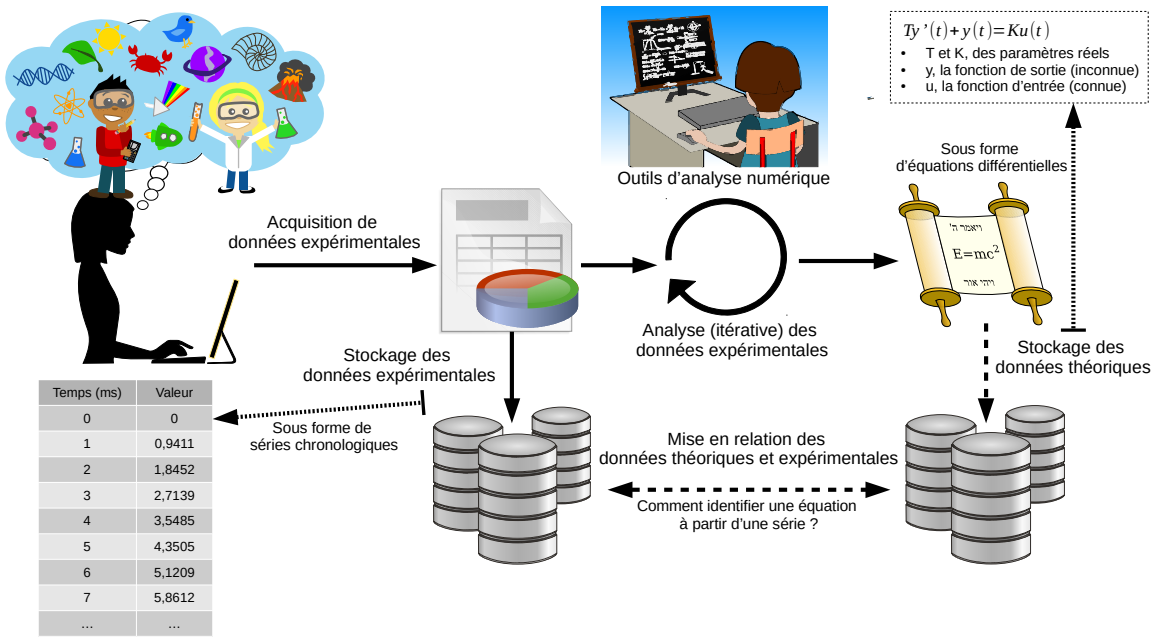


FIGURE 1.1 – Méthode de travail scientifique

[LFV+12], OpenTSDB [DF15]. En revanche, une fois le modèle conçu, il n'existe pas à notre connaissance une méthode standard bien définie permettant de formater et stocker ledit modèle. Cela entraîne l'apparition de système de stockage désordonnés. Cela rend la recherche et l'analyse de ces données complexes. De plus, le manque de format standard peut amener des problèmes d'interprétation du modèle. Une équation différentielle décrite comme une formule dans un fichier R, Matlab ou Python, peut être complexe à trouver, interpréter et réécrire de manière humainement plus lisible. Ce que nous cherchons à montrer sur le schéma de la figure 1.1, c'est que le stockage de données brutes (séries chronologiques) n'est pas suffisant pour les usages scientifiques, car le stockage se limite au stockage des données expérimentales et ne permet une gestion correcte des données théorique (que l'on peut comparer aux informations déduites à partir des données brutes, dans le domaine de la fouille de données). Ainsi, développer des systèmes de stockage et de gestion des données théoriques, avec la possibilité de faire des liens entre les données brutes et les données théoriques un enjeu important, pour le travail scientifique.

1.2 Contributions

Dans le cadre de nos travaux, nous proposons un système de stockage adapté aux modèles mathématiques correspondant à des séries chronologiques. Il aura pour objectif le stockage sous un format standard, la recherche et l'exploitation des modèles mathématiques associées à des séries chronologiques. Plus précisément, il doit répondre aux objectifs suivants :

- un système de stockage des équations différentielles, avec une couche logicielle supplémen-

taire, connectée au gestionnaire de données. Cette couche est basée sur un schéma Entité-Association, dont les caractéristiques sont inspirées des schémas en étoile ou en flocon des entrepôts de données, pour apporter une flexibilité à l'utilisateur, sur les métadonnées liées aux modèles stockées, qu'il souhaite également garder dans la base de données ;

- une couche similaire à la phase ETL (Extraction, Transformation, Chargement) du cycle de vie de conception des entrepôts de données qui permet de nettoyer les données extraites d'expériementations à l'aide des techniques issues d'analyse numérique. Deux principales méthodes sont utilisées dans notre travail : la méthode d'Euler et la méthode Runge Kutta ;

- et un système de « requête par les données », permettant de rechercher un modèle proche d'une série chronologique expérimentale, fournie par l'utilisateur, en paramètre de la requête.

Nous proposons aussi une description du fonctionnement théorique d'un tel outil et une implémentation (un prototype), afin d'aborder également quelques aspects pratiques. Le prototype que nous détaillerons dans ce manuscrit a été développé dans le but de proposer un système permettant de :

- peupler la base de modèles avec des équations différentielles fournies par l'utilisateur ;

- effectuer des requêtes « basées sur les données » qui, à partir de séries chronologiques fournies par l'utilisateur, retrouve le(s) modèle(s) le plus adapté aux données fournies.

La base de données sera aussi l'occasion de stocker des données contextuelles, liées aux équations qu'elle contient (paramètres expérimentaux, date de l'expérience, identité de(s) expérimentateur(s)...). Et ainsi permettre de conserver l'historique des expérimentations et de leurs résultats et d'assurer leur maintenance et leur traçabilité.

Pour tester notre prototype, nous avons utilisé des données issues de l'équipe d'automatique de notre laboratoire LIAS, souvent modélisées par des équations différentielles ordinaires (la définition du mot ordinaire dans le contexte des équations différentielles est expliquée dans la section 4.1) et linéaires. Ces limites supplémentaires permettent surtout d'implémenter la base de données, sans trop perdre de temps sur l'implémentation de méthodes de calculs numériques sur les équations différentielles. Ce qui peut être extrêmement complexe, lorsque nous nous aventurons en dehors des limites posées ci-dessus. Nous verrons également, que la structure de représentation choisie pour les équations différentielles (chapitre 4) permet tout de même de stocker aussi les équations non-linéaires. Nous avons travaillé sur des exemples de séries chronologiques de tailles connues finies (pas de streams de données). Les séries utilisées lors de tests n'excèdent pas 5000 éléments.

1.3 Organisation de la thèse

Ce manuscrit de thèse comprend trois parties principales comme le montre la Figure 1.2.

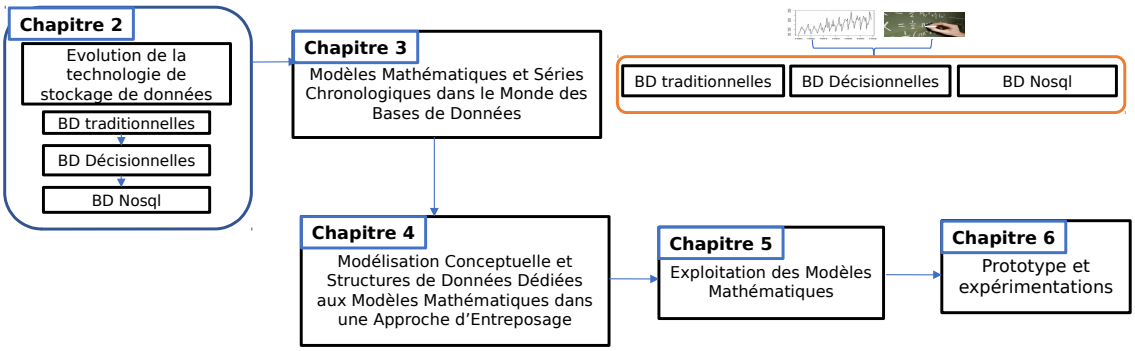


FIGURE 1.2 – Séquencement des chapitres de notre Thèse

1.3.1 Partie État de l'art

La première partie présente notre état de l'art, et s'articule autour de deux chapitres. Tout d'abord, le chapitre 2 est en lien avec notre introduction générale qui a montré l'évolution spectaculaire de la technologie des systèmes de stockage de données. Ce chapitre décrit cette évolution couvrant les bases de données traditionnelles, les bases de données décisionnelles, et les bases de données NoSql. Un intérêt particulier est porté aux entrepôts de données, du fait qu'ils sont utilisés par nos solutions.

Ensuite le chapitre 3 décrit une projection des modèles mathématiques et les séries chronologiques sur les diverses technologies de stockage de données. Cette projection montre l'intérêt de la communauté de bases de données à définir des structures de données, des formats d'échanges et méthodes de calcul dédiés à ces modèles.

1.3.2 Partie Propositions

La deuxième partie présente nos propositions et contributions, et s'articule autour de trois chapitres.

Dans le chapitre 4 nous détaillons la modélisation d'une équation différentielle et définissons les besoins, en termes de données, pour développer notre base de modèles et l'application que nous lui prévoyons (insertion et sélection des modèles et exécution de requêtes basées sur les données). Puis, nous définissons également les structures de données issues des besoins identifiés plus haut, ainsi que le schéma entité/association de la base de modèles. Nous y définissons aussi un format texte (XML), qui sers à échanger les modèles entre plusieurs programmes. Ce premier chapitre se concentre donc sur la couche « données », de la base de modèles.

Dans le chapitre 5 nous définissons les processus nécessaires au niveau logiciel. Ce qui comprend principalement, un algorithme de manipulation/résolution numérique des équations,

un algorithme permettant d'identifier une équation à partir d'une série chronologique. Nous y proposons également une architecture générique pour une base de modèles permettant de visualiser les interactions entre les différentes fonctionnalités principales de la base de modèles.

Le chapitre 6 nous décrivons le prototype que nous avons implémenté au cours de la thèse. Il s'agit d'une proposition d'implémentation de la base de modèle décrite dans les deux chapitres précédents. Ce prototype a été réalisé en Java, avec Postgresql et en respectant une architecture microservice. L'architecture microservice a été choisie pour trois raisons : (i) pour assurer le cloisonnement des différentes fonctionnalités principales et pouvoir les modifier et les faire évoluer indépendamment les unes des autres, avec un minimum d'impact sur le prototype global ; (ii) pour faciliter l'intégration/l'utilisation de technologies nouvelles ou différentes, dans les services pour lesquelles cela s'avérerait utile (par exemple, faire usage du langage Python dans les services qui font du calcul numérique, pour tirer avantage des librairies de calcul numérique puissantes disponibles dans ce langage) ; (iii) pour faciliter le passage à l'échelle du prototype, en déplaçant certains services exigeants en matière de puissance de calcul (pour la résolution numérique des équations notamment), sur des machines dédiées, ou en parallélisant les calculs à l'aide de plusieurs instances du même service réparties sur plusieurs machines. Après avoir décrit le prototype, nous décrivons les tests qui ont été réalisés sur ce prototype. Pour ces tests, nous avons utilisé trois exemples fournis par automaticiens, puis nous avons générées un ensemble des cents équations et trois-cents séries chronologiques, pour pouvoir tester sur un volume de données légèrement plus important.

1.3.3 Partie Conclusions

La dernière partie de cette thèse présente les conclusions générales de ce travail et esquisse diverses perspectives.

1.3.4 Annexes

Ce manuscrit comporte trois annexes. L'annexe A contient le détail de la résolution d'une équation différentielle ordinaire et linéaire d'ordre deux. L'annexe B contient le code XML Schema du format XML que nous avons défini pour pouvoir sérialiser nos équations différentielles. L'annexe C contient les fichiers sources (en langage Java) principaux du prototype que nous avons développé. Cette dernière annexe contient principalement le code source des processus et algorithmes décrits dans le chapitre 5.

1.4 Publications

- Un article long à la conférence EDA 2016 [PBB16] :
  C. Ponchateau, L. Bellatreche et M. Baron. Entrepôt de Données dans l'ère Data Science : De la Donnée au Modèle. *Revue des Nouvelles Technologies de l'Information*, XIIe journées francophones sur les Entrepôts de Données et l'Analyse en Ligne, RNTI-B-12 :65-80, 2016 ;

- Un article court, pour les sessions doctorales de BDA 2016 [PBOB16] :

  C. Ponchateau, L. Bellatreche, C. Ordonnez and M. Baron. A Database Model for Time Series : From a traditional Data Warehouse to a Mathematical Models Warehouse. In *Actes de la conférence BDA 2016 Gestion de Données – Principes, Technologies et Applications*, Poiters, France, 2016 ;

- Un article de démonstration, pour BDA 2017 [PBOB17] :

  C. Ponchateau, L. Bellatreche, C. Ordonnez and M. Baron. MathMOuse : A Mathematical MOdels WarehoUSE to handle both Theoretical and Numerical Data. *Actes de la conférence BDA 2017 Gestion de Données – Principes, Technologies et Applications*, Nancy, France, 2017 ;

- Un article pour le journal international de l'entreposage et la fouille de données (International Journal of Data Warehousing and Mining) [PBOB18] :

  C. Ponchateau, L. Bellatreche, C. Ordonnez and M. Baron. A Mathematical Database to Process Time Series. *International Journal of Data Warehousing and Mining (IJDWM)* : 14(3), 2018, p. 1-21.

Première partie

État de l'Art

Chapitre 2

Évolution de la technologie de stockage de données

Aujourd'hui, les bases de données (BD) interviennent dans de très nombreux aspects de notre vie quotidienne. Nous pouvons citer comme exemple : les réservations d'un voyage en train, d'une nuit d'hôtel, ou d'un vol en avion [EN10]). Mais aussi les réseaux sociaux (Facebook, Twitter, Diaspora*, Mastodon...) et les smartphones, avec de nombreuses applications, telle que Shazam l'application qui reconnaît des musiques, que OpenFoodFacts, qui, sur lecture du code à barre, donnent des infos sur la qualité d'un produit alimentaire, les applications de géolocalisation comme Google Maps ou OsmAnd... Dans [Nav92], l'auteur explique que même acheter un produit au supermarché déclenche automatiquement une mise à jour de l'inventaire du magasin.

Les (BD) interviennent dans de nombreux autres domaines, la recherche en sciences dures, comme en sciences humaines et sociales [AAB+08]. Elles sont également omniprésentes dans les entreprises, de la gestion des fonds et des clients d'une banque, à la gestion de l'inventaire d'une bibliothèque... Mais plus généralement, elles répertorient le personnel, les stocks, les transactions (achats, ventes...).

Les séries chronologiques sont également très utilisées dans de nombreux domaines, tel que la finance (le suivi des ventes, de l'état du stock[LKL03, Fu11]...), ou la médecine (les électrocardiogrammes, la surveillance et l'imagerie médicale [Fu11, LMP10, BRK+06, EA12]...) et notamment en sciences expérimentales [SZ96, LMP10]. Les séries chronologiques étant un cas particulier de données, l'évolution de leur stockage est étroitement liée à l'évolution générale des bases de données. Ce chapitre retrace l'évolution des bases de données et des technologies de stockage. Cette évolution est parsemée de nombreuses évolutions, généralement motivées par des besoins en stockage grandissant, des exigences sur la qualité du stockage de plus en plus contraignantes, et aussi l'apparition de nouveaux besoins.

Dans ce chapitre nous verrons :

- Comment sont nées les premières bases de données, avant d'aboutir aux bases de données dites relationnelles (BDR), et les technologies qui ont grandi autour, avant de voir comment le stockage de série chronologique a pu se faire à l'aide de ces bases de données ;
- Puis nous verrons comment les bases de données relationnelles sont devenues insuffisantes, face à l'appétit en données grandissant des entreprises, amenant celles-ci à évoluer, pour proposer de nouveaux outils et nouvelles technologies et comment les séries chronologiques ont profité de ces évolutions ;

- Enfin, nous verrons comment les bases de données relationnelles ont finie par perdre leur monopole, car jugé inefficaces, face à « l'avalanche de données », dit « Big Data », qui a commencé à sévir dans les années 2000 et n'a fais que se renforcer depuis. Nous y verrons, aussi, comment les séries chronologiques ont encore une fois, grandement bénéficié des nouvelles technologies qui se sont développées, pour tenir le coup, face aux Big Data.

2.1 Les bases de données relationnelles (BDR)

Les premières bases de données sont apparues dans les années 60 [Bou16]. Dans [PA16], Andrew Pavlo mentionne, entre autre, IMS, développé par IBM, comme un des tous premier projet contenant une base de données. Cette-dernière servaient à la gestion des inventaires des projets Saturne V et Apollo space. Le principe introduit par les bases de données était la séparation entre le programme informatique et les données dont il fait usage. Ainsi, les développeurs pouvaient écrire des programmes qui accèdent aux données, les lisent et les modifient, sans se préoccuper de la façon dont les-dites données sont stockées et gérées dans la mémoire.

D'après [Nav92], à l'époque des premières (BD), les logiciels géraient leurs données à l'aide de fichiers classiques. Concrètement, chaque utilisateur d'un logiciel pouvait créer les fichiers dont il avait besoin, afin d'y stocker et maintenir les données qui l'intéressaient. Mais, cela aboutissait à des redondances dans les données. Par exemple, dans le contexte d'une entreprise, le comptable a besoin de connaître entre autre, le flux de ventes, d'entrée et de sorties des stocks... Par ailleurs, pour des raisons différentes, un vendeur pourra aussi avoir besoin de connaître ces informations, pour informer au mieux un client par exemple. Ainsi, chacun des deux créera son propre fichier, contenant ces informations. Mais, si le vendeur effectue une vente, il mettra donc à jour son dossier ventes, et il lui faudra alors prévenir le comptable, pour que ce dernier mette à jour son propre fichier. Et si d'autres acteurs de l'entreprise ont également besoin de suivre cette information, il faut également les prévenir. Ainsi, la même donnée peut être répliquée dans un certains nombres de fichiers et maintenir l'information cohérente et à jour, peut être une tâche très lourde, pour l'ensemble des personnes qui dépendent de cette information. Les bases de données ont permis de centraliser l'information en un point unique, gérer par un logiciel dédié, libérant ainsi les utilisateurs de la gestion des mises à jour. Ainsi, lorsque le vendeur enregistre une nouvelle vente, si le comptable cherche à accéder à la même information, il aura directement accès aux données mises à jour, sans avoir besoin d'une notification du vendeur.

Aujourd'hui, les bases de données sont utilisées dans de nombreuses applications du quotidien, de la gestion des fonds et des clients d'une banque, à la gestion de l'inventaire d'une bibliothèque, en passant par les réservations d'un voyage en train, d'une nuit d'hôtel, ou d'un vol en avion [EN10]. Dans [EN10], les auteurs précisent d'ailleurs, que les exemples fournis ci-avant font appel à des bases de données dites traditionnelles, car les données stockées sont purement textuelles et/ou numériques. Mais l'évolution des technologies permet d'avoir des bases de données qui gère des contenus multimédias plus complexes. Ainsi, depuis leurs premières

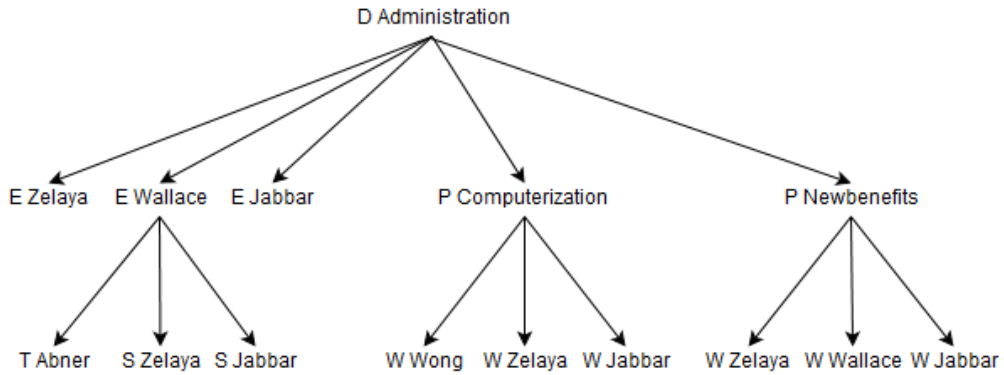


FIGURE 2.1 – Exemple d'arbre hiérarchique [Cha]

apparitions, le concept de bases de données a subit de nombreuses évolution. En particuliers, plusieurs modèles de représentation des données ont été proposés [Che76].

- Le modèle hiérarchique : en 1960, IBM a lancé son premier système de bases de données, appelé IMS (Information Management System, Système de Gestion de l'Information) [Bur20]. Il s'agissait d'une base de données dite hiérarchique. Le stockage de données reposait sur le principe de hiérarchie, telle que nous y sommes habitué dans la vie quotidienne. Par exemple, un patron, supervise une équipe de manager, qui eux-mêmes supervisent leurs équipes d'employés/techniciens.

  La base de données utilise donc une structure d'arbre (ou arborescence), qui a le mérite d'être simple et facile à implémenter [MS09]. Un arbre contient des nœuds, appelés segments. Ces segments contiennent un ensemble de champs, permettant de décrire le segment en question [Bur20, AVM15, Bou16]. Par exemple, un segment représentant un employé, pourra contenir un champ pour le nom, un autre pour l'adresse, un identifiant... Les segments sont reliés entre eux, par des liaisons 1 :N (ou one-to-many, un objet parent relié à plusieurs objets enfants), qui traduit la notion de nœuds parent unique, possédant plusieurs descendants, là où les descendants ne peuvent avoir qu'un seul parent. Une telle base de données est ainsi composée d'un ensemble d'arbres, que l'on nomme forêt.

  Le principal reproche de ce modèle est sa redondance. Observons l'exemple de la figure 2.1. Chaque élément est précédé d'une lettre : D pour département ; E pour employé ; P pour projet ; W pour travailleur (Worker) ; S pour supérieur (au sens supérieur hiérarchique dans l'entreprise) ; T pour subordonné (Dépendant, mais le D était déjà pris). Chaque segment doit avoir un unique parent. Si l'on prend l'employé Zelaya (niveau 2, tout à gauche), l'arbre nous dit qu'il s'agit d'un membre de l'administration. Mais, ce membre est aussi le supérieur hiérarchique de Wallace, et interviens dans les projets Computerization et Newbenefits. Comme un segment ne peut être connecté à plusieurs parents, on se retrouve obligé de dupliquer le segment concernant l'employé, afin de rendre compte de tout les rôles qu'il joue dans l'entreprise.
- Le modèle réseau : il s'apparente au modèle hiérarchique, sauf que l'on autorise les liaisons N :M (chaque objet peut être connecté à plusieurs autres objets). Les arbres deviennent donc des graphes [AVM15].

La navigation à travers le graphe nécessite de longs chemins d'accès, très difficiles à gérer pour les développeurs, qui, pour exploiter la base de données devaient faire face à des problèmes de gestion de mémoire de très bas niveau [Cod82]. Et cette surcharge de compétences nécessaire venait s'ajouter à l'utilisation croissante des bases de données, et des utilisateurs qui réclamaient donc de plus en plus de nouvelles applications liées aux bases de données.

Aussi, Codd [Cod70] explique, qu'à l'instar du modèle hiérarchique, les programmes et applications écrits sur la base d'un de ces modèles peuvent facilement être obsolète, si la structure de l'arbre ou du graphe change. Il faudrait alors les réécrire, à chaque changement dans la structure des données. Ainsi, dans le même article, il a introduit le modèle relationnel, qui avait pour but de séparer logiquement les données et les programmes, pour atteindre l'indépendance entre données et programmes.

- Le modèle relationnel : introduit par E.F. Codd [Cod70]. Il repose sur la définition d'une relation au sens mathématique [Cod70, Nav92]. Une relation est un ensemble (set) de n-uplets (tuples). Pour j un entier dans $[\![1..n]\!]$, tous les éléments j des n-uplets d'une même relation sont de même type.

  Les relations sont généralement représentées sous formes de tableaux, car il s'agit d'une représentation simple et naturellement comprise par l'humain [Nav92, Cod82]. Cependant, Codd insiste sur le faîte que l'ordre des lignes dans la table est totalement arbitraire, car il n'y a pas d'ordre dans une relation, en revanche, l'ordre des colonnes est essentielle, car il est représentatif de l'ordre des éléments dans les n-uplets [Cod70]. En général, les colonnes (attributs) possèdent un nom (label) afin d'aider à comprendre ce que représentent les valeurs qui y sont inscrites.

  Codd [Cod82] précise que le modèle relationnel permet d'adresser les données de manière logique. En effet, le nom de la relation permet de savoir dans quelle table chercher. Une clé primaire, permet de savoir quelle ligne lire et enfin le nom d'attribut permet de connaître la colonne. Ce qui permet aux concepteurs de référencées les données, sans connaître la structure de stockage, ni leur position en mémoire, et permet d'atteindre l'indépendance entre les données et les programmes applicatifs. De plus, cette manière de référencer les données est plus naturelle pour un être humain et peut être comprises aussi par les utilisateurs/clients des programmes, ce qui permet une meilleure communication entre les utilisateurs/clients et les développeurs.

  La table 2.1 est une représentation relationnelle possible de l'exemple présenté sur la figure 2.2. On y retrouve une table `Employé`, qui réunit des informations (attributs) concernant les employés, une table `Département` faisant de même pour les départements et une table `Projet` pour les projets. On y retrouve aussi une table `travail_dans`, qui permet de relier un employé à un département dans lequel il travaille, une table `dépend_de`, qui permet de relier un projet au département auquel il est rattaché, une table `travail_sur` qui permet de relier un employé à un projet sur lequel il travaille et une table `supervise` qui relie un employé superviseur aux employés qu'il supervise. De cette manière, l'employé Zelaya par exemple, aura une seule ligne dédié dans la table `Employé` et sera référencé dans les autres tables à l'aide de sa clé primaire `pk` (primary key), qui sert d'identifiant. Les informations concernant l'employé n'ont pas à être dupliquées dans chaque table, évitant ainsi les problèmes de redondances des données.

  Cependant, le modèle relationnel s'est vu dépassé par les exigences croissantes des utilisa-

Table	Attributes
Employé	pk; nom; prénom; date_embauche; #nom_département
Département	pk_number; nom; startdate
Projet	pk_number; nom; location
travail_dans	pk; date_integration
dépends_de	pk; #pk_département; #pk_projet
travail_sur	pk; #pk_employé; #pk_projet
supervise	pk; #pk_subordonné; #pk_superviseur

Table 2.1 – Exemple de modèle relationnel

teurs, et pas assez explicite, notamment sur les liens entre les relations [Bou16]. Exemple, je peux avoir une relation employée et une relation départements et chaque employée dépends d'un département. Donc la relation employée contiendra un lien vers la relation département, mais pour comprendre ce lien, il faut lire les attributs de la table employée et en comprendre le sens. Dans des cas plus complexes, ces liens peuvent être très difficile à repérer, et nécessité un effort de compréhension, notamment de la part des utilisateurs/clients, qui n'ont pas forcément l'habitude de déchiffrer des modèles de données.

- Le modèle entité-association : Peter Chen [Che76] a proposé le modèle entité-association, qui permettait d'expliciter les liens entre les entités et apportait ainsi une réponse au problème du modèle relationnel. Ainsi, après avoir émergé des échecs de ces prédécesseurs, c'est ce dernier qui a su s'imposer comme un standard dans le domaine des bases de données. À ce titre, la sous-section 2.1.1 lui est entièrement dédiée.

Dans les faits, le modèle relationnel reste la norme dans les bases de données, qui sont justement dites relationnelles. Mais le modèle relationnel des données, utilisé dans la base de donnée, n'est qu'une traduction du modèle entité-association, manipulé par les concepteurs. Le modèle entité-association sert à modéliser et décrire les données d'un point de vue « humain », qualifié de « haut niveau » (qui fait abstraction de la machine), tandis que le modèle relationnel sert d'intermédiaire entre l'humain et la machine. Le succès des bases de données relationnelles (BDR) a amené le développement de SGBD dédiés à ce type de bases de données, ils sont nommés SGBD Relationnels, ou SGBDR.

Les SGBDR sont équipés d'un système de transactions [Ull82, SF13, OO01]. La notion de transaction a été introduite pour régler trois problèmes majeures, que rencontraient les développeurs. (i) L'introduction d'incohérence : si deux personnes, A et B, réalisent chacune un virement sur le compte de C, il faut donc débiter A et B et créditer deux fois C. Or, il se peut qu'un bug ou une coupure de courant interrompe l'opération de mise à jour. Lorsque la machine redémarre, il n'est alors plus possible de savoir exactement à quelle opération le bug est survenu. Ainsi, il n'est plus possible de savoir si A et B ont été débités, ni de savoir si C a bien été crédité deux fois. Il faudrait donc refaire l'ensemble des opérations, en prenant le risque de détruire ou créer de la monnaie, sans en avoir conscience ; (ii) Le problème des accès concurrents : si A et B décident de réserver en même temps une place dans le même avion et qu'il n'en reste plus qu'une disponible, la place ne doit pas pouvoir être réservée

deux fois. Il faut qu'un des deux ait la place, tandis que l'autre verra sa réservation échouer ; (iii) L'écriture définitive des mises à jour : l'accès au disque dur étant très lent, les mises à jour ne sont pas réalisées au fur et à mesure, elles sont temporairement retenues en mémoire vive, puis lorsqu'il y en as un nombre jugé suffisant, elles sont écrites sur le disque, en une seule et même opération d'écriture. Ce mécanisme permet de gagner du temps, en optimisant les accès au disque, mais cela veut dire que les mises à jour sont susceptibles de restées en mémoire vive un certain temps et donc d'être perdues, en cas de coupure de courant. Par exemple, si je retire de l'argent au distributeur de ma banque et qu'une coupure de courant survient juste après l'opération, il se peut que celle-ci soit oubliée et que mon compte ne soit jamais débité, alors que les billets, eux, sont bien dans mes mains (et je jure, que j'ai pas fais exprès).

C'est là que les transactions entrent en jeu. Une transaction se définit comme une suite finie d'opérations de lecture et/ou d'écriture sur la base de données (par exemple, débiter A de x euros, crédité C de x euros, débité B de y euros et crédité C de y euros). Ces transactions respectent un ensemble de règles, une sorte de « charte », appelée ACID. ACID étant un acronyme [OO01, SF13] dont la signifie suit ci-après.

- Atomicity (Atomicité) : toutes les opérations contenues dans une transaction sont considérées comme un seul bloc. Autrement dit, si une transaction échoue (panne, bug...), toutes les opérations déjà effectuées sont annulées et doivent être refaites. Chaque transaction est donc considérée comme un tout indivisible (définition première de l'atome). Ainsi, si les virements de A et B, vers C échoue, la base de donnée est automatiquement remise dans l'état dans lequel elle était, avant le début des opérations. Celles-ci peuvent êtes refaites, sans risque de création ou destruction incontrôlée de monnaie.
- Consistency (Cohérence) : la cohérence est en fait une conséquence de l'isolation et la durabilité, décrite juste après. L'isolation permet d'éviter que deux modifications n'entrent en conflit et introduisent un bug, la durabilité assure que toutes modifications sera bien sauvegardée sur le disque sans jamais être perdu, même en cas d'accident. L'ensemble assure que les données seront toujours dans un état parfaitement connu et maîtrisé.
- Isolation (Isolation) : si deux opérations d'une transaction agissent sur une même donnée, elles ne sont jamais exécutées en même temps, mais l'une après l'autre. Plus simplement, tout est fait pour que, du point de vue d'un utilisateur lambda, les transactions sont réalisées les unes à la suite des autres, en file indienne (bien que pour des raisons d'optimisation, ce n'est pas réellement le cas). Ainsi, si A et B réserve la dernière place de leur avion en même temps, celui dont l'opération passera en première aura la place, l'autre verra sa réservation échouer.
- Durable (Durable) : si des modifications sont prévues, mais pas encore sauvegardées sur le disque (donc potentiellement perdues en cas de coupure de courant par exemple, car elles sont encore en mémoire vive), un inventaire des mises à jour à faire est maintenu et sauvegardé sur le disque, pour s'assurer que les modifications finiront toujours pas être écrites sur le disque, quoi qu'il arrive. Tout repose sur le faîte que l'inventaire est plus simple à mettre à jour (car moins volumineux que la base de données), que la base de données elle-même. Autrement dit, si je retire de l'argent de mon compte depuis un distributeur et qu'une coupure survient tout de suite après, avant que la transaction soit effectivement écrite sur

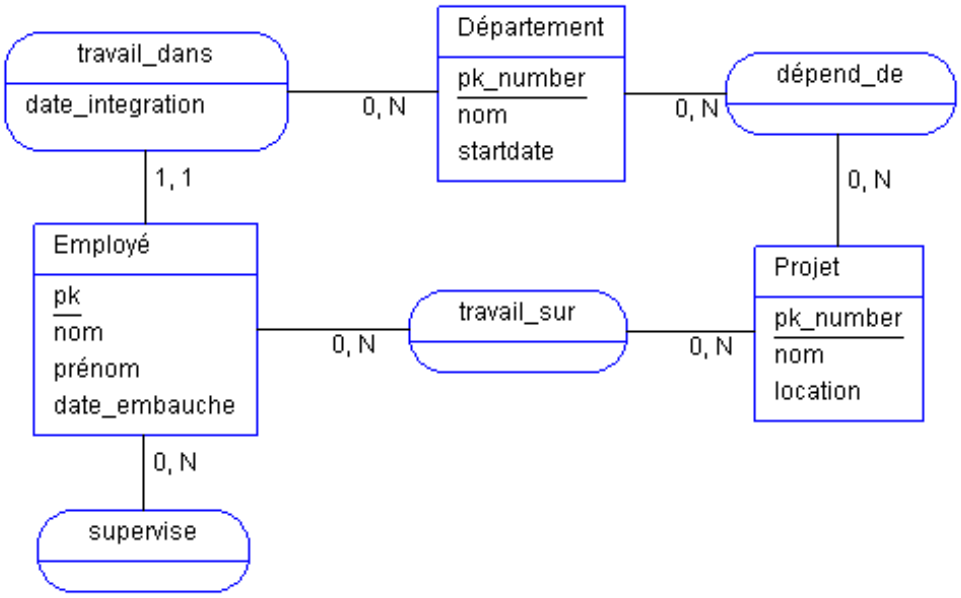


FIGURE 2.2 – Exemple de diagramme (E/A)

mon compte, je n'ai pas d'inquiétude à avoir, lorsque le courant reviendra, le système se rappellera de mon retrait, et retirera bien l'argent de mon compte (je suis rasséréné).

2.1.1 Le schéma entités-associations

Le modèle entité-association (E/A) décrit les données sous formes d'entités. Chaque entité représente une portion indépendante de la réalité (du monde) [EN10]. Une entité représente donc, par exemple, un employé, ou un département, car ce sont des concepts, qui peuvent se comprendre indépendamment l'un de l'autre. Ainsi, les concepts d'employé et de département seront représentés par les entités `Employé` et `Département`, respectivement. Les associations permettent de modéliser des liens entre plusieurs entités. Par exemple, l'appartenance d'un employé à un département est une association entre les deux entités `Employé` et `Département`. Le modèle (E/A) a ainsi l'avantage de proposer une représentation naturelle du monde (un ensemble d'objets, idées, concepts, qui interagissent entre eux). Il est donc plus compréhensible et intuitif, autant pour les utilisateurs/clients, que pour les concepteurs et/ou développeurs.

De plus, un autre gros avantage du modèle (E/A) [Che02, EN10, Bou16] c'est le diagramme (E/A). Il permet de représenter graphiquement un modèle (E/A), d'une manière lisible pour l'utilisateur. La figure 2.2 est un exemple de diagramme reprenant l'exemple utilisé sur la figure 2.1 et pour construire les tables 2.1. Ici, on peut voir qu'il existe trois entités (encadrés rectangulaires), qui sont `Employé`, `Département` et `Projet`. Ces entités sont reliées entre elles, par des associations. Par exemple, l'association `travail_dans`, associe un employé à un département. Notons que chaque entité possèdent des attributs, équivalents des attributs dans le modèle relationnel et des champs dans les segments. Cependant, si l'on prend la table

`Employé` du modèle 2.1, on note que les attributs sont les mêmes, à part que l'entité `Employé` du modèle (E/A) n'a pas d'attribut `nom_département`. En effet, cet attribut n'a pas de sens, lorsque l'on considère l'employé seulement. Il ne prend donc sens que lorsque l'on considère l'employé, un département et l'association, qui les relie. Cet attribut n'est donc pas propre à l'employé. Ainsi, le diagramme (E/A) rend de manière beaucoup plus explicite la notion de relation (de lien) entre les tables, car les liens sont directement visibles. Le diagramme est donc beaucoup plus lisible et compréhensible pour un utilisateur, même n'ont spécialiste des modèles (E/A).

Aussi, sur le diagramme (E/A), les liens sont annotés par des numéros, appelés cardinalités. Dans le cas de l'association `Employé-Département`, ce lien permet de préciser qu'un employé doit appartenir à un département et ne peut appartenir qu'à un seul département (1,1). Le nombre de gauche indique le minimum, et celui de droite, le maximum. Sur le même principe, un département, lui, peut ne pas avoir d'employés, mais peut aussi en posséder plusieurs (0,N).

Ainsi, en explicitant les liens entre les entités et en permettant d'annoter ces liens, pour y apporter des précisions sur les cardinalités, le diagramme, en plus d'être plus lisible et intuitif qu'une liste d'attributs dans un ensemble de tables, est également plus expressif (il permet de capturer davantage de sémantique).

2.1.2 L'algèbre relationnelle et le standard SQL

En 1970, Edgar Codd a posé les bases de l'algèbre relationnelle [OO01], dans son article [Cod70] (en réalité, il travaillait dans les laboratoires d'IBM et avait déjà publié c'est résultat en interne, un an avant la publication citée [Bru94]). Une algèbre étant un ensemble d'opérations, définie sur un ensemble de termes (sur un espace vectoriel) [GMUW09]. Dans le cas de l'algèbre relationnelle, les termes sont des relations et les opérations disponibles se divisent en deux catégories [Cod70, Cod72] :

- Les opérations héritées des ensembles (sets), car les relations sont des cas particuliers d'ensembles ;
- et les opérations spécifiques aux relations.

Les opérations sont les suivantes [Cod72, Gar94, GMUW09, Ull82] :

Les opérations ensemblistes

- La différence : notée $R - S$, il s'agit de l'ensemble des tuples de R, qui ne sont pas dans S. R et S doivent être de même arité (même ensemble d'attributs) ;
- L'union : notée, $R \cup S$, il s'agit de l'ensemble des tuples qui sont ou dans R, ou dans S, ou les deux. Là aussi, R et S doivent être de même arité ;
- L'intersection : notée $R \cap S$ et équivalente à $R - (R - S)$, il s'agit de l'ensemble des tuples qui sont à la fois dans R et S. La même arité étant, là aussi, requise ;

key	num
1	100
2	200

TABLE 2.2 – Table R

key	nom
1	A
2	B

TABLE 2.3 – Table S

R.key	num	S.key	nom
1	100	1	A
1	100	2	B
2	200	1	A
2	200	2	B

TABLE 2.4 – Produit cartésien : $R \times S$

key	num	nom
1	100	A
2	200	B

TABLE 2.5 – Jointure naturelle entre R et S

- Le produit cartésien : noté $R \times S$, réunis toutes les combinaisons possibles des tuples de R et S. En prenant les tables R et S données en exemple 2.2 et 2.3, on obtient la table 2.4.

Les opérations relationnelles

- La projection : notée $R[A]$ ou bien $\pi_A(R)$, elle se définit formellement par la formule : $R[A] = \{r[A] : r \wedge R\}$. A étant une liste des (ou d'une partie des) attributs de R et la projection de R sur A consiste à réorganiser les attributs de R selon A. Autrement dit, les attributs de R qui n'apparaissent pas dans A sont supprimés et si l'ordre des attributs données dans A est différents de celui des attributs R, alors il est modifié pour respecter l'ordre imposé par A.
- La sélection : notée $\sigma_C(R)$. C est un ensemble de conditions (également appelées prédicats). L'opération consiste à supprimer de R, tous les tuples, qui ne vérifient pas la condition C.
- La jointure « naturelle » (Natural join) : notée $R \bowtie S$. Elle consiste, dans un premier temps, à prendre le résultat du produit cartésien entre R et S. Ensuite, si R et S partage des attributs commun, alors parmi toutes les combinaisons de tuples, seuls les tuples dont les attributs sont égaux sont gardés. Enfin, les attributs communs étant égaux dans chaque tuple, il n'est pas nécessaire de les garder en doublon, une projection permet alors de supprimer les doublons, pour ne garder qu'un seul exemplaire des attributs partagés. Observons la table 2.5. Elle a été construite à partir de la table 2.4. Les attributs `key` de R et S étant les mêmes, seuls les tuples, pour qui ces attributs étaient égaux, ont été gardés. Puis, une des deux colonnes `key` a été supprimée (par projection), puisque cela créait une redondance dans la table.
- La jointure « théta » : notée $R \bowtie_C S$, équivalente à $(\sigma_C(R \times S))[L]$. Il s'agit du résultat de la jointure naturelle entre R et S, auquel on applique une sélection sur C. Elle est appelée « théta »-join, en anglais, car C étant une liste de condition, il arrive que les éléments de C soient décrit de la manière suivante : $i\theta j$. Où i et j sont des termes et θ (lettre grecque, nommée théta) est un opérateur (tel que : $=$; $<$; $>$...). Ainsi, l'opération se note parfois : $R \underset{i\theta j}{\bowtie} S$. Par exemple, en sélectionnant uniquement les tuples pour lesquels on a $num = 200$, sur la table 2.5, cela donnerait une table ne contenant plus que le tuple 2 de la table 2.5.

La table obtenue serait le résultat d'une jointure « théta » sur R et S où la condition C serait : $num = 200$.

Une expression de l'algèbre relationnelle, telle que $(\sigma_C(R \times S))[L]$ est une requête [GMUW09]. Les langages qui implémentent l'algèbre relationnelle, sont appelés langage de requêtes.

Edgar Codd a démontré que l'algèbre relationnelle était complète, c'est-à-dire que toutes opérations définies sur un ensemble fini de relations est une combinaison des opérations de l'algèbre linéaire, qu'il a introduite [Cod72]. Ainsi, dans [GMUW09, OO01, Ull82], les trois auteurs démontreront que certaines opérations de l'algèbre sont elles-mêmes des combinaisons d'autres opérations définie dans l'algèbre (comme nous avons pu le voir dans le descriptif ci-dessus), dans le but d'isoler les opérations dites élémentaires, dans le sens ou toute opérations qui n'est pas élémentaire, est une composition d'opérations élémentaires. Ce qui permet de définir un ensemble d'opération minimal, que tous langage de requête, doit être capable de réaliser, pour être complet (permettre de réaliser toutes les opérations possibles, sur les relations). Aussi, l'algèbre relationnelle est un standard d'évaluation des langages de requêtes [Ull82], car il est facile de vérifier l'implémentation correcte de chaque opération élémentaire, pour justifier la complétude d'un langage. Cependant, en pratique, les opérations telle que les jointures, par exemples, qui ne sont pas élémentaires, mais très souvent utilisée, sont généralement proposées en plus de l'ensemble des opérations élémentaires. Elles sont donc implémentées uniquement pour le confort de l'utilisateur [Ull82, OO01].

Le langage SQL est basé sur l'algèbre relationnelle et est devenu le langage de requête standard dans les bases de données relationnelles [w3s18]. L'exemple, qui suit, est un exemple de requête SQL, qui permet de sélectionner les employés du département administratifs, embauché avant une certaine date d, dans une base de données dont le schéma est celui donné sur la figure 2.2 :

Exemple 2.1

```
SELECT Employe.nom, Employe.prenom,
Employe.date_embauche, travail_dans.date_integration
FROM Employe, Departement, travail_dans
WHERE travail_dans.employe_pk = employe.pk
AND travail_dans.departement_pk = departement.pk
AND employe.date_embauche < d
AND departement.nom = 'administration'
```

Les quatre dernières lignes, permettent de préciser les conditions (prédicats) que les données doivent respecter. Ici, la date d'embauche de l'employé doit être inférieur à la date d, donnée par le créateur de la requête et le nom de département doit être « administration », puisque ce sont les employés de l'administration, qui nous intéressent. La première est un peu plus complexe à détailler. D'abord, le mot clé `FROM`, permet d'indiquer les tables dans lesquelles chercher l'information, ici `Employé`, `Département` et `travail_dans`. Le mot clé `FROM`, réalise un produit cartésien entre les tables. La table `travail_dans` possède un attribut `employe_pk`, qui est équivalent à l'attribut `pk` de la table `Employé`. Le premier prédicat (ligne deux) permet de sélectionner les combinaisons de tuples, qui réunissent des informations concernant le même employé. De même, le second prédicat, permet de sélectionner les combinaisons de tuples, qui

concernent les mêmes départements. Une fois les jointures, suivies des sélections réalisées, une projection est faite, pour ne garder que les attributs listés entre le mot clé `SELECT` et `FROM`.

En résumé, cette requête consiste en deux produit cartésien $Employe \times travail\_dans \times Departement$, suivi d'une sélection sur une condition C, contenant quatre prédicats. Suivi d'une projection. Ici, les jointures ne sont pas tout à faits des jointures naturelles, car la projection ne se contente pas de supprimer les doublons d'attributs entre les tables. Par exemple, les attributs `nom` et `startdate` de Département, ne sont pas des doublons, mais la projection les supprime tout de même. Cela se justifie ici, car ce sont les employés du département administration qui nous intéressent. Or, la dernière condition assure que les employés sont du département administration. Ainsi, il n'est pas nécessaire de conserver les informations concernant le département dans le résultat de la requête, puisque ce sera les mêmes informations, dans tous les tuples.

2.2 Les entrepôt de données

La philosophie des entrepôts de données (ED) trouvent son origine au début des années 80 [BHS+98], lorsque les outils de gestion des bases de données sont devenus des produits commerciaux. Il était alors possible d'extraire des données d'une base de données, dans le but de les analyser et en tirer des conclusions. Mais les analystes des données sont rarement les fournisseurs de ces-dernières [EN10]. Dans une entreprise, par exemple, ceux qui génèrent des données sont les vendeurs à chaque nouvelle vente ou commande enregistrée. De nos jours, les ventes et commandes peuvent être générées automatiquement via une boutique en ligne (à l'instar d'Amazon). Mais les analystes seront les décisionnaires de la boîte, qui ont besoin de connaître les tendances et l'évolution de cette-dernière, pour choisir de nouvelles lignes directives. Dans un autre registre, les sites de réservation de vol en avion par exemple, vont fouiller les bases de données de plusieurs compagnies, auxquels le site n'appartient pas, pour afficher à l'utilisateur, les vols qui correspondent à ces critères, généralement triés du moins chers au plus cher. Ainsi, ceux qui cherchent à analyser les données, n'ont pas pour rôle de les modifier, ils se contentent de les consulter (de les lire). L'analyse des données sert généralement de support décisionnaire, comme nous l'avons vue pour les décideurs d'une entreprise ou, dans le cas du futur voyageur, qui voudra choisir le vol qui lui convient le mieux.

Les entrepôts de données permettent de créer un historique complet des données, plutôt que d'écraser les anciennes par des nouvelles [BHS+98]. Comme il s'agit d'un historique, les données sont toujours (d'une manière ou d'une autre) associées à une période temporelle (la date d'une vente, le chiffre d'affaires réalisé au cours d'une année...) [Inm02a]. L'historique permet de réaliser des analyses plus fines, sur une base d'information plus complètes et d'en tirer davantage de connaissances. Ainsi, les entrepôts ont pour but de centraliser des données venant de sources diverses et variées, afin d'en offrir une vision unifiée à l'utilisateur et lui donner ainsi accès à un volume de données suffisamment large (on passe de volume de l'orde du méga ou du giga-octet dans les années 80, à des volumes de l'ordre du téra ou du pétaoctet (1 000 téraoctets), dans les années 90 [Bou16]), pour couvrir ses besoins (prendre une décision

cohérente par rapport à l'évolution de l'entreprise ; avoir le choix d'un vol suffisamment adapté à ses besoins et à son porte-feuille...).

2.2.1 Architecture

La particularité des entrepôts de données vient donc du faîte qu'il s'agisse d'un outil permettant de centraliser des données déjà stockées dans d'autres sources de données. Ce qui génère une difficulté supplémentaire (en plus du volume), car les-dites sources de données sont hétérogènes, dans le sens où elles peuvent être très différentes les unes des autres [Ada06]. Il peut s'agir de différences de format (fichiers XML ou autres formats textuels, d'autres bases de données...), et s'il s'agit du même format, du simple fait que leur provenance n'est pas la même. Il peut s'agir d'un ensemble de systèmes de sauvegarde issues des divers logiciels utilisés dans une entreprise pour suivre quotidiennement l'activité ou un type d'activités de la-dite entreprise [Ell13]. Ou encore, dans le cas des sites de réservation de vols en avion, chaque compagnie possède sa propre base de données, construites selon des règles et des visions différentes, par des personnes différentes. Il n'y a donc aucune raison pour que les schémas des bases soient les mêmes, par exemple. Ou encore, selon les compagnies, les prix peuvent être exprimés en euros ou en dollars. Bref, il y a une réelle variabilité dans les sources, qu'il faut pouvoir gérer jusqu'à uniformiser l'ensemble dans un entrepôt commun. Dans [Ell13] les auteurs recommandent d'ailleurs de considérer ses sources comme externes à l'entrepôt de données, car leur contenu et leurs formats ne sont généralement pas ou peu contrôler par le(s) responsable(s) de l'entrepôts et/ou les utilisateurs finaux.

Afin de gérer cette hétérogénéité, les entrepôts de données viennent avec une couche logicielle appelée processus ETL (Extract, Transform and Load). Elle a pour but de lire les données des sources, d'en extraire les informations utiles à l'entrepôt et d'effectuer un ensemble d'opérations de transformations qui peuvent varier selon les sources [Ell13]. Le rôle des transformations peut être de : corriger certaines erreurs (orthographes par exemple) ; résoudre des problèmes de conflits de nommage entre les sources ; filtrer (ou nettoyer) les données ; recouper des données de différentes sources ; convertir les données aux formats de l'entrepôt (euros en dollars, par exemple) ; supprimer les doublons... Une fois les transformations adéquates effectuées, les données sont chargées dans l'entrepôt. Dans [Ell13], le processus ETL est présenté comme une étape critique, car la qualité des données dont pourra se servir l'utilisateur en dépend. L'auteur l'explique à l'aide d'une comparaison de l'architecture d'un entrepôt de données avec un restaurant. Le processus ETL est comparé à l'arrière-cuisine et ils insistent sur l'importance de la qualité, l'intégrité et la cohérence des produits (les données) qui y transitent et y sont transformés, car ceux-ci seront ensuite servis aux clients.

Une fois les données dans l'entrepôt, elles doivent être mises à disposition de l'utilisateur, pour que celui puissent y accéder par requêtes [Ell13]. Un ou plusieurs magasins de données (datamart) seront donc crée. Dans [Ell13] les magasins de données sont comparés à la salle de restaurant, qui est pensée et conçue pour le confort du client et le séparer du processus qui se déroule en cuisine, car il serait dangereux (risque de perte de l'intégrité des données et la cohérence des données) de laisser le client y accéder librement.

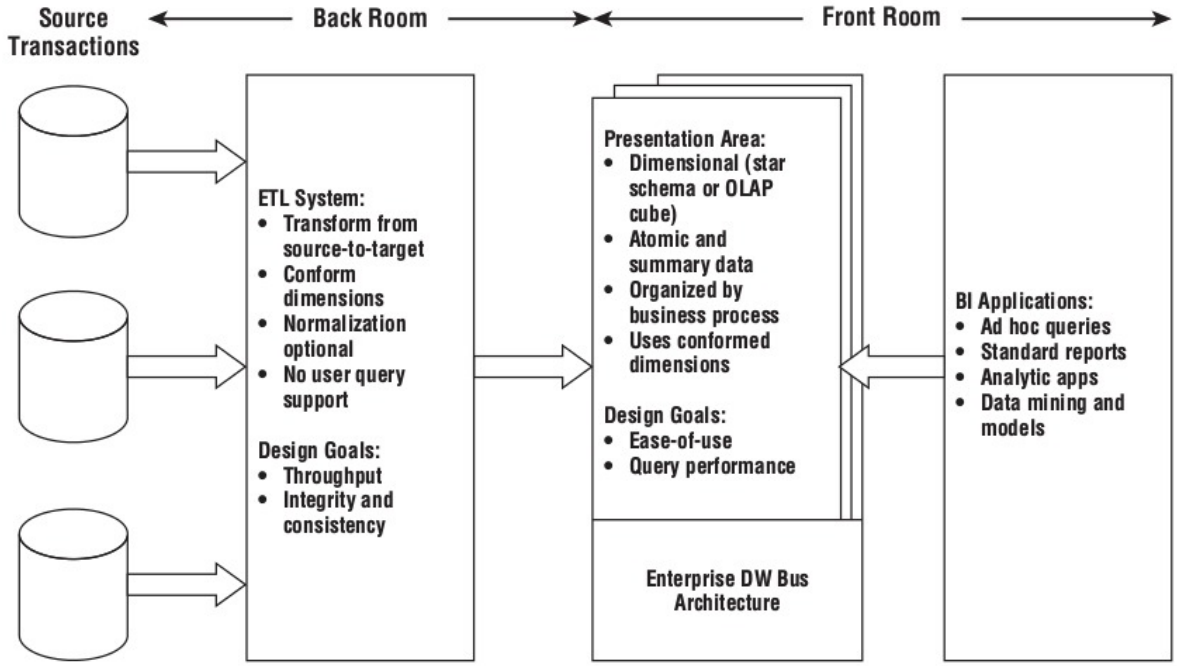


FIGURE 2.3 – Architecture d'un entrepôt de données, source :[Ell13]

Une fois les données mises à disposition, il faut donner aux clients les moyens d'obtenir les données qu'il souhaite (effectuer des requêtes) et de les analyser, de les visualiser [Ada06] sous différents supports (tableaux, graphiques, histogrammes...). Il faut donc fournir aux clients un ensemble d'outils [Ell13] qui lui permettront d'exploiter les données dans un but précis. C'est le rôle des systèmes d'analyse et de traitement en-ligne (OLAP : OnLine Analytical Processing), qui fournissent les outils nécessaires à l'exploitation des données.

La Figure 2.3 résume la structure générale d'un entrepôt décrites ci-avant. Sur la figure, nous retrouvons les différents éléments identifiés lors de la comparaison avec un restaurant, faite dans [Ell13]. La description qui suit, détaille les différents éléments visibles sur la figure de la gauche vers la droite).

1. D'abord les fournisseurs de matière première, qui peuvent être plusieurs, c'est-à-dire, les différentes sources de données, qui peuvent être multiples.
2. Ensuite, la matière première arrive dans l'arrière-cuisine, cuisine, où « la magie » opère, afin de créer un ensemble « comestible » pour les clients. C'est le processus ETL, qui permet d'extraire des sources les données qui ont de l'intérêt, et de les transformer pour les adapter au format de l'entrepôt, afin de faciliter l'exploitation avenir des données par les clients.
3. Puis, une salle de réception, permettant au client de profiter de ce que le restaurant peut offrir, dans un lieu sécurisé, plutôt que de devoir aller en cuisine et de manipuler les couteaux, les hachoirs et les appareils de cuisson à haute températures. La salle de réception permet aussi d'offrir au client un décor (une vue sur les données), correspondant à son besoin. En effet, quelqu'un cherchant à trouver un vol, sera intéressé par : les

prix ; les horaires ; les temps de vol ; les temps de trajets ; les correspondances... Tandis que quelqu'un qui chercherait à acheter une compagnie aérienne, sera plus intéressé par : la couverture (l'ensemble des destinations vers lesquelles la compagnie proposent des vols) ; la taille et la composition de la flotte ; l'état des appareils ; leur nombres de vols ; l'historique des chiffres d'affaires annuels ; la masse salariale... Ainsi, pour chacun de ses usages, les données seront organisées de façon à offrir au client un accès rapide et lisible aux informations qui l'intéressent.

4. Une carte du restaurant, permettant au client de savoir comment demander ce qu'il souhaite aux serveurs, qui prendra la forme d'une application dédiée au client et qui implémente les outils OLAP.

Ce concept d'entrepôt de données a rapidement été adopté dans divers pan de l'industrie [Bou16] : production, vente au détail, services financiers, transports, télécommunications, médecine....

2.2.2 La modélisation dimensionnelle

La modélisation dimensionnelle a été popularisé par Ralph Kimball en 1990 [Bou16, Ada06]. Il s'agit d'une méthode de modélisation des données, pensée pour optimiser l'exploitation des entrepôts de données. En effet, les entrepôts étant destinés à traiter des flux de données extrêmement volumineux [BFG+06, Ell13], cela induit une difficulté à naviguer dans l'ensemble des données. Un des désavantages de cette modélisation est le surcoût en espace mémoire, mais avec l'évolution des technologies de stockage, capable de stocker de plus en plus d'information, dans un espace de plus en plus petit et de moins en moins cher, ce n'est plus vraiment un problème [BFG+06]. Son grand avantage en revanche est sa facilité de compréhension, permettant aux concepteurs professionnels, de discuter avec leur client dans le même langage. Mais cette technique permet aussi une grande précision technique, requise pour les concepteurs, les développeurs et les gestionnaires de bases de données.

Le modèle multidimensionnel comprend les éléments suivants :

- Les Faits : les faits représentent ce que l'on cherche à analyser [BFG+06, Ell13]. Dans une entreprise, il peut s'agir des ventes, de la production, des commandes clients... Les faits peuvent être associés à un ensemble de mesures (attributs) [Bou16]. Un exemple d'attribut pour une vente est le montant total de la vente.

- Les Dimensions : les dimensions sont un ensemble d'éléments, qui précise le contexte d'un fait [BFG+06]. Dans le cas de l'analyse des ventes d'une entreprise, les différentes dimensions pourront être : (i) la date de la vente ; (i) le lieu de la vente (pour une entreprise possédant plusieurs enseignes) ; (i) le vendeur ; (i) le client... Les dimensions servent donc à préciser tout éléments de contexte jugé utile à l'analyse d'un fait. Les dimensions sont des listes d'éléments, qui peuvent s'organiser en hiérarchies [BFG+06]. Dans ce contexte, une hiérarchie est un découpage d'une dimension en différents niveau de granularité, généralement de la moins fine à la plus fine. Par exemple, la date d'enregistrement d'une vente peut-être découpée de la manière suivante : années > semestre > trimestre > mois

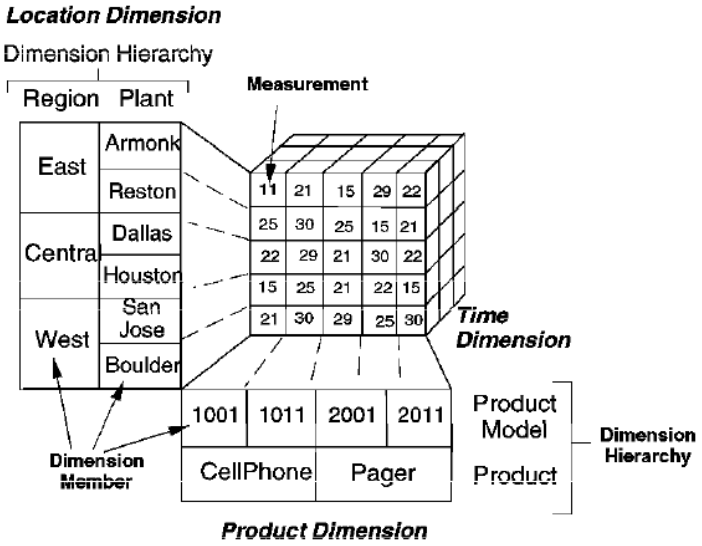


FIGURE 2.4 – Exemple de cube OLAP [BHS+98]

> semaine > jour. Un autre exemple, l'enregistrement d'une adresse peut se faire selon la hiérarchie suivante : pays > région > ville > rue.

2.2.3 Différentes représentations du modèle multidimensionnel

Le cube OLAP Pour un schéma contenant une table de fait et trois dimensions, il est possible de visualiser les données en trois dimensions spatiale (une par dimension du schéma), sous la forme d'un cube. En général, il y a plus de trois dimensions, les données sont alors difficilement visualisables dans l'espace, mais conceptuellement, il suffit de rajouter des dimensions au cube. Ce dernier devient donc un hypercube, puisqu'il possède plus de trois dimensions [BHS+98]. La figure 2.4 est un exemple de cube. Le fait analysé est le volume de production. Ce volume peut être découpé selon plusieurs point de vue : le volume de production sur une période donnée ; le volume de production par produit fabriqué ; le volume de production par usine, repérée par la région et le nom de l'usine. Nous avons donc trois dimensions : temps, produit, adresse. La dimension produit est donc un tableau, dont les indices sont les différents produits, chaque case du tableau peut être subdivisée en plusieurs cases, qui sont ici indexées par les différents modèles du produit. Ainsi, chaque dimension peut être visualisée comme un tableau, dont chaque case peut être divisées en plusieurs cases. Les cases finales (qui ne sont pas divisées), contiennent le volume de production (la mesure de ce volume). Les dimensions du cube sont donc des tableaux à plusieurs entrées et les bases de données basées sur cette visualisation du modèle multidimensionnel sont dites MOLAP (Multidimensional OLAP) [Vas98].

Le schéma en étoiles En 1998, Panos Vassiliadis a fourni les outils théoriques permettant de transformer un cube OLAP en tables relationnelles [Vas98], qui elles-mêmes possèdent une

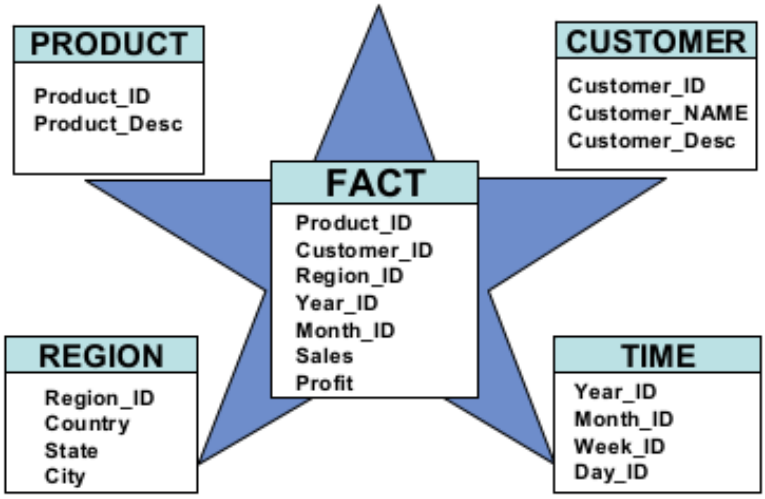


FIGURE 2.5 – Schéma en étoile [BFG+06]

traduction dans le modèle entité-association. Il a ainsi montré que les entrepôts de données étaient implémentables sous forme de bases de données relationnelles, même si des extensions sont nécessaires, notamment pour implémenter le processus ETL.

La figure 2.5 est un exemple de schéma en étoile. La table de faits est mise au centre du schéma et les différentes dimensions sont réparties autour d'elle. Chaque ligne de la table de faits, correspond à une occurrence du fait [Ada06, BFG+06, Ell13]. La table de faits est reliée aux tables de dimensions (une table par dimension), sur la figure 2.5, les cinq premiers attributs de la table sont des références vers des clés contenus dans les autres tables. Le schéma en étoile est considéré comme une méthode optimale d'organisation d'un modèle dimensionnel [Ada06]. Les bases de données basés sur cette visualisation relationnelle du modèle multidimensionnelle sont dites ROLAP (Relational OLAP) [Vas98].

Le schema en flocon Le schéma en flocons est une modification du schéma en étoiles. Si la dimension est hiérarchisable (voir 2.2.1), alors sa table de dimension peut être éclatée en plusieurs tables reliées les unes autres [BFG+06]. Chaque table possède un lien avec la table de granularité directement inférieure (par exemple, la table année possédera donc un lien vers la table mois et la table mois avec la table jour, pour une hiérarchie : année > mois > jour). La figure 2.6 donne un exemple d'un schéma en flocon, avec les dimensions éclatées en plusieurs tables. Deux dimensions différentes ne doivent pas partager de table en commun.

Dans [Ada06] Christopher Adamson explique que ce schéma provoque une multiplication des tables qui peut mettre en péril les performances des requêtes et du processus ETL. Ils recommandent d'éviter le schéma en flocons, à moins d'avoir de bonnes raisons de le faire.

Le schéma en constellation Le schéma en étoile (ou en flocon) possède toujours une seule table de faits. Cependant, il est possible d'avoir plusieurs tables de faits dans un même

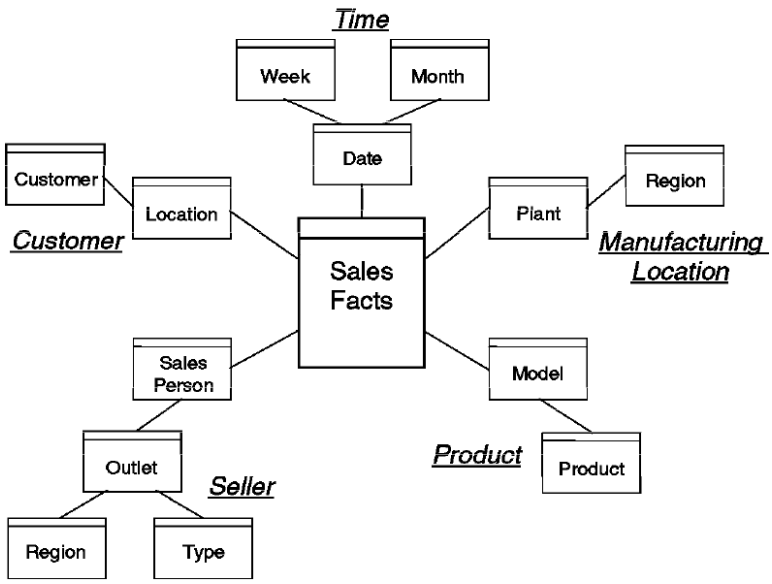


FIGURE 2.6 – Exemple de schéma en flocon [BHS+98]

schéma, il s'agit alors d'un schéma en étoiles multiples ("Multi-star model" [BFG+06]), dit aussi schéma en constellation [Bou16]. Dans ces schémas, deux tables de faits ne peuvent être reliées entre elles, mais elles peuvent partager une ou plusieurs dimensions.

2.3 Remise en question du standard relationnel

En 2008, un rapport d'un rassemblement de nombreux membres de la communauté bases de données à Claremenont [AAB+08] explique que le domaine de recherche des bases de données est en train de vivre un tournant historique. En effet, notamment à grâce à internet, de plus en plus de données circule en réseau et les usages nécessitant l'utilisation de données se diversifient. Ce n'est plus seulement les entreprises, qui sont friandes de données, mais aussi les chercheurs en sciences dures, comme en sciences humaines et sociales, s'y mettent aussi, ainsi que les particuliers (débuts des blogs et réseaux sociaux, par exemple). Sans compter [Bru94], les bases de données médicales, les agences nationales de défenses du territoire, les organismes qui calculent les indices économiques et sociaux... S'ajoute à cela, l'explosion des smartphones et en particuliers, des applications sur smartphones, qui participent elles aussi à l'augmentation du volume de données qui circulent sur internet [SBPR11]. De plus, le cloud fait ses débuts, donc le trafic de donnée n'a pas fini de croître. Mais l'augmentation du volume de données n'explique pas à elle seule, cette révolution, la nature des données à traiter se diversifient également. Pour citer quelques chiffres, en 2017 : (i) Facebook [Noy18, Coë17a] : chaque jour, entre 300 et 350 millions de photos ajoutées, 10 milliards de messages échangés, chaque minute, 510 mille commentaires, 293 milles mise à jour de statuts... 350 gigaoctets de données échangées ; (ii) Twitter [Coë17b] : 500 millions de tweets par jour.

En effet, traditionnellement, les données proviennent de bases de données relationnelles et

son conforme à un modèle de données (généralement relationnel) bien défini. Ces données sont dites structurées [PHM, KPG+]. Cette structure facilite grandement leur exploitation. Seulement, les données échangées sur internet peuvent être sous des formats tel que l'XML [w3c16] et/ou JSON [Int, jso], qui sont des formats standard d'échanges de données. Il s'agit de format dont la description (le schéma) des données est compris dans les données, de sorte qu'il ne soit pas nécessaire de le connaître à l'avance. Ce type de données sont dites semi-structurées [Bun97, PHM]. Aussi, les données peuvent également être incluses dans des fichiers/documents en langages naturelle (pdf, emails, blogs, tweets, documents divers...) [AAB+08, KPG+], elles ne sont donc pas accompagnées d'un schéma les décrivant, car elles n'ont aucune structure particulière, et sont alors dites non-structurées [CP14]. Les données non-structurées peuvent être au format textuel, comme dans le cas précédent, mais pas seulement, les images, par exemple, sont aussi des données non-structurées [PHM].

Ce nouveau contexte, dans lequel les données peuvent être déstructurées et arriver en grande quantité possède un nom devenu très populaire aujourd'hui « Big Data ». Ce terme a été introduit et défini par Doug Laney, en 2001. Il a défini les trois propriétés principales, appelées communément les 3V, qui définissent la tendance Big Data [KPG+, LAP14] :

- Un V pour Volume : les quantités de données à stocker se compte en téraoctets ou plus ;
- Un V pour Velocity (Vitesse) : les données arrivent en grandes quantités et doivent être gérées rapidement, or les bases de données relationnelles sont limitées par les temps d'accès aux disques durs, malgré les hautes performances des processeurs ;
- Un V pour Variety (Variété) : les sources de données sont hétérogènes, mais ne sont plus systématiquement structurées ;
- Aux 3V originaux, s'est ajouté un quatrième V pour Variabilité. Les sources et leurs formats peuvent évoluer au cours du temps et obligeant ainsi les systèmes de gestion de bases de données à se mettre perpétuellement à jour [LAP14] ;
- Et un cinquième V est également considéré par certains chercheurs, pour Véracité, car la véracité des données est un critère essentiel à leur analyse.

Il suffit qu'au moins une de ses trois propriétés (originelles) soient vérifiées, pour parler de Big Data et les bases de données relationnelles sont difficiles à déployer dans ce contexte, ce qui laissera le champ libre à des solutions nouvelles [Bou16]. Cependant, les bases de données relationnelles sont déjà très répandues, et les grandes entreprises en avoir des dizaines, ou des centaines. Les entrepôts de données ont été une première solution aux « Big Data », mais ils sont construits sur les RSGBD traditionnelles [SBPR11] et en souffre donc aussi de leurs limites. Dans [LAP14], les auteurs expliquent que les entrepôts de données sont trop lents, autrement dit, il gère très la vitesse des données. Ils donnent en exemple la détection de fraude, dans une grande entreprise. Les données à analyser pour détecter une fraude peuvent être extrêmement volumineuses et complexes (fichiers logs, historique, généralement sous formes de fichiers textes, des activités de l'entreprise, entretenu en temps réel), pour autant, si une fraude est en train de se produire, celle-ci doit être détectée très rapidement et en temps réel, car les fraudes peuvent faire beaucoup de dégâts en un laps de temps très courts. Dans [WQS16], les auteurs prennent Twitter comme sujet d'étude. En effet, la plate-forme de microblogging qu'est Twitter, permet à des individus de poster des contenus malhonnête et/ou malveillant, appelés communément spams. Le débit de nouveaux tweets rend nécessaire l'automatisation

de la détection des spams, mais pour être efficace (limiter au maximum la visibilité des spams avant leur suppression), les algorithmes de détection doivent traiter les données en temps-réel, malgré le débit élevé de données.

Rakesh Kumar [KGCJ14] résume les contraintes imposées par les Big Data : (i) La scalabilité : les gros volumes obligent les concepteurs à concevoir des systèmes qui une fois mis en application, devront être capable d'assumer les volumes de données qui pourront dépasser le pétaoctet ; (ii) Le temps de réponse rapides : malgré la quantité de données, les certaines applications requièrent des temps de réponses de l'ordre de la centaine de millisecondes, et parfois moins ; (iii) Une maintenance minimale : pour des raisons de coût, le système doit pouvoir fonctionner sans induire de taches de maintenance trop lourdes ou trop nombreuses ; (iv) Des systèmes facilement débuguable : en cas de bug, un système de log doit permettre d'analyser l'activité du système, afin de comprendre l'origine du bug ; (v) Une bonne extensibilité : les applications pouvant être diverses et variables dans le temps, il faut pouvoir ajouter de nouvelles fonctionnalités, pour un coût de développement minimal ; (vi) Une grande robustesse : face à l'utilisation massive des bases de données, les erreurs humaines (déploiement d'un code incorrecte), informatique (duplication de données) ou les problèmes matériels (une machine qui tombe en panne) ne doivent pas empêcher le système de continuer à fonctionner. Ainsi, de nombreuses nouvelles solutions vont être proposées, pour répondre aux challenges des Big Data. Certaines vont essayer d'enrichir les BDR traditionnelles, afin de l'acclimater au nouvel environnement et à ses contraintes, d'autres vont faire fi du standard relationnel et des BD classiques, afin de construire depuis zéro, des solutions « maisons ». Les premières sont dites NewSQL [GHTC13, PA16, KPG$^+$], les secondes sont aujourd'hui appelées NoSQL (« pas de SQL »).

2.3.1 Les bases de données NoSQL

Selon [SF13], le terme « NoSQL » remonte à juin 2009. Johan Oskarsson, originaire de Londres, était à San Francisco, pour une courte période et voulait profiter de l'occasion pour en connaître plus sur certaines technologies de bases de données nouvelles. N'ayant pas le temps de visiter tout San Francisco, à la rencontre des créateurs de ces technologies, il a décidé de créer un meeting, pour qu'ils viennent tous à lui. Lui permettant ainsi d'en voir un maximum d'entre eux, en un minimum de temps. Pour assurer le succès du meeting, il voulait lui donner un nom, qui puisse servir de hashtag puissant, sur Twitter. Un certain Eric Evans lui a proposé « NoSQL », ce nom avait l'avantage d'être court, unique et facile à retenir. Ce terme n'avait pas pour but de devenir le terme consacré d'une tendance nouvelle, sa seule ambition était d'assurer la visibilité, et donc le succès, du meeting organisé par Oskarsson. Mais il s'imposera malgré lui dans la communauté, victime, en quelque sorte, de son succès.

Les technologies NoSQL sont très diverses, mais possèdent en général, un certains nombres de caractéristiques communes [Bru94, GHTC13, KPG$^+$] : ce n'est plus à la base de données de gérer le schéma des données (lorsqu'il y en as un), mais à l'application cliente ; il n'y a plus de mécanisme permettant de spécifier des relations entre des éléments d'une collection de données avec une autre, autrement dit, les données ne sont plus relationnelles. Dans [LAP14],

les auteurs expliquent que le mot est en effet trompeur, car l'expression « NoSQL » ne désigne pas des bases de données sans le langage SQL, mais bien des bases de données « non-relationnelles ». L'expression est donc, en quelque sorte « impropre » ; une capacité à passer à l'échelle par répartition du volume de données sur plusieurs machines ; la disponibilité des données, elles doivent être toujours accessibles, lorsqu'un utilisateur souhaite y accéder (cela peut impacter la cohérence, au sens ou, si lorsque la donnée circule entre plusieurs point d'un réseau, elle peut être altérée, perdue ou modifiée sans que la modification soit répercutée sur un éventuel doublon de cette dernière, ou sur l'écran de l'utilisateur qui en demandé la visualisation) ; [Bru94] insiste sur le faîte que malgré l'expression NoSQL, communément employé, certaines implémentations proposent un langage de requête, parfois proche du langage SQL. Cassandra [Fou16], par exemple, propose le language CQL (Cassandra Query Language). Ce qui rejoint le point de vue de [LAP14], sur la maladresse de l'expression « NoSQL ».

Le mécanisme de transactions et la « charte » ACID (voir section 2.1) sont généralement perdues, mais certaines implémentations obéissent parfois à un autre ensemble de règles, nommé BASE [KPG$^+$]. Il s'agit d'un acronyme pour [GHTC13] : « Basically Available » (disponible en permanence) ; « Soft state » (les données peuvent être incohérentes, sur une courte période de temps) ; « Eventually consistent » (si une incohérence est introduite, celle-ci finira par être résolue).

La tendance NoSQL remporte énormément de succès, bien qu'elles soient encore considérées comme assez peu matures. Elles sont donc encore utilisées avec prudence, mais d'après Zane Bicevska [BO17], ce n'est qu'une question de temps, avant qu'elles ne s'imposent réellement. Les bases de données NoSQL peuvent être réparties en quatre types principaux [Bru94, Mat17, KPG$^+$, GHTC13, AVM15, Bou16].

Pairs clé-valeur Il s'agit de type considéré comme le plus élémentaire, les données sont stockées, dans une table de hachage, sous forme de couple (clé, valeur) dont la clé est unique et est le seul point d'accès vers les données. Ces dernières sont dites opaques, ce qui veut dire que contrairement à un tableau relationnel, qui permettait d'accéder aux données en fonction de leur valeur, ici, les valeurs ne sont pas directement lisibles. De plus, les valeurs ne sont pas contraintes à être de même type, d'un élément à un autre, la valeur peut être un entier, une chaîne de caractère, ou un type plus complexe quelconque. Les données ne respectent aucun schéma particuliers. Le système est très pratique dans un environnement distribué (réseau de plusieurs machines), mais n'est pas adapté pour représenter des données structurées ou liées entre elles.

Orientée colonne Les bases de données orientées colonnes ressemblent un peu aux bases de données relationnelles, avec les colonnes jouant le rôle des lignes, et vice-versa. Cependant, il n'y a pas de schéma défini. En effet, les colonnes ne sont pas contraintes à un type particulier et peuvent être sub-divisée en sous-colonnes, le nombre de sous-colonnes n'étant pas non toujours le même. De plus, le découpage des colonnes peut évoluer dans le temps, au fil des besoins. La sub-division de colonne est pratique pour stocker des données liées entre elles,

mais la nature des relations n'est pas représentée. En revanche, les valeurs contenues dans la base de données ne sont pas opaques, contrairement au cas précédent, ce qui permet d'avoir un système de requête plus puissant. Certains systèmes ont même un langage de requête. Cassandra [Fou16], par exemple, utilise le langage CQL (Cassandra Query Language), basé sur le langage SQL.

Orientée documents Les bases de données orientée document sont une extension des bases de données clé-valeur, dans lesquelles les valeurs peuvent être toutes sorte de document. Les documents sont des fichiers textes de données semi-structurées, dont le format n'est pas imposé (XML, JSON...). Aussi, les documents sont lisibles et il est possible de rechercher et d'extraire directement une valeur d'un document. MongoDB [Mon18] utilise un langage de requête basé sur le format JSON.

Orientée graphes Cette fois les données sont représentées sous formes de graphes, permettant de stocker de données possédant des liens complexes. Dans les bases de données relationnelles classique, les graphes possèdent généralement un seul type d'arcs (un seul type de relations). Mais, les graphes issus des réseaux sociaux, par exemple, contiennent des millions d'utilisateurs. Les utilisateurs peuvent être amis, de la même famille, abonné à la page d'un autre utilisateur, membres d'un même groupe... Une même entité, peut avoir plusieurs types de relations. Ainsi, les bases de données orientées graphes représentent les données sous forme d'entités, qui sont les nœuds du graphe et de relations, qui sont les arcs du graphe et autorisent plusieurs types d'arcs et autorisent également chaque entité, à être associées à plusieurs arcs, qui peuvent être de types différents. Il est possible de faire des requêtes basées sur les données d'un nœud, de rechercher des chemins entre plusieurs entités dans le graphe et de le parcourir selon diverses heuristiques.

De nombreuses technologies NoSQL existent [Mat17], Apache Hadoop a développé un certain nombre de technologies qui servent de support à de nombreuses autres [Fou14]. Ces technologies sont pensées pour aider à créer des bases de données et programmes capable traiter de très large quantité de données, qui peuvent aller jusqu'à l'exaoctet [PHM] et dont les ressources en stockage et/ou puissance de calcul peuvent être répartis (distribué) sur plusieurs machines. Nous présentons ici quelques exemples des technologies proposées par Apache Hadoop.

- Apache HDFS (Hadoop Distributed File System) [Fou14] : il s'agit d'un système de gestion de fichiers pensées pour un environnement distribué. Les données sont dupliquées et distribuées sur un réseau de machines et est pensé pour fonctionner aussi bien avec une machine, qu'avec un réseau de plusieurs milliers de machines [KPG+].
- Apache HBase [Fou18] : c'est une base de données distribuées, orientée colonne, prévue pour stocker de très larges tables de données (des matrices contenant des milliards de lignes et des millions de colonnes). La base de données fonctionne sur la base du gestionnaire de fichiers HDFS.
- Apache MapReduce [Fou14, PHM] : un framework permettant d'écrire et de déployer des programmes pensés pour fonctionner dans un environnement distribué. Un environnement distribué permet entre autre de faire du calcul parallèle (plusieurs machines réalisent des

sous-tâches d'un même programme général), mais un programme n'ayant pas été prévu pour gérer les calculs en parallèle peut mal, voir ne pas fonctionner, dans un tel environnement. Le framework gère automatiquement la parallélisation et permet d'abstraire les concepts de programmation parallèle pour les développeurs. Il gère également les problèmes de pannes (une ou plusieurs machines du réseau, qui cesse de fonctionner). Toutes ces caractéristiques ont rendu ce framework très populaire [KPG+].

Les technologies développées par dans le cadre d'Hadoop, sont assez populaire dans le monde des BD non-relationnelles [Mat17]. Rudi Bruchez [Bru94, p. 129] a dit du projet Hadoop : « on peut dire qu'il est à l'origine d'une petite révolution informatique ». Voici quelques exemples d'utilisation : (i) [TYS+14] fait usage de HBase, pour stocker des relevées de températures, humidités et d'autres paramètres, dans une pièce et s'appuie ensuite sur MapReduce, pour construire l'algorithme d'analyse. Le but étant de développer un système capable d'analyse la qualité de l'air ambiant dans un environnement clos ; (ii) [HGZ+16] utilise également HBase et MapReduce dans leur projet energy-CRADDLE, qui est un entrepôt de données. HBase sert de stockage, à la place d'une BD relationnelle classique, tandis que MapReduce est utilisé pour le développement des outils d'analyse, à l'instar des outils OLAP ; (iii) [KK13] a développé HadoopTS (voir section 3.2.3) ; (iv) [CN16, CPMC16, ATB+, PTd16, AGR15] utilisent tous MapReduce, pour développer des algorithmes de manipulation et d'analyse de données.

2.3.2 Les bases de données NewSQL

Le terme est introduit en 2011, par Matthew Aslett [KGM+], un analyste de 451 Group, dans un article issu de l'entreprise [PA16]. Les technologies dites NewSQL, visent à conserver les transactions ACID, ainsi que le langage SQL, afin d'en garder les avantages. Le besoin vient entre autres des grandes entreprises, qui possèdent parfois des centaines ou plus encore de bases de données relationnelles et souhaitent donc en conserver les avantages, notamment les propriétés des transactions ACID, tout traitant des volumes de données typique du « Big Data » [SBPR11] et/ou en s'adaptant à un environnement distribué [KPG+]. C'est, entre autre, sur la cohérence des données que certaines entreprises ne peuvent se permettre de lâcher du leste [KGM+]. Diverses innovations seront donc apportées, pour permettre aux SGBDR de s'acclimater à un environnement distribué [KPG+, SBPR11]. Ainsi, contrairement aux technologies NoSQL, les technologies NewSQL ne sont pas prévus pour autre chose que des données structurées (le schéma des données doit être connu et figé) [GHTC13]. De sorte que toutes les manipulations disponibles dans le langage SQL, puissent s'appliquer. Cependant, certaines technologies n'utilisent pas forcément un SGBDR comme cœur applicatif, même la représentation des données est parfois différente du modèle relationnel [GHTC13], mais s'arrange tout de même pour que tout fonctionne comme en relationnel, du point de vue de l'utilisateur final. Les technologies NewSQL sont généralement classées en trois catégories [PA16, KC14].

- Les solutions reconstruites de zéro : lorsque les SGBDR ont été développés dans les années 70, la mémoire informatique était chère et encombrante. Ainsi, l'exploitation d'une base de données de quelques gigaoctets, nécessitait de nombreuses lectures du disque, car la mémoire ne pouvait tout contenir. Seul une portion (appelée page) de 8 ko pouvait être chargées

en mémoire vive, afin de travailler dessus. La manipulation d'une donnée qui n'était pas dans la page, nécessitait de réécrire le contenu de la page en cours sur le disque et de charger une nouvelle page, contenant la donnée nécessaire aux manipulations demandées. Cette opération, s'appelle un swap. Cependant, à l'heure actuelle, 4 GB peuvent tenir dans une clé USB, et les mémoires vives de 4 GB, voir 8 GB, sont devenues banales. Ainsi, certaines technologies NewSQL sont reprises à zéro, de façon à tirer profit des technologies actuelles, comme la mémoire vive plus conséquente, ou l'utilisation de mémoire dont les temps d'accès sont plus courts qu'un disque dur classique, tel que les disques SSD ou de dispositif de mémoire très peu encombrant et donc disponible en plus grande quantité, tel que les mémoires flash. Cependant, [PA16] précisera, que les performances gagnées ne sont pas toujours à la hauteur du coup de développement, car les SGBDR ont subit de nombreuses évolutions et optimisations, qui compensent les contraintes auxquelles ils ont fait face.

- Les solutions par remplacement partiel : les SGBDR ont un module interne, appelé moteur SQL, chargé de gérer les opérations SQL sur la base de données. Dans un même SGBDR, il est possible d'avoir plusieurs moteurs différents. La plupart des moteurs classiques des SGBDR ont eux aussi été développés dans les années 70 et 80, avant le Big Data et ses nouveaux enjeux. Ainsi, certaines technologies NewSQL, consistent à proposer de nouveaux moteurs, qui s'intègrent aux SGBDR classiques, mais conçu pour faire face aux contraintes imposées par les Big Data.
- Les architectures distribuées (par partitionnement ou par sharding) : le partitionnement, consiste a répartir les données en sous blocs, selon un critère commun. Par exemple, un établissement tel qu'une université, maintenant une base de données de ses étudiants, pourra créer des blocs d'étudiant, selon leur année d'étude et selon les différentes facs. Ainsi, si je cherche la liste des étudiants inscrit à un cours particulier, je connais le niveau d'enseignement du cours (je sais que c'est un cours de L1, par exemple) et je sais aussi que c'est un cours de la fac de droit. Je vais donc chercher uniquement dans le bloc des élèves de L1 de la fac de droit, plutôt que dans toutes la liste des étudiants. Cela permet de gagner énormément de temps.

  Le partitionnement consiste à créer des blocs de données, sur un même disque (partitionnement logique), ou sur plusieurs disques d'une même machine (partitionnement physique). Le sharding va plus loin, en répartissant la base de données sur plusieurs machines mises en réseaux. Chaque machine contiennent le même SGBD, mais avec des portions différentes de la base de données. Un logiciel qui réalise ce sharding, fera en sorte que du point de l'utilisateur final, cette répartition soit invisible (transparente). Quelle que soit la machine, l'utilisateur aura accès à toutes les données de toutes les machines, comme si elles étaient toutes stockées sur une seule et même machine.

  Le sharding permet de proposer du stockage via le cloud. Les hôtes peuvent avoir leur propre réseau de serveur et utiliser le sharding, pour offrir à l'utilisateur un stockage d'apparence unique (comme si toutes les données étaient sur le même serveur). Et ClearDB [PA16] en particuliers ne fournit pas de service cloud, mais fourni un logiciel qui permet à un utilisateur de « fusionner » plusieurs services cloud dont il dispose, en un seul (en apparence, évidemment).

Cependant, le partitionnement entre autre, n'est pas un concept propre aux NewSQL, il exis-

tait déjà dans les bases de données traditionnelles. D'ailleurs, Andrew Pavlo [PA16] insiste sur le faîte que de nombreuses solutions NewSQL ne proposent pas de réelles avancées technologiques, mais ont tout de même le mérite d'avoir su améliorer des technologies connues, en leur apportant des propriétés qu'elles n'avaient pas avant. Par exemple, certaines implémentations permettent de migrer les données d'une partie, d'une machine à une autre, pour soulager une machine surexploitée, par exemple.

Les technologies NewSQL possèdent, en général, un certains nombres de caractéristiques [KGM+, KPG+, KGCJ14, KC14, PA16].

- Elles conservent les fonctionnalités du langage SQL et permettent à l'utilisateur d'interagir avec les données avec le langage SQL, comme dans un SGBDR traditionnel.
- Elles conservent le mécanisme de transactions et leurs propriétés ACID (Atomicity, Consistency, Isolation, Durability, voir section 2.1).
- Elles s'adaptent à une architecture distribuée, avec calcul parallèle. Elles supportent aussi les architectures dites « shared nothing » (« zéro partage »), qui consiste à ne pas dupliquer les données sur plusieurs nœuds, pour que chacun d'entre eux reste indépendant. Ces propriétés permettent de fonctionner sur plusieurs nœuds, sans souffrir du problème du goulot d'étranglement (ralentissement de tous le traffic, à cause d'un seul nœuds, plus lent que les autres).
- Elles possèdent un mécanisme de gestion des processus concurrent (plusieurs processus essayant d'accéder à la base de données en même temps, typiquement, plusieurs utilisateurs cherchant à accéder à la même base de données), dit non bloquant. Cela signifie notamment que les processus de lecture ne sont pas bloqués par les processus d'écriture.
- Les diverses astuces d'utilisation employées permet à chaque nœud d'une architecture distribué, d'avoir à lui seul, de meilleures performances qu'un système OLTP basé sur un SGBD relationnel. De manière globale, un système NewSQL est quarante à cinquante fois plus rapide, qu'un système OLTP traditionnel [KGM+, KPG+, KGCJ14, KC14].

2.4 Conclusion

Au cours du chapitre, nous avons vu l'évolution des technologies de stockage de données, des bases de données relationnelles aux bases de données NoSql. Nous avons vu qu'à l'origine, les programmeurs devaient gérer eux-mêmes leurs données et utilisaient généralement des fichiers textes. Cette méthode était très peu pratique et déviait l'attention des programmeurs sur des sujets qui ne relevaient pas de leurs compétences (puissent qu'il ne s'agissait plus forcément de problèmes liés au code, à son écriture et son débogage). Par conséquent, à partir des années soixante, diverses technologies sont apparues, qui avaient pour but de séparer les données du code et donc de séparer les problèmes de gestion des données des problèmes de développement du code. Il est apparu plusieurs technologies de bases de données (hiérarchiques, réseau et relationnel) et ce sont les bases de données relationnelles qui, par leurs nombreux avantages sur les autres technologies, ce sont imposées dans les entreprises, jusqu'à devenir un standard. Un standard qui ne sera remis en cause qu'à partir des années 2000, avec la démocratisation

des technologies NoSql et cela sera principalement dû aux « boom » d'internet, dans ces années, qui a amorcé l'arriver d'un déluge de données, qui ne cesse de s'intensifier depuis et face auquel les bases de données relationnelles, venu d'un monde sans internet, ont bien du mal à tenir le choc.

Des bases de données relationnelles a découlé la notion de bases de données décisionnelles (les entrepôts de données). En effet, une fois les technologies de stockage mature, les entreprises ont naturellement voulu valoriser davantage leurs données stocker, c'est-à-dire transformer la donnée en information, qui permettent d'alimenter les choix des décisionnaires. Et cette nouvelle tendance, venu de l'aptitude à stocker les données de façon pratique, à inciter à stocker davantage de données. De plus, l'analyse de données produit des résultats, de l'information, également stockés sous forme de données. Même si leur nature est différente. En effet, si nous prenons l'exemple de l'astronomie, les photos de l'espace prises par les télescopes sont souvent inexploitables à l'œil nu (en bref, on y voit rien), c'est l'analyse par ordinateur, qui permet d'y détecter des choses intéressantes et de « faire parler » les images. Si une nouvelle planète est détectée, il faut alors indiquer sa position sur l'image, sa position physique, lui donner un nom, indiquer si elle est tellurique ou gazeuse, ses propriétés (densité, rayon, présence d'une atmosphère, les principaux éléments chimiques qui la composent...), l'étoile (ou les étoiles) autour de laquelle (desquelles) elle tourne, à quelle distance de ce(s) dernière(s)... Toutes ces informations peuvent faire l'objet d'un stockage dans une base de données.

En sciences, l'information extraite de l'analyse des données, peut être une description, un modèle, mathématique (une équation différentielle, dans notre cas d'étude) du comportement d'un système (par exemple, la trajectoire d'une balle lancée en l'air peut être décrite par une équation, faisant intervenir la masse de la balle et la force avec laquelle elle a été lancée). On parle alors de modèle mathématique et les équations différentielles sont un type particulier de modèle mathématique. Il devient alors intéressant pour les scientifiques de stocker à la fois, des séries chronologiques, mais aussi, des modèles théoriques. Dans le chapitre suivant, nous verrons, que si les technologies de stockage de séries sont aujourd'hui matures, le stockage et la gestion de modèles mathématiques nécessite encore de l'approfondissement.

Chapitre 3

Modèles Mathématiques et Séries Chronologiques dans le Monde des Bases de Données

---

Aujourd'hui, de nombreux domaines font usages de séries chronologiques. Nous pouvons citer la médecine, l'économie et la finance, les sciences dures ou humaines, la surveillance militaire, le recensement des catastrophes naturelles et la gestion des ressources des territoires, les réseaux sociaux, les réseaux téléphoniques, les réseaux électriques... [RCM+13, KK, AAB+08, DSK10, CN16].

Parmi les applications actuelles extrêmement productives en termes de données et surtout de données sous formes de séries chronologiques, il y a également l'Internet des Objets, dit IoT (Internet of Things) en anglais [Ham15, CPMC16, BT11, GHTC13]. Il s'agit d'un nouveau sous réseaux d'internet, composé uniquement d'appareils connectés et capable de communiquer entre eux. Dans [Cha14], il est précisé, que les données issues d'instruments, de capteurs, sont aujourd'hui les plus répandus. Autrement dit, ce ne sont plus les humains qui génère le plus de données, mais les machines qu'ils construisent et qui communiquent entre elles. L'ensemble génère de grandes quantités de données, parfois si volumineuse, qu'il est parfois hors de question de les dupliquer sur les machines des utilisateurs qui souhaitent les analyser [CKF16]. [TYS+14] utilise un réseau de capteurs pour l'analyse et le contrôle de la qualité de l'air ambiant dans un environnement clos (intérieur d'un bâtiment). Ainsi, les séries chronologiques ne sont pas épargnées par le phénomène de déluge de données des années 2000, elles en sont même au cœur, étant donné l'engouement pour les objets connectés de notre « ère d'internet ».

Les sciences ne sont pas en reste, en termes de production de données. En astronomie, les satellites et les télescopes fournissent de plus en plus de données, qui sont mises à disposition des scientifiques, charger de « faire parler les images » [CN16, DSK10, SBPR11]. Dans [CN16], les auteurs précisent même qu'habituellement, les scientifiques chargent les données qui les intéressent sur leurs serveurs locaux, avant de faire leurs analyse, mais que la quantité de données disponible est si élevé, qu'il est aujourd'hui moins coûteux de réaliser les calculs directement sur le serveur source des données et de ne faire transiter que les demandes (requêtes, processus de traitement...) de l'utilisateur et les réponses du serveur. Aussi, il est de plus en plus courant d'avoir des bases de données publiques, permettant à n'importe qui de les récupérer et de réaliser ses propres analyses [KK13]. Par exemple, le site kaggle [Inc18] propose un ensemble de jeux de données d'apprentissage, qui permettent de développer/tes-

ter des algorithmes de machine learning, ou à n'importe quel curieux, d'essayer de créer ses algorithmes, que ce soit dans un but précis, ou simplement pour satisfaire sa curiosité.

Et en sciences aussi, les séries chronologiques ont leur importance. D'après [RCM+13], en termes de données chronologiques, l'avalanche est telle, que les scientifiques, de même que les industrielles ont à leur disponibilité des milliards de séries chronologiques, qui n'attendent qu'à être explorées, car les algorithmes d'analyse sont encore incapables de suivre la cadence. En effet, le débit de production des données est extrêmement rapide et est un réel défi à relever [KK, Cha14]. Dans [Cha14], ce sont les compagnies électriques, qui sont prises en exemple. Il est expliqué que ces dernières avaient l'habitude de relever la consommation des utilisateurs une fois par mois (par utilisateurs). Pour une masse classique d'un million d'utilisateurs, cela fait douze millions de relevés par an. Ces données sont notamment utilisées pour répartir le courant dans le réseau. Aujourd'hui, il est possible de faire des relevés plusieurs fois par jours, ainsi, les douze millions de relevé peuvent être très facilement atteints en une seule journée, là où il fallait un an, auparavant. Et le site quandl [Inc17], par exemple, est une plate-forme spécialisée dans les données de type séries chronologiques, souvent fournies par des entreprises (évolution du cours d'une monnaie, évolution de la production d'un produit particuliers...). Les séries sont visualisables et téléchargeable sur le site (ou au sein d'un programme, via diverses API), aussi bien sous forme de graphiques, que de tableaux. Et il est souvent possible de voir l'évolution de la courbe selon plusieurs niveaux de granularité temporelles (par heure, par jour, par mois, par an).

3.1 La nécessité d'avoir des gestionnaires spécifiques aux séries chronologiques

En 1994, Dreyer et al. [DDS94b] constatent que les banques sont composées de plusieurs départements, qui font grand usage de séries chronologiques. L'augmentation du volume de données disponible amène entraîne donc une difficulté à gérer efficacement les données et à trouver les séries adéquates pour analyser, traiter et/ou gérer un problème spécifique. Aussi, en 1995, Schmidt et al. [SDDM95] ont présenté un papier dans lequel ils expliquaient pourquoi les séries chronologiques nécessitaient un traitement particulier. À l'époque, le concept de base de données temporelles (temporal databases) faisait déjà partie du décor des bases de données et beaucoup de chercheurs pensaient les séries chronologiques comme un cas particulier de données temporelles. Cependant, Schmidt et al. soutiennent, qu'il y a une différence fondamentale entre les données temporelles et les séries chronologiques qu'il est nécessaire de prendre en compte pour traiter correctement les séries chronologiques.

3.1.1 Le problème du modèle de données

Une base de données temporelles, est une base de données qui conservent l'historique des données. Par exemple, supposons qu'une entreprise embauche un employé, que l'on appellera Tom, à une date d_1. L'entreprise donne à Tom le poste p_1, avec un salaire s_1. Puis au bout

d'un certain temps, à la date d_2, Tom est promut à un poste p_2 avec une augmentation de salaire, qui passe de s_1 à s_2. Dans une base de données relationnelles classique, le poste et le salaire de l'employé seront mis à jour, pour passer de p_1 à p_2 et de s_1 à s_2. Mais une fois la mise à jour faîte, il n'y a plus la trace de la promotion. En effet, la date de la promotion n'a pas été enregistré et l'ancien poste, ainsi que le salaire associé ne sont plus renseignés non plus.

Une base de données temporelles est en principe, une base de données relationnelles, sauf que les données sont associées à un intervalle temporel indiquant leur période de validité. Ainsi, quand Tom est embauché, son poste et son salaire sont enregistrés et considérés comme valable à partir de la date d_1. À la date d_2, lorsque Tom est promut, les anciennes données de l'employé sont considérées valides de d_1 à d_2 et les nouveaux postes et salaires p_2 et s_2 sont enregistrés et considérés valides, à partir de la date d_2. Ainsi, les anciennes données ne sont pas effacées, ce qui permet de conserver une trace de l'ancien poste et de l'ancien salaire et également de la date de modification des informations, qui correspond à la date de la promotion. On notera, que dans l'intervalle $[d_0, d_1]$, les valeurs p_1 et s_1 sont constantes. Autrement dit, l'intervalle de validité des valeurs, indique la durée pendant lesquelles les valeurs étaient valables et constantes. Lorsque celles-ci changent, elles sont alors associées à un nouvel intervalle, où elles sont à nouveau valables et constantes.

Or, l'hypothèse des intervalles de valeurs constantes ne tient pas, avec les séries chronologiques. Admettons, par exemple, que je cherche à analyser l'évolution d'une voiture, sur un parcours de 100km. Pour cela, je munis le véhicule d'un GPS, pour pouvoir le géolocaliser. Je note sa position de départ comme étant la position zéro. Puis toutes les minutes, je note la distance parcourue depuis le départ. Donc à la minute zéro (que l'on notera m_0), la distance est nulle. À la fin du parcours, j'obtiens une série chronologique (un tableau de valeurs observées). Supposons qu'à m_1 (donc au bout d'une minute après le départ), la distance relevée est de $0,1$km, ou 100 mètres. Je ne peux raisonnablement pas affirmer que la voiture est restée au départ pendant les premières 59 secondes et s'est téléportée 100 mètres plus loin à la 60ième. La voiture s'est manifestement déplacée dans la minute qui sépare les deux relevés. La distance parcourue n'est pas restée constante pendant toute la durée de l'intervalle. Pour le dire autrement, je suis dans l'incapacité de dire où était la voiture au bout de 10, 20, 40 ou $31,4159265358$ secondes, je sais juste qu'elle n'était probablement plus au départ et qu'elle n'avait probablement pas encore atteint les 100 mètres. Cela illustre le fait que, dans une série chronologique, une valeur (mesure/observation) n'est valable qu'à l'instant auquel elle est associée et il n'est pas du tout exclu que cette valeur est pu varier entre deux mesures. C'est la différence fondamentale que pointe du doigt les auteurs dans [SDDM95]. Ainsi, la gestion des séries chronologiques requiert un traitement particulier.

Ainsi, Schmidt et al. [SDDM95] expliquent que le modèle temporel échoue à décrire une série chronologique, puisqu'il se base sur l'hypothèse d'intervalles de valeurs constantes (dreyer et al. [DDS94b] résume le même propos en ces termes « most temporal database systems use an interval model, whereas a TSMS needs a time point model » (la plupart des bases de données temporelles utilisent un modèle de temps basé sur les intervalles, alors qu'un gestionnaire de données chronologiques a besoin d'un modèle de temps ponctuel)).

3.1.2 Le manque de fonctionnalités spécifiques et nécessaires

D'un point de vue fonctionnel, les séries chronologiques peuvent nécessiter un certains nombres d'opérations statistiques et d'opérations spécifiques aux séries chronologiques, qui ne sont pas toujours disponible dans le langage SQL ou les gestionnaires de données relationnelles, ou bien qu'implémentés, ne sont pas utilisables sur des séries chronologiques [DDS94b, DDS94a]. Ramakrishnan et al. [RDR+98] ajoutent également que le langage SQL échoue à répondre à certaine requête pourtant classique dans le commerce ou les sciences, telles que le calcul de la moyenne mobile (qui fait partie des fonctions statistiques dont parlent [DDS94b, DDS94a]). De plus, Schmidt et al. [SDDM95] expliquent que les bases de données temporelles possèdent un langage de requête spécifique, dont les spécificités rendent impossible, l'écriture de fonctions statistiques. Les bases de données temporelles ne sont donc pas faîtes pour fonctionner avec des tableaux de valeurs, alors que les séries chronologiques sont des tableaux de valeurs et la plupart des fonctions statistiques s'appliquent à des tableaux de valeurs. Aussi, dans une base de données temporelles, les valeurs étant considérées constantes dans un intervalle, il n'est pas utile d'implémenter des fonctions d'interpolation, alors que cette fonctionnalité est nécessaire pour travailler avec des séries chronologiques. Schmidt et al. ajoutent qu'il en va de même pour l'extrapolation de valeur. D'ailleurs, [DDS94b] mentionnait déjà l'utilisation d'outils statistique variés (feuilles Excel, programmes spécifiques...). Ils ajoutent que tous ces outils n'utilisent pas toujours les mêmes formats et avoir recours à un DBMS permet d'aider à gérer les différents formats. Ils mentionnent également l'existence de TSMS (Time Series Management Systems), mais que c'est dernier ne sont pas assez performants et les économistes qui essaient de s'en servir, ne peuvent le faire sans l'intervention d'un spécialiste en base de données.

En 1996, [WSV96] réalise un constat similaire. En termes de fonctionnalités liées au traitement des séries chronologiques, il mentionne, entre autre :

— l'introduction de types de données complexes (les séries chronologiques sont parfois composées de données plus complexes que de simples nombres réels) ;

— la nécessité de réaliser des requêtes spécifiques aux séries chronologiques, en temps raisonnables ;

— le besoin de visualiser les données, à différentes échelles de temps, par simulation, ou animation ;

— la possibilité de combiner plusieurs séries chronologiques.

Aux travers de diverses publications, de nombreuses fonctionnalités essentielles à la gestion de séries chronologiques et indisponibles dans les SGBDR ont été identifiés.

— La réalisation d'opérations spécifiques (filtrage, changement d'échelle temporelle, calcul de la moyenne courante, calcul de la différence entre deux séries...), ces opérations ont notamment pour but de pouvoir préparer les données à un traitement spécifique [DDS94b, DDS94a].

— Les quelques opérations statistiques (minimum, moyenne...) implémentée dans les bases de données relationnelles ne sont pas pensées pour traiter des volumes de données aussi large, que ce que peut contenir une base de séries chronologiques [DDS94a].

— Une modélisation plus complexes des évènements, qui rejoins [WSV96] sur l'introduction de type plus complexes. Prenons l'exemple d'une bibliothèque, qui suit les emprunts et les retours de livre. À chaque nouvelle entrée (nouvelles date) sera associée un ou plusieurs évènements (qui peuvent être un emprunt ou un retour). Un emprunt, par exemple, pourra être décrit par le livre emprunté (titre, auteur, numéro d'identifiant, rayon...) et l'identité de l'emprunteur. On voit que dans cet exemple, les évènements ne sont pas de simples nombres, comme ça peut être le cas avec un capteur connecté. De plus, une série chronologique contient des informations, qui ne dépendent pas du temps : la localisation du capteur ou de la bibliothèque dont on suit l'activité, la date départ de la série... Les données indépendantes du temps sont des données qui concerne la série en général (des métadonnées) et sont généralement regroupées dans un en-tête. Ce type de données peut également être plus complexes qu'une suite de nombres [DDS94b].
— Les séries chronologiques étant généralement volumineuse, il faut un système capable de gérer une grande quantité de données en temps raisonnable [WSV96], car les mécanismes (moteurs de recherches) construits pour les bases de données relationnelles ne sont pas adaptés à la recherche de séquences dans un ensemble de séries chronologiques [DDS94a].
— La visualisation des séries chronologiques est également un point important et l'utilisateur doit pouvoir combiner plusieurs séries sur un même graphe et modifie l'échelle de temps [WSV96].

[DF15] ajoute que les séries chronologiques sont rarement modifiées (mises à jour), là où les bases de données temporelles sont prévues pour sauvegarder les mises à jour des données. Et que les valeurs sont lues/extraites par séquences continuent de valeurs (par blocs de valeurs successives). Le but des gestionnaires de séries chronologiques est de fournir une base de données capable de stocker plusieurs séries chronologiques, en fournissant des outils de requêtes optimisés pour les séries chronologiques.

Ainsi, ce sont donc développé, en parallèle des systèmes de stockage précédent, divers systèmes de gestion de séries chronologiques (SGSC), des équivalents aux SGBD, adaptés aux besoins particuliers des séries chronologiques. L'évolution des systèmes de gestion de séries chronologiques étant des cas particuliers de gestionnaire de bases de données, leur évolution est très lié à ces derniers. En effet, avant l'apparition des SGSC, les séries chronologiques étaient également stockées dans des fichiers textes. Mais pour être efficace, le nombre de fichiers devait être petit, pour une quantité de données (valeurs) par fichiers plutôt élevée. Autrement dit, il valait mieux avoir des fichiers peu nombreux, mais volumineux, plutôt qu'une myriade de petits fichiers [DF15]. Et il manquait la couche logicielle, qui, comme dans les SGBD, permet d'automatiser la gestion des données et dans ce cas particuliers, de gérer également toutes les opérations nécessaires spécifiques aux séries chronologiques. Ce qui impliquait donc l'écrire de programmes supplémentaires [DDS94b].

Dreyer et al. [DDS94b] mettent en avant les gestionnaires de données orientés objets, dont la flexibilité permet l'implémentation de types complexes et le modèle temporelle adéquat peut y être facilement implémenté dans des classes spécifiques. De plus, les méthodes de classes, peuvent servir à implémenter les diverses opérations nécessaires sur les séries chronologiques. Les auteurs pensent qu'il s'agit d'une très bonne base, pour développer des SGSC. Dans [DDS94a], Dreyer et al. Introduisent d'ailleurs un système de gestion de séries chro-

nologiques orienté-objet. Ils fournissent principalement deux classes `TimeSeries` et `Group`, qui contiennent toute deux les fonctionnalités de base pour manipuler, récupérer et transformer des séries chronologiques (le système contient notamment une opération permettant de modifier l'échelle temporelle d'une série chronologique). Le système est également équipé d'un moteur de requêtes spécifique aux séries chronologiques. Ainsi, l'utilisateur n'a plus à se soucier de l'implémentation de ces fonctionnalités et peut se concentrer sur l'adaptation des classes à son cas particulier.

3.1.3 Les limites du langage SQL

L'intérêt des séries chronologiques réside dans la possibilité de rechercher des « pattern » particuliers, ce qui n'est pas utile et donc pas supporté par les gestionnaires de données relationnelles [BRLTZ70]. Les patterns peuvent être plus ou moins complexe, de la recherche d'une simple « période de profit » ou l'on recherche une fenêtre temporelle dans laquelle la valeur en fin de période est strictement plus grande que la valeur en début de période [PP99] à des patterns tel que le double minimum (double-bottom), qui consiste à rechercher une tendance à la hausse, suivi d'une tendance à la baisse, jusqu'à un premier minima local, suivi d'une deuxième hausse, puis d'une seconde baisse, pour atteindre un second minima, suivi encore d'une troisième tendance à la hausse [BRLTZ70]. Le langage SQL n'est pas fait pour considérer des données dites séquentielles (des listes de valeurs dont l'ordre est important), car le langage SQL est fait pour travailler sur des relations, pour lesquelles il n'y a pas de notions d'ordre entre les différents tuples. De fait, les opérations basiques du langage SQL (et de l'algèbre relationnelle) ne permettent pas de décrire des patterns, tel que défini plus haut. De plus, Perng et al. [PP99] ajoutent qu'un langage de requête adéquat pour les séries chronologiques devrait respecter les contraintes suivantes : (i) le résultat d'une requête devrait être un/des segment(s) de(s) série(s) considérées ; (ii) les patterns doivent être définies de manière descriptive, c'est-à-dire que l'utilisateur ne doit pas avoir à fournir un exemple sous forme d'un échantillon avec lequel comparer la(es) série(s). Là aussi, le langage SQL ne satisfait pas ces contraintes. Il est donc nécessaire d'étendue les opérations de base de l'algèbre relationnelles, pour supporter des requêtes portant sur des séquences de données [BRLTZ70, PP99, RDR+98]. Bai et al. [BRLTZ70], en 1970, mentionne le projet Informix, comme le premier DBMS commerciale proposant des fonctionnalités spécifiques aux séries chronologiques et de nombreuses propositions d'extension du langage SQL ont été proposées dans la littérature, depuis les années 70 :

— Le projet Informix [HSK+98] : il s'agit d'un gestionnaire de données commercialisée par Informix Software Inc. Il contient un module nommé Informix TimeSeries DataBlade Module, qui fournit des fonctions permettant de gérer (entre autres) des séries chronologiques. Perng et al. [PP99] font mention du projet et indique notamment que le module fournit une solution de stockage avec un certains nombres de fonctions spécifiques, permettant de créer des requêtes sur les séries chronologiques. Mais il n'a pas de langage de requête propre rendant le système peu évolutif.

— Data Management System (DMS) for Time Series [Hel79] : il propose à la fois une

solution de stockage et de gestion des séries chronologiques, mais permet aussi d'effectuer diverses analyses et transformations sur ces dernières, de les écrire dans un tableau ou de tracer des graphes (aussi avec les données originelles, que les données transformées). Il définit également un certain nombre de commandes (fonctions), telle que :

- `ADJUST` : qui multiple une séquence (ou une sous-séquence) par un réel ;
- `AGGREGATE` : qui permet d'agréger les données d'une séquence, sur une échelle des temps plus larges ;
- `CHANGE` : qui permet d'appliquer à une séquence une fonction spécifique (logarithme, exponentielle...) ;
- `STATISTIC` : qui permet d'effectuer des opérations ou des analyses statistiques sur une séquence (calcul de la moyenne, de la variance, corrélations, test du chi-deux ou de kolmogorov-smirnov...) ;
- et beaucoup d'autres.

Là aussi, le système défini de nombreuses fonctions, généralement pour faciliter l'analyse, mais il ne va pas jusqu'à proposer un langage de requête spécifique.

— Shape Definition Language (SDL) [APWZ95] : il s'agit d'un langage focalisé sur la notion de forme. La description de forme se base sur un alphabet définissant des forme de base, tel que `stable`, pour désigner une portion de valeurs consécutives quasi-constantes, ou texttzero, pour désigner une portion de valeurs parfaitement égales. Pour obtenir des formes complexes, les formes de base sont combinées, à l'aide de divers opérateurs, tel que :

- `concat` : `concat(`F_1`,`F_2`)` permet de définir une forme composée de F_1 immédiatement suivie de F_2. Cette fonction permet d'enchaîner autant de forme que l'on souhaite ;
- `any` : `any(`F_1`,`F_2`)` permet de définir une forme « mulitple », c'est-à-dire que les séquences qui correspondront à cette forme seront celle qui dont la forme correspond à au moins une des formes F_1 ou F_2.

Il est possible d'obtenir davantage de précision dans la description des formes, à l'aide d'opérateurs d'occurrence. Par exemple, l'expression `(exact n F)` contient l'opérateur `exact` et sert à spécifier que la forme F doit apparaître n fois exactement. De manière similaire, les opérateurs `atleast` et `atmost` permettent d'imposer la présence d'une forme, au moins n fois et pas plus de n fois, respectivement. Quelques mécanismes supplémentaires permettent d'augmenter l'expressivité du langage SDL.

Perng et al. [PP99] notent cependant que ce mécanisme de description de forme fonctionne uniquement pour les séries chronologiques d'une seule variable numérique. Ils notent également que le langage permet de décrire uniquement des relations entre éléments adjacents et les agrégations, tel que la moyenne mobile, ne peuvent être exprimées dans ce langage.

— TREPL [MZ97] : permet de définir des ensembles de règles. Ces règles sont nommées et définissent des ensemble d'évènements (ou patterns), qu'il est ensuite possible de combiner à l'aide d'opérations logiques (disjonction, conjonction) et temporelles. Les opérations temporelles permettent de définir comment deux évènements s'enchaînent. Par exemple, prenons deux évènements E1 et E2 :

— E2 arrive après E1, signifie que la séquence qui vérifie E2 doit être après celle qui vérifie E1 ;

— mais E2 arrive immédiatement après E1, impose que le premier élément de la séquence qui vérifie E2 et le dernier élément qui vérifie E1 soient consécutifs ;

— et E2 arrive éventuellement après E1, signifie que si E2 arrive, ça doit être après E1, mais il n'est pas obligatoire que E2 arrive.

Selon Perng et al. [PP99] TREPL permet de décrire des patterns de manière efficace et possède une grande expressivité. Mais les auteurs lui reproche de ne pas avoir un mécanisme d'exécution incrémentale, qui permettrait d'optimiser l'exécution des requêtes.

— SRQL [RDR+98] : les auteurs proposent de considérer les données séquentielles comme des relations ordonnées, à l'aide d'élément du langage SQL qui tirent avantage des séquences de données triées. Sur cette base, ils proposent une algèbre, qui étend l'algèbre relationnelle, ainsi qu'une implémentation de cette-dernière le langage SRQL (Sorted Relational Query Language, prononcé « circle » en anglais).

Les relations sont ordonnées à l'aide d'une clé (qui est un attribut de la relation ou une combinaison d'attributs), sur laquelle il est possible de définir un ordre (réels, entiers...). Cette attribut est nommé *attributs de séquençage* (« sequencing attributes »).

— SQL/LPP [PP99] : les auteurs proposent d'enrichir le langage SQL avec les LPP (Limited Patience Patterns). Les LPP sont des expressions permettant de définir des patterns de manière déclarative et algébrique et peuvent être donc intégrés à la syntaxe de base de SQL. Les LPP contiennent des fonctions de bases, telles que : $First(s, k)$: permet de récupérer l'élément k de la sous-séquence s, d'une série chronologique ; $Last(s, k)$: permet de récupérer le troisième élément de la sous-séquence s, en partant de la fin de s ($Last(s, 1)$ correspond au dernier élément de s, $Last(s, 2)$ à l'avant-dernier...) ; $Length(s)$: renvoie la longueur de s ; $Avg(s, c)$ et $Sum(s, c)$: renvoie, respectivement, la valeur de la moyenne et la somme des éléments de la colonne c de s ; $Max(s, c)$ et $Min(s, c)$: renvoie, respectivement, la valeur maximale et la valeur minimale de la colonne c de s ; $prev(e, k)$ et $next(e, k)$: permet de récupérer, respectivement, le k-iéme élément suivant et précédent e. Le langage permet ainsi de définir des expressions telles que $[ALL\ e\ IN\ s](e.price > prev(e, 1).price)$, qui recherche tous les éléments e dans une séquence s, tel que l'attribut `price` de e est strictement supérieur à celui de l'élément qui le précède (uptrend pattern, ou tendance haussière). Il est également possible de créer des expressions plus complexes et plus expressives en combinant plusieurs expressions simples, à l'aide d'opérateurs logiques.

Il est possible d'avoir plus de contrôle sur les résultats en utilisant des directives telles que `NON_OVERLAPPING`, signifiant que le résultat de la requête doit contenir des séquences qui se recouvrent pas les unes les autres (si une séquence vérifie un pattern, il se peut qu'une sous-séquence de cette dernière le vérifie aussi).

Exemple 3.1

```
CREATE PATTERN uptrend AS
SEGMENT s OF quote WHICH_IS FIRST MAXIMAL, NON_OVERLAPPING
ATTRIBUTE date IS last(s,1).date
ATTRIBUTE low IS first(s,1).price
```

```
ATTRIBUTE high IS last(s,1).price
WHERE [ALL e IN s](e.price > prev(e,1).price)
AND length(s) >= 5
```

L'exemple 3.1 est un exemple de LPP. Ce-dernier définie un « uptrend pattern » (tendance haussière). Il recherche toutes les séquences s de la série *quote*, qui soit la plus courte séquence de taille maximal, sans recouvrement entre les séquences (cela veut dire que si une séquence solution est une sous-séquence d'une autre solution, elle est ignorée, pour ne garder que les séquences solution les plus longues et parmi toutes les séquences distinctes trouvées, on retient la plus courte), avec un attribut date, définie comme la date du dernier élément de la séquence, l'attribut low, défini comme le premier élément de la séquence, l'attribut high, défini comme le dernier élément de la séquence et l'attribut e est tel que pour tout e, le prix de l'élément courant est plus grand que celui de son prédécesseur. Enfin, la longueur de la solution doit être supérieure à 5. Autrement dit, il recherche une séquence de s sur laquelle l'attribut *price* ne fait qu'augmenter.

Exemple 3.2

```
SELECT TimeSeries(db.date,db.price)
BY SEARCHING uptrend IN ds.quotes
FROM daily\_stock ds
```

L'exemple 3.2 est un exemple de requête basé sur le pattern défini dans l'exemple 3.1. Cette requête recherche les séquences qui vérifie le pattern *uptrend* dans la série *ds*, composée des attributs *date* pour l'index temporel et *price* pour les valeurs, construite à partir d'une série nommée *daily_stock*. La recherche de pattern est introduite dans le langage SQL, à l'aide de la directive `BY SEARCHING`.

— SQL-TS [SZZA01] : prévu pour la fouille de données continues dans des séquences de taille infinies (streams) et pour la recherche d'information dans un historique. Il se base sur le langage SQL, avec des fonctionnalités supplémentaires, afin de manipuler des séquences de données.

  — L'opérateur `CLUSTER BY` permet de séparer un flux (stream) de données en plusieurs, selon un attribut. Par exemple, si A et B visitent un site web en même temps et que l'on souhaite analyser séparément l'activité de A et l'activité de B sur le site. Si A et B sont connectés en même temps, les informations concernant leurs activités seront mélangées dans un seul flux et il faut séparer (trier) les données concernant chaque utilisateur en deux flux distincts.

  — L'opérateur `SEQUENCE BY` permet de créer une séquence de données, indexées selon un attribut relatif au temps (un attribut de type `DATE`, `TIMESTAMP`...).

  — L'opérateur `AS` permet de définir un pattern (voir exemple 3.3).

Exemple 3.3

```
SELECT B.PageNo, C.ClickTime
FROM Sessions
CLUSTER BY SessNo
```

Time	Time Series key	Value
15 :51 :00	101	0.01
15 :51 :03	102	1.16
15 :52 :07	101	0.04
15 :52 :11	101	0.08
15 :53 :17	103	4.18

TABLE 3.1 – Stockage relationnel de série chronologiques [DF15]

```
SEQUENCE BY ClickTime
AS (A,B,C)
WHERE A.PageType='content'
AND B.PageType='product'
AND C.PageType='purchase'
```

L'exemple ci-dessus utilise les données de `sessions`. Chaque utilisateur possède un numéro de session, pour l'identifier, l'opérateur `CLUSTER BY` est donc utilisé pour séparer le flux des données de sessions, en autant de flux, qu'il n'y a d'utilisateur. Puis les flux sont ordonnés chronologiquement selon l'attribut *clickTime*. Enfin, on cherche à repérer les utilisateurs qui ont visité la page des contenus, avant de cliquer sur un produit particulier et de l'acheter (les actions doivent avoir été réalisée spécifiquement dans cet ordre). Cela se traduit par une séquence de trois tuples consécutifs, nommées A, B et C, à l'aide de l'opérateur `AS`, dont les attributs *PageType* correspondent aux pages désirées, dans l'ordre spécifié. Il existe d'autres mécanismes permettant de définir d'autres type de patterns.

3.2 Stockage des séries chronologiques dans les bases de données traditionnelles, décisionnelles et NoSql

3.2.1 Les séries chronologiques en bases de données relationnelles

Les bases de données relationnelles ont été très critiquées quant à leur capacité à stocker des séries chronologiques. Dans [Nam15], Dmitry Namiot explique que le design n'est pas difficile, à première vue. Il faut une colonne pour les timestamps, et une colonne pour la clé primaire, puis autant de colonnes que de valeurs à stocker. Chaque nouvelle mesure faisant l'objet d'une nouvelle ligne. Dunning et Friedman dans leur livre [DF15] fournissent le même design comme exemple. Ils donnent la table 3.1, comme exemple. La première colonne est le timestamp, permettant d'attacher les valeurs à un instant. La seconde colonne indique à quelle série se rattache la valeur, cette-dernière étant stockée dans la troisième colonne. Il faut également une autre table, qui contient les informations concernant les différentes séries chronologiques. Cependant, plusieurs problèmes se posent alors.

- Si plusieurs dispositifs (machines, capteurs...) peuvent écrire en même temps, le problème

est double. D'abord, la base de données doit pouvoir gérer les programmes qui essaient d'écrire dans la base de données en même temps. Ensuite, si les fréquences des dispositifs (capteurs, par exemples) sont différentes (par exemple, un capteur qui émet une mesure toutes les heures, et un autre toutes les dix minutes), alors de nombreuses lignes auront des attributs manquants et donc beaucoup d'espace mémoire sera utilisé pour rien [Nam15].

- Stocker des séries chronologiques a généralement pour but de réaliser des analyses sur les valeurs (on parle de fouille de données, ou data-mining). Les analyses sont généralement réalisées sur les données les plus récentes (car ce sont celles qui permettent de faire des prédictions cohérentes sur les tendances futures). Ainsi, il faut que la base de données soit capable de renvoyer les données les plus récentes, tandis que la base évolue et est continuellement mise à jour [Nam15].
- Dunning et Friedman [DF15] expliquent également que les bases de données relationnelles sont rapidement inefficaces pour de très gros volume de données. Elles fonctionnent pour des applications qui ne nécessitent pas plus de quelques milliards de point, hors le NASDAQ, par exemple, génère ce volume en trois mois. Les auteurs ajoutent qu'il existe des applications qui génèrent cette quantité de données en une seule journée ;
- Dans [DF15] et [Nam15], il est précisé, l'exploitation (lecture et récupération des données et exécution des requêtes) de la base de données est de plus en plus coûteuse en temps, lorsque le volume de données augmente et certaines applications, comme le machine learning peut requérir et traiter près d'un million de données (valeurs), par seconde.

Mais malgré les critiques, plusieurs travaux ont été proposés afin de stocker des séries chronologiques dans des bases de données relationnelles, en développant diverses techniques pour en corriger les faiblesses et en gommer les défauts. Parmi les technologies proposées, on peut citer :

- TokuDB [Nam15, Bar14] : L'astuce de TokuDB est d'indexer les données à l'aide d'un arbre fractal (Fractal Tree Index). Comme son nom l'indique, l'index est un arbre (non-binaire). L'arbre référence les données dans l'ordre chronologique, ce qui permet de rechercher rapidement une valeur dans l'arbre. Chaque nœud possède une mémoire cache, permettant de sauvegarder en mémoire vive les modifications de l'arbre et ainsi de mieux planifier/optimiser les opérations d'écriture sur le disque. Ainsi, chaque opération d'écriture sera réalisé lorsque le volume à écrire sera considéré comme optimal.
- Vertica [Foc18, Nam15, LFV+12] : Vertica est une plate-forme d'outils d'analyse, qui propose une base de données relationnelle optimisée pour le traitement de séries chronologiques. La plate-forme propose notamment des outils d'interpolation, permettant de remplacer les données manquantes. En effet, il arrive que les intervalles temporels soient irréguliers, ce qui introduit des « trous » dans les données à certains points dans le temps. La plate-forme propose aussi un système de détection d'évènements dans les données et de segmenter (découper des blocs de points successifs) les données autour de ces évènements. Vertica utilise un système de compression des données, en découpant les tables selon les colonnes (partitionnement vertical).
- LittleTable [LLC18, RWW+17] : Il s'agit d'une base de données utilisée par Cisco Meraki (une entreprise qui fabrique des bornes d'accès WiFi, des switchs, des firewalls, des téléphones VoIP et des caméras de sécurité pour les professionnels) depuis 2008, pour stocker

notamment les données statistiques de l'entreprises. Mais aussi des données d'historique et d'autres données elles aussi sous formes de séries chronologiques. Les données sont regroupées selon deux critères : le temps et leur provenance (son appareil d'origine et le réseau dont fait partie l'appareil). La fragmentation temporelle permet de rapidement retrouver les données récentes, sans pour autant induire une perte de performance dans la récupération de données plus anciennes.

À l'instar des deux technologies présentées au-dessus, les données les plus récentes sont d'abord mises en cache en mémoire vive, puis les données du cache sont écrites sur le disque, lorsque le cache atteint une taille limite, ou lorsque la donnée la plus ancienne atteint un âge limite. Une fois la limite atteinte, les données ne sont plus modifiables. Ainsi, les données enregistrées ne sont jamais modifiées (mise à jour), une fois écrites sur le disque. De plus, elles sont toujours ajoutées les unes à la suite des autres (dans l'ordre chronologique).

- TimescaleDB [Fre17] : Timescale est basé le SGBDR PostgreSQL. Mike Freedman explique que les bases de données relationnelles ont du mal à passer à l'échelle, en particuliers à cause du système de pages. En effet, les bases de données sont pensées pour stocker des volumes de données qui ne passent pas en mémoire vive. Il faut donc constamment lire et écrire sur le disque. Ainsi, c'est seulement une portion (page) de 8 kB maximum, qui est chargée en mémoire, afin d'être lue, et/ou modifiée, puis réécrite sur le disque. Une opération de mise à jour, peut donc nécessiter la réécriture de la page en cours sur le disque, puis remplacer la page actuelle avec la page sur laquelle la donnée à modifier est inscrite, avant de réécrire cette dernière sur le disque. Cette opération de changement de page s'appelle un swap (échange). Les opérations de lectures et écritures sur le disque sont très coûteuses en temps, la multiplication de cette opération diminue donc très fortement les performances de la base de données. Or, plus il y a de données dans la base de données, moins il y a de chance pour que la/les donnée(s) à mettre à jour soit déjà dans la page en cours (ou toutes contenues sur une même page), ce qui augmente la nécessité de réaliser des swaps.

  Cependant, l'auteur précise aussi que les séries chronologiques sont un type de données particuliers. En effet, les mises à jour sont rares, surtout sur les données anciennes, et les données entrantes sont souvent des données « fraîches », c'est-à-dire qu'elles sont liées à un intervalle de temps récent. Ainsi, si les données sont triées dans l'ordre chronologique, non-seulement la majorité des opérations seront de nouvelles écritures, mais en plus, il s'agira de données qui viendront à la suite des données déjà présentent. Il est donc possible de limiter au strict minimum les opérations de swaps.

  Ainsi, l'auteur détaille la stratégie employée pour tirer avantage de la particularité des séries chronologiques. La méthode consiste à fractionner verticalement (à séparer les colonnes d'une même table), selon deux dimensions : la dimension temporelle et la dimension de la clé primaire. Ce fractionnement permet d'avoir des données ordonnées chronologiquement et de réduire la taille (espace mémoire) de chaque table, ainsi, cela diminue la nécessité des swaps et permet de gros gains en performances.

Ainsi, on note que la plupart des technologies proposées tirent avantages de la particularité temporelles des données contenues dans les séries chronologiques, pour optimiser les systèmes relationnels au traitement de ces données. La technique fondamentale est l'utilisation de la mémoire vive pour stocker les données les plus récentes avant de les écrire. En effet, ce sont les données les plus susceptibles d'être mises à jour, et cela permet de ne pas lancer une

opération d'écriture sur le disque à chaque nouvelle donnée, mais d'écrire des blocs très larges de données en une fois. L'écriture en blocs étant plus efficace, car elle limite le nombre de demande d'accès au disque, et la nécessité des échanges. Les données sont également triées et/ou regroupés de manière chronologique.

Mais si ces solutions font l'effort de proposer des systèmes capables de gérer la quantité de données que génèrent les séries chronologiques, elles s'attaquent rarement aux problèmes spécifiques aux séries chronologiques. Vertica par exemple, est orienté analyse de données et proposent des fonctionnalités comme l'interpolation. Mais la plate-forme fait tout de même usage du langage SQL standard. Ce langage peut, tout de même, avoir son utilité, si l'utilisateur souhaite réaliser des actions qui rentre dans le cadre de l'algèbre relationnelle, mais nous avons vu qu'il manquait d'expressivité, dans le cas des séries chronologiques. Or, si l'utilisateur souhaite analyser des séries chronologiques, il aura besoin de fonctionnalités statistiques qui sortent des limites du langage SQL.

3.2.2 Les séries chronologiques en entrepôts de données

Les entrepôts de données sont pensés pour stocker des données liées à une période temporelle (des historiques), et pour gérer les gros volumes. Les entrepôts de données sont donc bien adaptés au stockage et à l'analyser des séries chronologiques [EN10]. Nous présentons ici quelques exemples.

- Un tableau de bord pour l'analyse de fréquence de mots clés [Dah17] : l'auteur décrit un système d'analyse de séries chronologiques, dont les valeurs sont des mots clés. L'objectif étant de proposer un tableau de bord, qui serait un support à l'analyse de fréquences de mots clés et s'applique aux réseaux sociaux. L'auteur explique que le système contient un processus ETL dit « streaming based » en plus du gestionnaire de données. L'ensemble n'est pas proprement défini comme un entrepôt de données, mais on en retrouve les caractéristiques : des sources variées (réseaux sociaux divers) ; un gros volume de données (typique des réseaux sociaux actuels) ; un processus ETL ; une application cliente dédiée, offrant des outils spécifiques, pour aider l'utilisateur à exploiter les données. Les données sont par définitions des séries chronologiques, chaque mot clé apparaissant à une certaine date.
- SHAPE (Statistical Hybrid Analysis for load ProfilE) [LGCE14, LGCP15] : Il s'agit d'un projet visant à créer une plate-forme en-ligne, pour les compagnies d'électricité. Cette plate-forme comprend un entrepôt de données permettant de stocker : les profils de charge quotidien, un profil de charge est une série chronologique de 96 points, où chaque point représente l'énergie utilisée dans les 15 dernières minutes ($96 * 15$ minutes faisant bien 24 heures) ; des informations techniques et commerciales complémentaires ; et des relevés de températures. Les données sont extraites de sources de données diverses, appartenant à différentes entreprises. La plate-forme sert à faire des prédictions sur les futures charges, permettant d'aider à la gestion de l'électricité.
- Un entrepôt de données, pour le trafic routier [BMCDV$^+$06] : Le suivi et contrôle du trafic

routier sur autoroutes fait appel à un ensemble d'outils développés par l'INRETS (Institut National de REcherche sur les Transports et leur Sécurité). Ces outils fournissent des rapports périodiques sur un aspect du trafic. Les différents aspects sont :

— Macroscopiques : ce sont des données telles que le débit, la vitesse moyenne de circulation, le taux d'occupation, la concentration (au sens densité des véhicules) et les temps de parcours ;

— Topologiques : la structure du réseau de capteurs (les positions des capteurs sur le réseau) ;

— Événementielles : les divers événements qui sont à l'origine de perturbation sur le réseau (panne d'un feu de circulation, travaux, accidents...) ;

— Météorologiques et environnementales (neige, verglas, pollution...).

Ainsi, les diverses sources et capteurs qui permettent d'obtenir ces données sont nombreuses et hétérogènes et l'ensemble produit un volume de données conséquents. Ainsi, un entrepôt de données est bien adapté à la situation.

De plus, les données macroscopiques (les mesures de capteurs) sont dites temporelles (elles sont également spatiales, dans le sens ou elles proviennent de capteurs physiques, donc localisé dans l'espace), car chaque mesure fournie par un capteur, est attaché à un instant t. Elles prennent donc naturellement la forme de séries chronologiques. Le but du projet CADDY est de tirer parti, au maximum, et même étendre les fonctionnalités OLAP de l'entrepôt de données, pour analyser le plus finement possible, les données de l'entrepôt.

- PEPR (Public Expression Profiling Ressource) [CZM+04] : Il s'agit d'un entrepôt de données utilisé en biologie pour analyser des protéines et le RNA (RiboNucleic Acid). Il s'agit de données très volumineuses et temporelles. L'entrepôt fourni un outil d'analyse de requête sur les gênes (Single Gene Query Tool), afin de générer des graphes et des tableaux. L'entrepôt utilise des tableaux multidimensionnels.

Dans le cas des entrepôts de données, ils sont généralement développés et adapté pour une ou plusieurs applications précises. Lorsque cette application se base sur des séries chronologiques, les fonctionnalités nécessaires à l'application, mais qui ne serait pas déjà fournies par la base relationnelle sur laquelle repose l'entrepôt, ou par les outils OLAP, est alors implémentée. Cela génère un coût de développement supplémentaire, qui pourrait être absorbé par l'utilisation d'une base de données déjà adaptée aux séries chronologiques (à l'aide d'un gestionnaire de séries chronologique, par exemple).

3.2.3 Séries chronologiques en NoSQL

Les technologies NoSQL, en s'affranchissant du standard relationnelles sont une excellente base pour développer des gestionnaires de séries chronologiques. Nous présentons ici quelques exemples de technologies de stockage de séries chronologiques, développées à l'aide de bases de données NoSQL.

- Hadoop TS [KK13] : Un framework dédié à l'analyse de séries chronologique. Il permet de distribuer le stockage et la charge de calcul sur plusieurs machines. Il ne s'agit pas d'un

serveur de données, mais d'une plate-forme, capable d'optimiser les opérations classiques (jointure...) sur un large ensemble de données.

- OpenTSDB [Aut17, DF15] : basé sur HBase et MapReduce. Il implémente un « Time Series Deamon » (TSD). Il s'agit d'un processus en tâche de fond, qui a pour rôle de trier les données entrantes (les nouveaux relevés d'une série en cours), afin de les ajouter à la bonne série chronologique. Prenons un exemple concret pour comprendre. Si une maison est équipée d'un ensemble de capteurs (températures et humidité dans chaque pièce), chaque capteur ne peut sauvegarder lui-mêmes ses propres données, à moins de les munir chacun de leur propre disque. Ils envoient donc régulièrement leurs mesures à un boîtier central, qui lui est en mesure de stocker les données dans une base de données. Mais, la base de données contiendra une série par capteurs, et il appartient au boîtier d'assurer que chaque nouvelle est bien ajoutée à la série correspondante. Le TSD a aussi pour rôle de compresser les données anciennes, afin de les sauvegarder sous une forme moins gourmande en mémoire, mais plus lente d'accès, le cas échéant.
- R2Time [ACRWW14] : est un framework dédié à l'analyse de données en R. Le framework utilise R comme environement d'analyse numérique relié à une base de données OpenTSDB.
- KairosDB [Nam15, Kai15] : KairosDB [Nam15] est un système similaire à OpenTSDB, mais basé sur Cassandra. Cassandra (Apache Cassandra) [Fou16, Fou17] est un SGBD non-relationnel, qui utilise le partitionnement des données, pour pouvoir les distribuer sur plusieurs machines.
- PhilDB [Mac17, Mac16] : Une base de données de séries chronologiques, pensée pour des séries dont les valeurs peuvent changer (être corrigées) après leur stockage initial. En hydrologie, par exemple, les valeurs peuvent être corrigées à l'issue d'un processus de contrôle qualité. Les mises à jours sont journalisées, lorsqu'elles se produisent, ainsi les valeurs d'origines et les mises à jours successives sont toutes conservées.
- SciDB [SBPR11] : SciDB est un système de données pensé pour des analyses scientifiques. Il intègre des opérateurs statistiques et d'algèbre linéaire. Il a été conçu pour supporter un volume conséquent (de l'ordre du pétaoctet) de données. Il tire avantage d'une architecture distribuée (cloud), dont chaque machine est indépendante. Il possède un langage de requête proche du langage SQL et il possède aussi un langage de programmation fonctionnel.
- RiakTS [BT11] : Un SGBD de type clé-valeur, distribuée et optimisée pour lire et écrire rapidement des données temporelles. Il prend avantage de son caractère distribué, pour assurer qu'aucune données ne sera perdue, en cas de panne d'un nœud. Les données peuvent être aussi bien structurées que semi-structurées ou non-structurées. Le système permet de réaliser des requêtes sur un gros volume de données chronologiques et de vérifier, en temps réel, l'intégrité des données entrantes. Un langage de requête est également proposé pour simplifier l'analyse des données sans que l'utilisateur n'est à coder lui-même tout le processus ETL nécessaire à ces besoins.
- Druid [dru, dru17, YTL+14] : Un magasin de données orienté colonne, conçu comme un entrepôt de données. Il contient des outils équivalent aux outils OLAP (drill-downs, agrégations...) des ED. Le système est distribué sur plusieurs nœuds indépendants. Il permet de réaliser des requêtes sur des jeux de données qui peuvent atteindre le pétaoctet. Il a

été pensé pour permettre l'analyse en temps réel, des données entrantes, avec une capacité d'ingérer des données nouvelles très rapidement.

Le problème principal de ces technologies, c'est qu'elles prennent, puisqu'il s'agit de leur première raison d'être, principalement en compte l'aspect « Big Data » du stockage de séries chronologiques, mais qu'elles ne prennent pas forcément en compte (ou pas intégralement) les besoins spécifiques aux séries chronologiques. Quelques systèmes pensent à prendre en compte, en partie, le besoin en fonctionnalités dédiées aux séries chronologiques. Parmi les exemples donnés, R2Time est un framework qui comble le manque de fonctionnalités d'OpenTSDB, en associant cette dernière à un environnement numérique (R). L'association des deux permets d'avoir un environnement complet. Ensuite, il y a RiakTS, qui intègre un langage dédié à l'analyse des données et SciDB, qui met l'accent sur l'implémentation de fonction statistiques.

3.2.4 Séries chronologiques en NewSQL

Les technologies NewSQL semblent assez peu répandues, ou connues, CrateDB [Cra17] étant le seul exemple, que nous ayons trouvé. Il propose un nouveau moteur SQL, qui a été conçu pour faciliter le passage à l'échelle dit horizontal. Autrement dit, pour supporter la distribution (sharding, voir 2.3.2) des données sur plusieurs machines. L'intérêt de ce mécanisme est l'ingestion de plusieurs millions de nouveaux éléments (nouveaux points dans la(les) série(s) stockée(s)) par seconde, permettant de faire face à des débits de données élevés. Le moteur permet également de répartir la charge de calcul des requêtes et ces dernières sont optimisées, notamment, par l'utilisation du partitionnement.

En interne, les données sont en réalité stockées au format JSON (format textuel), mais il est masqué par le moteur SQL, de sorte que la BD soit relationnelle du point de vue externe de l'utilisateur final. Cette astuce ne permet pas de s'affranchir d'un schéma de données, mais celui peut cependant varier, sans avoir à redévelopper la BD. Enfin, le moteur SQL permet l'utilisation du langage SQL standard. Il propose également quelques fonctionnalités supplémentaires, comme les requêtes géospatiales, permettant de réaliser des requêtes dans séries de données sous la forme de coordonnées GPS (latitude, longitude) et la possibilité pour l'utilisateur de créer et exécuter ses propres fonctions.

Dans cet unique exemple, nous noterons que le langage proposé est toujours le langage SQL, inadapté aux séries chronologiques. Tout comme les exemples relationnels données dans la section 3.2.1, la solution données tentent avant tout d'absorber rapidement les données, en tirant avantage du fait que les séries chronologiques sont rarement mises à jour (et que le données les plus susceptibles d'être mises à jour, sont les données les plus récentes). Il proposes cependant quelque fonctionnalités supplémentaires, mais ne propose pas une extension de fonctionnalités complète (pas de langage étudié pour proposer toutes les fonctions de base nécessaires), permettant de manipuler et analyser les séries chronologiques.

3.3 Stockage de modèles mathématiques

Dans la section précédente, nous nous sommes attardés sur les technologies de stockages spécifiques aux séries chronologiques et leur raison d'être. Nous avons vu que les séries chronologiques méritaient effectivement, un soin particulier, quant à leur stockage et à leur gestion. Nous avons vu de plus près l'intérêt d'utiliser des gestionnaires de séries chronologiques en nous penchant sur les diverses techniques, basés sur des bases de données relationnelles, décisionnelles ou NoSQL. Ainsi, les stockages de séries chronologiques est un objet d'intérêt, dans la communauté bases de données, depuis le début des années 90. Ce qui manque aujourd'hui, d'un point de vue scientifiques, c'est le stockage de modèles mathématiques et la mise en lien avec les données expérimentales (séries chronologiques, pour notre cas d'étude). Dans les travaux visant à stocker des modèles mathématiques dans des bases de données, il existe divers travaux, issues de domaine scientifiques variés.

— Dans [RTB+06], les auteurs présentent QxDB, présenté comme une amélioration d'une base de données existantes nommée QKDB. Dans leur article, les auteurs s'intéressent aux domaines de la physiologie et de la physiopathologie. Ils expliquent que les modèles utilisés dans ses domaines utilisent des paramètres dont la fiabilité est cruciale. Ces paramètres font l'objet de nombreuse expériences publiée dans la littérature scientifique du domaine. Or, en oncologie notamment, le nombre de publication augmente de façon exponentielle, rendant le travail d'investigation difficile pour la communauté de chercheurs du domaine. Pour faciliter ce travail, les auteurs proposent une base de données relationnelle, développé avec MySQL et munie d'une interface web, pour permettre aux utilisateurs de naviguer dans la base de données et y rechercher les publications, modèles et paramètres qui les intéressent. Il existait une base de données similaire, nommée QKDB, mais cette dernière se limitait à la physiologie des reins. Le souhait des auteurs était de proposer une solution plus générique, bien que toujours limitée aux domaines de la physiologie et de la physiopathologie.

— Dans [DSK10], les auteurs expliquent comment ils s'attendent à « une avalanche de nouveaux modèles d'astéroïdes dans la prochaine décennie » et propose une base de données nommées DAMIT (Database of Asteroid Models from Inversion Techniques), afin de fournir une plateforme de partage des modèles d'astéroïdes trouvées par la communauté de recherche en astronomie. Les astéroïdes sont décrits par un ensemble de paramètres physique : forme, axe de rotation et période de rotation. Et leur forme est définie à l'aide d'un maillage de polyèdres décrits par les coordonnées cartésiennes de leurs sommets. Chaque astéroïde est associé à un certain nombre d'information supplémentaires, tel que le modèle utilisé pour obtenir la forme à partir des données brutes, ainsi qu'un lien vers l'article dans lequel les informations sur l'astéroïdes ont été publiées. La base de données a été implémentée avec MySQL et est accessible en ligne avec MySQL server et une interface Web permet de naviguer dans la base de données et d'en extraire du contenu dans un fichier texte ou de générer (par ordinateur) des images (au format png) des astéroïdes, à partir des données sur leur forme. Leur base

de données est accessible en ligne [1].

— Dans [PWB+11], les auteurs expliquent que de plus en plus de structures expérimentales de protéines sont disponibles. Cela entraîne une augmentation du nombre de modèles fiables de protéines. La base de données contient des modèles 3D, conçu à l'aide de logiciel spécialisés. La base contient 10 355 444, que l'utilisateur peut mettre à jour quand il le souhaite et demander la modélisation d'une séquence supplémentaire, à l'aide d'une interface sur un serveur spécialisé, nommé ModWeb. Le serveur ModWeb associé à la base de données permet de faire de la modélisation comparée. Pour cela, le serveur a besoin de données d'entrées, qui peuvent lui être fournit sous deux formes.

  — Le serveur accepte des séquences peptidiques, dans un format nommé FASTA. Puis, afin d'en trouver le modèle, il s'appuie sur une banque de données sur les différentes protéines connues (Protein Data Bank).

  — Il est aussi possible de fournir des structures de protéines, dont on ne sait à qu'elle séquence peptidique elle correspond. À partir de cette structure, le serveur ModWeb est alors capable de générer les différentes séquences qui aurait pu produire cette structure et de rechercher les séquences générées dans une base de séquences peptiques connues, nommées UnitProtKD.

— Dans [KXD12], les auteurs proposent une base de modèles d'objets pour la préhension d'objet en robotique. Les auteurs expliquent que pour manipuler correctement un objet, avec un robot, il faut un modèle précis des objets utilisés. Mais les méthodes d'apprentissage dynamique (apprentissage par expérience, à l'instar de l'humain) n'était pas encore suffisamment mature. Ainsi, la méthode la plus répandue, consiste à fabriquer le modèle et de le charger dans le robot, avant de l'activer (apprentissage hors-ligne). L'objet est scanné (il peut être scanner plusieurs fois dans diverses positions, en fonction de sa complexité), pour obtenir un maillage (ici, les polyèdres sont des triangles) 2D de la surface de l'objet. Une fois la forme obtenue, on y ajoute des indications sur la texture de la surface, car la texture est importante pour la préhension d'un objet (un objet lisse, aura plus de chance d'être glissant et donc plus difficile à tenir, qu'un objet dont la surface est plus rugueuse). Chaque objet est associé à : (i) un maillage, représentant la forme sa forme et dont il existe plusieurs versions (plusieurs résolutions) ; (ii) un ensemble d'image sous différents angles ; (iii) les données de textures de la surface ; (iv) les données contenant la stratégie de préhension choisie pour l'objet, représentée dans des fichiers XML. Les auteurs annoncent qu'ils ont implémenté une base de données avec une interface web, pour que le plus grand nombre puisse accéder en-ligne aux données contenues dans leur base de données. Ils ne précisent cependant rien sur le type de base de données et la technologie utilisée.

— Dans [WGH+14], les auteurs expliquent que le nombre de publications de modèles mathématiques de processus biologiques augmente. Ils veulent tirer avantage d'un format standard de représentation de ces modèles (SBML, Systemes Biology Markup Language, basé sur le langage XML), déjà très répandu dans la communauté des biologistes, pour construire une base de données capable de stocker ces modèles de processus. Cela permettrait d'indexer, de rechercher et retrouver plus facilement les modèles. Dans leur

1. `http://astro.troja.mff.cuni.cz/projects/asteroids3D`

article, ils proposent une méthodologie de traduction des modèles SBML en triplets RDF (Resource Description Framework). Le résultat de cette traduction est nommé *BioModels Linked Dataset* et chaque triplet d'un *Linked Dataset* est identifié par une URI (Uniform Resource Identifier). Les données sont stockées dans OpenLink Virtuoso et les triplets sont rendus accessibles grâce à SPARQL. La base de données est également accessible sur le net [2].

— Dans [GBKF15], les auteurs se penchent sur le cas de l'agriculture. Dans ce domaine, il y a eu une augmentation du nombre de publications de modèles mathématiques d'organismes dangereux dans la chaîne alimentaire. Cependant, il n'y a pas de standard établit sur la méthode de modélisation. Leur but est donc de fournir un standard et une base de données accessible en ligne. Pour cela, ils proposent un standard basé sur le langage SBML, déjà largement adopté par les biologistes et ont implémenté un dépôt de test, nommé PhyM-Database [3].

— Dans [GHS16], les auteurs cherchent à proposer une base de modèles gratuite accessible via le web. Les modèles sont soit purement mathématiques, soit basé sur la physique. Les auteurs expliquent, qu'après avoir interrogé non seulement des chercheurs, mais également des ingénieurs, ils ont la confirmation, qu'il y a une réelle demande pour ce type d'outils. L'idée est de proposer un accès à des modèles fiables, éventuellement accompagnés d'information utile supplémentaire, tel que des exemples de code implémentant le modèle, des résultats de simulation, ou encore des liens vers des publications liées aux modèles. Une version bêta de la base de données est accessible en ligne [4].

— Dans [KAL+17], les auteurs expliquent que les technologies récentes ont permis aux biologistes de divers domaines de récolter d'énorme quantités de données. Ce qui a entraîné le recours de modèles mathématiques, pour modéliser, étudier, analyser les divers processus rencontrés par les biologistes dans leurs domaines respectifs. Les modèles sont généralement des équations différentielles, tantôt ordinaires, tantôt partielles, des équations stochastiques ou aussi, des modèles hybrides. Les auteurs expliquent qu'avoir accès à des modèles biologiques précis décrivant divers phénomènes, facilitent la composition de modèles plus complexes, permettant de modéliser des systèmes vivants, si bien que selon les auteurs, à l'heure de la rédaction de l'article, plus de 1000 modèles complexes différents avaient été publiés et ce nombre continue d'évoluer. Mais, la réutilisation, l'extension ou la modification de ces modèles est un travail laborieux, car ils sont présentés comme de gros modèles intégrant un certain nombres de sous-modèles (les systèmes vivants sont très complexes et peuvent être vu comme des assemblages de sous-systèmes plus simples et interagissant entre-eux) et il n'existe pas d'outils permettant de les décomposer/décortiquer de manière automatique ou semi-automatique. Les auteurs proposent donc un outil qu'ils ont appelé : Mathematical Models of bioMOlecular sysTems (MAMMOTh), qui comprends une base de données stockant un ensemble de sous-modèles, qui constituent des modèles plus complexes. Les sous-modèles sont obtenus par décomposition de modèles globaux ayant fait l'objet d'une publication scientifique. Le but de cette base étant de faciliter ce travail de décomposition/décorticage et

2. `https://www.ebi.ac.uk/biomodels-main/`
3. `https://sites.google.com/site/test782726372685/tutorial`
4. `http://mathematicalmodels.put.poznan.pl`

de reconstruction des modèles, afin de supporter le travail de conception de modèles de plus en plus complexes, permettant de modéliser de plus en plus de systèmes vivants.

Dans l'ensemble de ces travaux, les diverses solutions proposées de base de modèles partagent trois objectifs principaux :

— fournir un standard (s'il n'existe pas) de représentation d'un modèle et afin de proposer une solution de stockage permettant d'organiser et retrouver plus facilement un/des modèles ;

— le stockage et l'organisation des modèles a pour but de faire face à une « explosion de la quantité modèles » mis au point et publiés par les scientifiques, une « explosion » qui est une conséquence directe du « déluge de données » dû aux « big data » ;

— un troisième objectif poursuivi par les bases de modèles est de facilité le travail de vérification des sources et de la fiabilité des modèles, car il s'agit d'une notion essentielle dans tous travaux scientifique et l'augmentation exponentielle du nombre de modèles publiés rends la tache de vérification à la fois indispensable et beaucoup plus ardue.

Les objectifs donnés plus haut découlent d'un besoin qui s'est développé aussi bien parmi les chercheurs, que les communautés plus techniques (les ingénieurs) [GHS16] et dans de nombreux domaines scientifiques, ici, biologie, médecine (physiologie et physiopathologie), agronomie, astronomie, robotique... Nous ajoutons l'automatique à cette liste non exhaustive.

3.4 Formats d'échange de modèles mathématiques

3.4.1 Formats d'échange de données « brutes »

MathML : Selon [WC06], le langage TEX a été développé avant l'essort d'internet et ne l'a donc pas anticipé. Ce langage avait pour but de faciliter l'écriture de formules mathématiques dans les documents scientifiques. Il était donc plutôt destiné aux chercheurs, enseignants... qui produisent des documents scientifiques et/ou académiques (articles, support de cours...). Mais le langage était avant tout développé pour faciliter l'impression des formules, c'est-à-dire optimisé pour fournir un rendu humainement lisible des formules mathématiques. Avec l'essors d'internet (et de l'informatique en général), une autre problématique c'est posé : échanger des formules mathématiques entre deux programmes informatiques. Autrement dit, ne plus considérer l'écriture sous forme humainement lisible des formules, mais s'attacher à créer un format de réprésentation compréhensible pour un programme informatique [Row06]. Ainsi, MathML est un langage de description basé sur le langage XML, qui permet, comme le langage TEX, de spécifier un format d'écriture humainement lisible des formules mathématiques, mais contient aussi un mécanisme de capture de la sémantique des éléments et des symboles apparaissant dans le document. Les différents objets représentés en MathML peuvent donc combiner des éléments orientés présentation graphique et des éléments purement sémantiques [ABC+14, Row06, WC06].

Le langage MathML permet à un programme informatique de comprendre les concepts ca-

chés derrière les symboles mathématiques, ce qui permet une meilleure qualité de recherche de contenu mathématique (le programme est capable de faire une recherche sur la sémantique des éléments et non uniquement textuelle, sur la base des symboles). Contrairement au langage TEX, MathML n'est pas pensé pour être écrit par un humain, il existe des logiciels d'édition, qui permettent de définir graphiquement des formules et les objets mathématiques qui les composent, de sorte que le logiciel puisse éditer le fichier MathML correspondant. L'utilisation du langage XML comme base permet de se baser sur une large quantité d'outils de manipulation du langage XML, ce dernier étant un standard très largement répandu, notamment sur le web. Cela limite le coût de développement des applications, plugs-ins... qui font usage de MathML.

OpenMath : OpenMath est également un langage de représentation de formules mathématiques basé sur le langage XML [Soc17, Dew00]. La différence avec MathML, c'est que ce langage ne prend en compte que la sémantique des formules et objets mathématiques [DNS04]. Comme MathML, il se destine surtout à l'échange de formules mathématiques entre programmes informatiques et n'est pas conçu pour être édité directement par un humain.

L'étendu des connaissances du langage (les différentes notions mathématiques et symboles qu'il maîtrise) est partagée dans des « Content Dictionaries » (CD). Il existe un CD officiel, mais il est possible de l'étendre, permettant à chaque utilisateur de définir des symboles et des notions dont il a besoin, qui ne serait pas présente dans le CD officiel. Le CD défini un certain nombre d'objets mathématiques (fonctions, opérateurs, ensembles...), et contient :

— le nom de l'objet mathématique défini ;

— une définition textuelle (en langage naturel) de l'objet ;

— une définition mathématique et textuelle des propriétés de l'objet (optionnel) : il s'agit de commentaires (« Commented Mathematical Properties », abrégé CMP), qui viennent compléter la définition donnée plus haut et rédiger à l'aide de symboles mathématiques ;

— une définition mathématique en langage OpenMath des propriétés de l'objet (optionnel) : il s'agit d'une réécriture en langage OpenMath (« Formal Mathematical Properties », abrégé FMP) des propriétés mathématiques données plus haut ;

— un ou plusieurs exemples d'utilisation (optionnel).

La figure 3.1 est un exemple de CD. Ce dernier défini la fonction `log`. La balise `Name` donne le nom de la fonction définie. La balise `Description` décrit le fonctionnement de la fonction, tandis que la balise `CMP` défini une de ces propriétés à l'aide d'une formule mathématique conçue pour être comprise par un humain. Ensuite, la même propriété est écrite en langage OpenMath, de manière à être comprise par un programme. Enfin, la balise `Exemple` contient un exemple du comportement de la fonction.

```
<CDDefinition>
<Name>log</Name>
<Description>
This symbol represents a binary log function; the first argument is the
   base, to which the second argument is loged. It is defined in
   Abramowitz and Stegun, Handbook of Mathematical Functions, section 4.1
</Description>
<CMP>
a^b = c implies log_a c = b
</CMP>
<FMP>
<OMOBJ>
<OMA>
<OMS cd="logic1" name="implies"/>
<OMA>
<OMS cd="relation1" name="eq"/>
<OMA>
<OMS cd="arith1" name="power"/>
<OMV name="a"/> <OMV name="b"/>
</OMA>
<OMV name="c"/>
</OMA>
<OMA>
<OMS cd="relation1" name="eq"/>
<OMA>
<OMS cd="transc1" name="log"/>
<OMV name="a"/>
<OMV name="c"/>
</OMA>
<OMV name="b"/>
</OMA>
</OMA>
</OMOSJ>
</FMP>
<Example>
log 100 to base 10 (which is 2).
<OMOBJ>
<OMA>
<OMS cd="transc1" name="log"/>
<OMF dec="10"/>
<OMF dec="100"/>
</OMA>
</OMOBJ>
</Example>
</CDDefinition>
```

FIGURE 3.1 – Exemple de CD, pour la fonction `log`

3.4.2 Formats d'échange de modèles et de méthodes de calculs

Il existe une proposition de standard, servant le même objectif, mais basé sur le format JSON [PBG16]. Il est dis extensible, à l'inverse de PMML.

Predictive Models Markup Language (PMML) : Le langage PMML (pour « Predictive Models Markup Language ») est un standard, basé sur le langage XML, qui permet d'échanger des modèles prédictifs couramment utilisés en machine learning [GZLW09]. Aussi, depuis sa version 4.0 (la version actuelle est 4.3 [5]), le langage PMML supporte les séries chronologiques [Pec09] (permet de définir des données sous formes de séquences de valeurs, avec un index temporel et de définir une méthode d'interpolation des valeurs). Les modèles sont fournis « prêt à l'emploi », c'est-à-dire que le langage permet de définir les modèles utilisés, ainsi que les données d'entrées utilisées pour l'entraîner et les données de sorties obtenu avec le modèle. Les librairies qui interprètent le langage PMML implémentent les algorithmes permettant d'utiliser les modèles supportés par PMML. Ce qui allège les programmes lecteurs du coût de développement des implémentations des modèles utilisés. Le langage permet non-seulement de définir les modèles et les données qui vont avec, mais également les processus de pré et post-traitement sur les données, notamment les méthodes d'extractions des bonnes « features » (« caractéristiques »), le cas échéant. D'après [Pec09, GZLW09], le langage PMML est devenu un standard de fait, dans le domaine de la fouille de données et est utilisé même dans l'industrie. Un fichier PMML contiens divers éléments.

- Un en-tête `Header` (obligatoire) : il contient des informations générales sur le fichier (l'application qui a généré le fichier, la version de PMML utilisée...) ;
- Un `Mining Build Task` (optionnel) : il contient des informations supplémentaires sur la façon dont le modèle a été obtenu.
- Un dictionnaire de données `Data Dictionary` (obligatoire) : il contient les détails des différentes variables (dites « Data fields » : champ de données) des données d'entrées. Les variables possèdent un nom de champ, un type (entier, réel, énumération...) et un type d'utilisation (valeurs continues, valeurs de catégories...). Il peut également contenir une explication de ce que décrit la variable et sont utilité dans le modèle, ainsi qu'une liste des valeurs autorisées et/ou interdites et/ou manquantes.
- Un dictionnaire de transformations `Transformation Dictionary` : il contient les différentes transformations à réaliser sur les données d'entrées, pour qu'elles soient exploitables par le modèle. Les différentes transformations peuvent être : (i) la normalisation ; (ii) la discrétisation d'un champ continu ; (iii) des mappings entre divers attributs ; (iv) des fonctions permettant de calculer un nouvel attribut, à partir d'un ensemble d'attributs ; (v) et des fonctions d'agrégation.
- un modèle `Model` : décrit un modèle, correspondant un algorithme de fouille de données spécifique. Le langage contient un certain nombre de modèles [Pec09, GZLW09] [6] :

5. `http://dmg.org/pmml/v4-3/GeneralStructure.html`
6. `http://dmg.org/pmml/v4-3/GeneralStructure.html`

Règles d'association ; Clustering ; Régression générale ; Composition de modèles : arbres de décision et modèles de régression qui peuvent être utilisés pour chaîner ou sélectionner plusieurs modèles ; Naïve Bayes ; Réseaux de neurones ; Support Vector Machines ; Arbres de décision ; K plus proches voisins... Les modèles sont représentés à l'aide d'un ensemble de sept éléments.

- `Mining Schema` : sers à décrire les paramètres que l'utilisateur doit fournir avant de pouvoir utiliser le modèle.
- `Outputs` : sers à décrire les résultats que le modèle est supposé fournir.
- `Model Statistics` : contiens une analyse des résultats du modèle, qui découle généralement du processus global d'analyse des données et peut-être utile à la validation du modèle.
- `Targets` : détaille l'intérêt des attributs des données d'entrées et des attributs dérivée, de façon à clarifier la manière dont les sorties du modèle sont calculées.
- `Model Explanation` : contiens quelques détails supplémentaires sur le modèle. Ces-derniers ne sont pas nécessaires à son utilisation par un programme, mais permet à un utilisateur de mieux appréhender le comportement et la qualité du modèle.
- `Local Transformations` : un ensemble de transformations spécifiques au modèle en question et qui n'ont de sens, que dans le contexte de ce-dernier.
- `Model Verification` : contiens un jeu de données d'entrées associé aux données de sorties correspondantes, permettant de vérifier le bon fonctionnement du modèle.

Les fichiers PMML sont édités par un programme, il existe par exemple une plateforme, nommée KNIME [MSLB11], qui contient une interface graphique. Via cette interface, l'utilisateur peut définir un modèle et les diverses informations nécessaire à l'élaboration d'un fichier PMML, qui sera ensuite automatiquement édité.

Portable Format for Analytics (PFA) : Le langage PFA (Portable Format for Analytics) a été développé pour aider au déploiement de modèles analytiques dans un environnement dont les contraintes peuvent être strictes. C'est-à-dire que le code doit être impérativement et rigoureusement testé [PBG16]. Le langage a été développé dans l'objectif de satisfaire les contraintes suivantes : (i) le langage doit être extensible, pour pouvoir représenter une certaine diversité de modèles ; (ii) il doit permettre la composition de modèle ; (iii) il doit être facile à intégrer dans un environnement distribué et/ou un environnement dirigé par les évènements ; (iv) et il doit être possible de le déployer dans un environnement opérationnel, sans risque de perturber la fonction dudit environnement.

Le langage PFA est basé sur le langage JSON, qui, tout comme avec le langage XML, permet d'avoir un langage multi-plateforme et d'économiser une partie du coût de développement, le langage JSON étant également un standard répandu. Un fichier PFA contient ce que les auteurs appellent un *moteur PFA* (PFA engine). Il s'agit d'un programme définit par trois éléments :

- l'élément `"input"` décrit le type des données d'entrées du programme, qui peuvent être basiques (entier, réel...) ou plus complexes (enregistrements...) ;

— l'élément `"output"` décrit cette fois les données de sorties du programme ;

— et l'élément `"action"` décrit le processus (modèle) qui permet de calculer les sorties à partir des entrées.

Les auteurs donnent un exemple trivial de moteur PFA : `{"input": "null", "output": "null", "action": "null"}`. Il s'agit du plus simple exemple de programme valide. Ce dernier ne prend rien en entrée, ne rend rien et ne fais rien. En revanche, il permet de voir que le programme est bien exprimé selon les règles du langage JSON.

Le langage PFA permet à l'utilisateur de définir ses propres fonctions, qu'il peut écrire directement en PFA, ce qui permet notamment de combiner plusieurs modèles, mais aussi d'étendre les possibilités du langage, lorsque ça s'avère nécessaire. Les fonctions que l'utilisateur ajoute n'ont pas besoin d'être officiellement ajoutées aux langages PFA pour qu'il fonctionne sur n'importe quelle plateforme. En effet, celles-ci étant écrites en PFA, un programme lecteur est en mesure de lire et comprendre la définition, afin de l'exécuter correctement, même si la fonction n'est pas connue dans le standard.

Bien que le langage PFA soit un langage de programmation (même si le langage JSON n'en soit pas un), le langage possède un certain nombre de restriction (particulièrement en termes d'accès aux fichiers systèmes de l'hôte sur lequel il s'exécute), qui assure que son exécution ne puisse pas dégrader le système.

L'avantage du langage PFA sur le langage PMML est son extensibilité. En effet, en PFA, l'utilisateur peut définir de nouvelles fonctions directement dans le langage, alors qu'en PMML, c'est dans les librairies de lecture, qu'il faut les implémenter, ce qui signifie qu'il faut que toutes les nouvelles fonctionnalités doivent être approuvées et intégrées au standard PMML. En termes d'application, dans le domaine de la fouille de données, le langage PMML est le plus répandu [PBG16].

3.5 Conclusion

Le besoin de développer des méthodes standards d'échange de modèles théorique est partagé à travers les communautés des chercheurs scientifiques et même parmi des communautés techniques tel que les ingénieurs [GHS16]. Ce besoin vient d'une augmentation de données disponibles à analyser, qui ont entraîné la conception de modèles permettant de comprendre et interpréter ses données. Plusieurs initiatives de partages de modèles dans divers domaines (de la fouille de données, à l'astronomie, en passant par la biologie).

— Il y a plusieurs solutions qui font usage des bases de données (voir section 3.3). Toutes ces solutions proposées cherchent à créer une base de données capables de stocker des modèles préalablement établis (généralement sur la base de modèles publiés dans la littérature scientifique). Mis à part la contribution de [KAL+17], qui cherchent à rendre automatique ou au moins semi-automatique, le travail de décomposition d'un modèle

complexe en ses sous-modèles constitutif, ainsi que la contribution de [PWB+11], qui propose une méthode de modélisation comparée, pour trouver des structures 3D de protéines.

La base de modèles que nous voulons proposer suit les mêmes objectifs que les travaux cités plus haut et à l'instar de [KAL+17] et [PWB+11], nous cherchons également à tirer avantage de cette base de modèles pour automatiser ou semi-automatiser une partie du travail d'analyse de données expérimentales et de conception de modèles. Dans le cas particulier des séries chronologiques, avec lesquelles nous travaillons, nous voulons associer à la base de données une couche logicielle permettant de faire le lien entre les séries chronologiques obtenues à l'issue d'un processus expérimentales et les modèles théoriques stockées dans la base de données. Nous cherchons donc à développer un système de « requête par les données », qui permettrait à l'utilisateur de réaliser des requêtes sur les modèles, en utilisant des séries chronologiques expérimentales en paramètres.

— D'autres solutions se basent sur un langage de description avec diverses extensions, afin d'adapter son usage aux échanges de formules mathématiques (MathML, OpenMath, voir section 3.4.1). Ces langages sont conçus pour capturer la sémantique des objets mathématiques utilisés, de sorte à ce que ces-derniers puissent être compris par un programme informatique.

  Nous verrons que, quoique pratique pour des formules et pour les applications dédiées au calcul symbolique, nous aurons besoin d'une représentation permettant de faire du calcul numérique. Dans le cadre de notre travail, les modèles échangés (équations) doivent être « solvable numériquement », c'est-à-dire, qu'en plus de la formule, il faut des informations supplémentaires indispensables à l'application d'un algorithme de résolution numérique (voir section 4.1).

— Enfin, il y a des solutions qui se base également sur des langages de description, mais qui sont, cette fois, conçues pour échanger un modèle théorique et l'algorithme qui l'implémente (PMML, PFA, voir section 3.4.2). Ces solutions remplissent donc le besoin énoncé plus haut, dans le sens qu'en plus des modèles théoriques, ils fournissent également une quantité d'information supplémentaires, suffisantes (à quelques éventuels paramètres près) pour utiliser le modèle (suffisantes pour exécuter l'algorithme qui implémente le modèle).

  Cependant, parmi les deux langages présentés (PMML, PFA), aucun ne possède nativement un modèle « équation différentiel ». Dans la section 4.2.4, nous proposons une représentation XML, dont la conception est inspirée du langage PMML. L'objectif étant, à terme, de pouvoir proposer l'intégration d'un modèle d'équation différentielle dans le langage PMML en développant un standard, qui pourra se baser sur la représentation que nous proposons. L'avantage principale du langage PMML, c'est qu'il supporte déjà les séries chronologiques comme type de données, ce qui permet de s'appuyer dessus pour représenter certaines données, notamment décrire la solution d'une équation sous forme de série chronologique (qui est la sortie du modèle), lorsqu'elle est connue (voir section 4.2.4).

Deuxième partie

Nos propositions

Chapitre 4

Modélisation Conceptuelle et Structures de Données Dédiées aux Modèles Mathématiques dans une Approche Entreposage

---

Pour pouvoir manipuler des équations différentielles, de manière pratique, d'un point de vue informatique, nous ne pouvons utiliser une représentation simple sous forme de chaîne de caractères. Mais, les chaînes de caractères ne suffisent pas à décrire la sémantique des symboles utilisés. En plus de la sémantique, la structure doit également permettent de réaliser des calculs numériques, à partir de la formule d'une équation, en prenant en compte les problèmes de priorités opératoires, que peuvent poser une formule mathématique écrite en langage naturel. Cela nécessite donc une structure de données plus complexe que les chaînes de caractères. Dans la suite de ce chapitre, nous décrirons les structures adoptées, pour répondre aux contraintes sus-citées.

Aussi, nous avons vu que la conception d'une base de données, nécessite la définition d'un schéma entité-association. Nous proposons donc un schéma-entité association, qui permette de stocker les équations différentielles, de manière cohérente avec la structure de données choisie pour les représenter. Nous avons également vu que ce schéma doit pouvoir être suffisamment flexible, pour supporter l'ajout de données contextuelles, selon les besoins de l'utilisateur. Nous avons donc basé notre schéma sur le principe de la modélisation dimensionnelle, utilisé dans les entrepôts de données (voir 2.2.2).

Nous souhaitons également que la base de modèles puisse être utilisée en complément d'un environnement numérique pré-établit. En effet, l'utilisateur peut avoir besoin d'extraire les données du modèle, pour réaliser de plus amples analyses. Nous avons donc voulu proposer un format d'échange entre application, par « sérialisation », i.e écriture les données sous forme textuelle, afin de pourvoir les sauvegarder dans des fichiers textes, qui sont alors très facile à échanger, autant en local, que via Internet.

4.1 Équations différentielles : définition et description

Une équation différentielle est une équation qui relie une fonction à une où plusieurs de ses dérivées. Un exemple classique est l'équation suivante :

Exemple 4.1
Soit f fonction de $t \in \mathbb{R}$, telle que :

$$\forall t \in \mathbb{R}, f(t) = f'(t) \tag{4.1}$$

Où f' est la fonction dérivée de f.

Une des solutions de cette équation est la fonction exponentielle : $f(t) = e^t$.

De manière générale, une équation différentielle s'écrit [PM08, Aca07, RDO01] :

$$F(t, y, y', \ldots, y^{(n)}) = 0 \tag{4.2}$$

Où y est la fonction inconnue, y' sa dérivée, dite première, et la notation $y^{(n)}$ permet de désigné la dérivée n-iéme (la dérivée première peut donc s'écrire aussi $y^{(1)}$). Un exemple d'équation différentiel classique en automatique est l'équation suivante :

Exemple 4.2

$$\forall t \in \mathbb{R}, Ty'(t) + y(t) = Ku(t) \tag{4.3}$$

Où :

- y est la fonction inconnue de l'équation, en automatique, elle est définie comme la sortie du système ;
- u est une fonction connue, appelée entrée du système ;
- T et K sont des constantes littérales, nommée respectivement constante de temps et gain.

Remarque Ici, nous avons défini un cas particulier d'équation différentielles, dites ordinaires. Une équation différentielle est ordinaire, lorsque la fonction inconnue y ne dépend que d'une seule variable, notée t ici (j'ai pris la notation t, au lieu du x classique, car en physique, il est commun que la variable soit le temps et soit donc notée t). Lorsque y a plusieurs variables (une fonction qui dépend des trois coordonnées spatiales, par exemple), l'équation fait alors intervenir les dérivées partielles de y, on parle alors d'équation aux dérivées partielles. Pour simplifier l'approche au niveau du traitement numérique (notamment la résolution numérique des équations), nous nous sommes limité aux équations différentielles ordinaires, tel que définies ci-avant.

Les solutions d'une équation différentielles sont définies à une constante prés [PM08, Aca07].

Exemple 4.3
En reprenant l'équation de l'exemple 4.1, nous avons dit que la fonction exponentielle était une solution de l'équation et non pas la solution. En effet, une équation différentielle peut avoir plusieurs solutions et dans le cas présent, il y en as même une infinité (à commencer par la fonction $f(t) = 0, \forall t$). En effet, la fonction $f(t) = 2e^t$, par exemple, est aussi une solution de l'équation. De manière générale, toutes les fonctions de la forme $f(t) = Ce^t, C \in \mathbb{R}$ est solution de l'équation [RDO01, pp. 201-202].

Ainsi, pour désigner une et une seule solution de l'équation, il faut des conditions initiales [PM08, Aca07]. Les conditions initiales consistent à préciser une valeur connue de la fonction. Pour l'équation de l'exemple 4.1, si je fixe par exemple $f(0) = 1$, alors la fonction exponentielle devient la seule et unique fonction, qui soit à la fois solution de l'équation et qui respectent la condition initiale $f(0) = 1$.

Il faut noter aussi, qu'ici la condition initiale est unique : $f(0) = 1$. Cela est dû au fait que l'équation est dite d'ordre un, car elle ne fait intervenir que la dérivée première (appelé aussi dérivée d'ordre un) de la fonction. Mais nous avons vu qu'une équation différentielle pouvait faire intervenir plusieurs dérivées (e.g des dérivées d'ordre supérieures). L'ordre d'une équation est l'ordre maximal des dérivées d'une équation [RDO01]. Par exemple, l'équation $f(t) = f^{(2)}(t)$ est une équation d'ordre deux. Pour une équation différentielle d'ordre deux, fixer uniquement $f(0) = 1$ ne sera pas suffisant.

Exemple 4.4
L'équation différentielle d'ordre deux $f(t) - f^{(2)}(t) = 0$ a pour solution générale : $f(t) = K_1e^t + K_2e^{-t}, K_1, K_2 \in \mathbb{R}$. Fixer $f(0) = 1$, par exemple, donne : $f(0) = K_1e^0 + K_2e^0 = K_1 + K_2 = 1$. Deux inconnues, pour une équation, il nous faut une équation supplémentaire (la théorie de la méthode de résolution est détaillée dans [PM08, pp. 35-36], le détail du calcul est disponible en annexe A).

L'exemple 4.4 nous montre que pour résoudre une équation d'ordre deux, il faut deux équations, pour déterminer les valeurs des constantes. Cela se généralise à n'importe quel ordre. Une équation d'ordre n nécessite n équations pour déterminer les valeurs des n constantes. Il faut donc des conditions initiales supplémentaires. En général, on fixe des valeurs des dérivées d'ordre supérieur. Pour l'équation donnée dans l'exemple 4.4, on pourra ajouter $f'(0) = 0$, par exemple. Cela donne une équation supplémentaire, qui permet de terminer la résolution (dans le cas présent, on trouve $K_1 = K_2 = \frac{1}{2}$, voir annexe A). De manière générale, pour une équation d'ordre n, les conditions initiales sont les valeurs initiales des dérivées d'ordre zéro à $n - 1$ (ce qui fait bien un système de n équations, permettant de définir une solution unique).

En pratique, cela veut dire que pour qu'une équation différentielle soit solvable, il est impératif d'avoir les conditions initiales. En effet, ici « solvable » signifie que l'on doit pouvoir obtenir une solution, pour laquelle on peut calculer des valeurs numériques. Pour $f(t) = 2e^t$, par

exemple, on est capable de dire combien vaut f exactement, pour (théoriquement) n'importe quelle valeur de t. En revanche, il est impossible de calculer des valeurs numériques de $f(t) = Ce^t, C \in \mathbb{R}$, sans connaître la valeur de C et donc, sans avoir de condition initiale, pour déterminer C.

Notons aussi que dans l'équation 4.3, nous avons deux constantes T et K, qui interviennent dans l'équation. La formule donnée est une forme générique d'équation classique en automatique, mais là aussi, pour pouvoir résoudre l'équation (obtenir des valeurs numériques de y), il faut connaître les valeurs numériques de T et K. Enfin, dans l'exemple 4.3, une fonction u intervient. Cette fonction est appelée entrée et n'est pas l'inconnue de l'équation. Cela signifie que pour résoudre l'équation, les valeurs numériques de la fonction u doivent être connues aussi.

Dans la suite de cette thèse, nous appellerons « équations différentielles », la donnée d'une formule littérale et de toutes les données nécessaires pour qu'elle soit solvable. Cet ensemble de données comprend :

- les conditions initiales ;
- les valeurs numériques de toutes les constantes littérales qui interviennent dans l'équation ;
- les valeurs numériques des fonctions d'entrées (toutes les fonctions autres que la fonction inconnue).

Nous appellerons cet ensemble de données « l'ensemble des paramètres de l'équation ». En résumé, nous considérons qu'une équation différentielle est la donnée de sa formule littérale et de ses paramètres.

Remarque Nous considérons également, que si au sein d'une équation, une constante littérale est par définition constante (invariable), il se peut que la constante (son label, tel que T, par exemple) soit réutilisée d'une équation à une autre, mais avec des valeurs différentes (donc variable). En effet, dans l'équation 4.3, les constantes T et K représentent, respectivement, la constante de temps et le gain d'un système physique. Elles ont donc un sens physique (elles décrivent un aspect du système) et peuvent varier d'un système, à un autre et donc, d'une équation à une autre. Nous adopterons ce point de vue et considérerons T et K, comme des variables. Mais, ces-dernières sont dépendantes du système (et donc de l'équation) et non du temps, raison pour laquelle au sein d'une équation (donc pour un système donné), elles sont bien invariables (constantes). Nous les désignerons donc sous le terme de *variables littérales*.

Ainsi, pour représenter une équation différentielle, il faut une structure composée de la formule et l'ensemble des paramètres. Nous rappellerons également que nous travaillons avec des séries chronologiques, que l'on peut décrire mathématiquement comme des suites de nombres indexés par des valeurs temporelles. Une série chronologique pourra être noté comme suit : $(s(t_n))_{n\in\mathbb{N}}$ où s est le nom de la série et l'ensemble des $(t_n)_{n\in\mathbb{N}}$ est un ensemble discret de valeurs réelles distinctes et ordonnées dans l'ordre croissant (chronologique). Pour allégé l'écriture, nous noterons :

- une série $(s(t_n))_{n\in\mathbb{N}}$ par son nom s ;

- un élément $i \in \mathbb{N}$ de s sera noté s_i (on as donc $s_i = s(t_i)$) ;
- et nous noterons n_s, le nombre d'éléments de s.

De plus, si on reprend l'équation 4.3, elle contient deux fonctions y et u. Comme nous travaillons avec des séries chronologiques, y et u ne sont pas décrites par des formules mathématiques ($y(t) = t^2$), par exemple), mais par des séries chronologiques. Aussi, comme l'équation 4.3 est un exemple de modèle fournit par un automaticien du laboratoire, nous pouvons donc décrire complètement l'équation différentielle (formules et paramètres). Tout d'abord, la formules littérales :

$$Ty' + y = Ku \tag{4.4}$$

Puis l'ensemble des paramètres :

- Les conditions initiales : $y_0(= y(t_0)) = 0, t_0 = 0.0s$;
- Les valeurs numériques des variables littérales : $T = 25s$ et $K = 2,4$;
- L'ensemble des valeurs de u : celles-ci sont fournies sous forme de série chronologique, stockées dans un fichier texte au format csv. Elle contient mille valeurs, sur une plage temporelle de 1000s (une valeur par seconde, numérotées de 0 à 999). La suite des $(t_n)_{n\in\mathbb{N}}$ va donc de $t_0 = 0$s à $t_{999} = 999$s avec un écart de 1s entre chaque $t_i, i \in \mathbb{N}$ consécutif.

4.2 Structure de données et ses différents formats de représentation

Dans la section 3.4.1, nous avons vu les langages MathML et OpenMath, qui permettent de représenter des formules mathématiques. Un langage comme OpenMath serait intéressant, car il est conçu pour se concentrer sur la sémantique et la compréhension des concepts mathématiques par les programmes informatiques. Cependant, chacun des langages se concentre sur la représentation de formule mathématiques, mais dans nos travaux, une équation différentielle regroupe une formule et un ensemble de paramètres. Cet ensemble de paramètres, et en particuliers les fonctions d'entrées, ne fait pas parti des fonctionnalités du langage. En revanche, le langage PMML supporte les séries chronologiques depuis sa version 4.0 [Pec09] et le langage contient nativement les outils nécessaires pour représenter des données d'entrées et de sorties d'un modèle (le langage PFA permet aussi la représentation des données d'entrées et sorties, mais n'a pas de support natif pour les séries chronologiques).

4.2.1 Modélisation de l'ensemble des paramètres

Nous allons commencer par décrire les structures de représentation des différents type de données. Les diverses structures décrites ont été implémentées sous forme d'objet en Java. Le code est disponible dans l'annexe C (l'ensemble du projet est téléchargeable sur la forge

du laboratoire [1]). Pour représenter l'ensemble des paramètres, nous aurons besoin de deux structures pré-requises : (i) Les séries chronologiques : elles sont représentées sous formes de liste de couple ($clè : valeur$), que nous pourrions décrire ainsi : $[(t_0 : s_0); \ldots; (t_{n_s-1} : s_{n_s-1})]$ (rappelons que n_s est le nombre d'éléments de s et que l'indexation commence à zéro). Pour alléger cette notation, nous utiliserons la notation, plus compacte, suivante : $[(t_n)_{n\in\mathbb{N}} : s]$. (ii) Les fonctions : il faut pouvoir à la fois distinguer y et u, mais aussi y et y', tout en gardant le lien qui existe entre y et y'. Pour cela, chaque fonction sera attribuée à une clé, qui lui est unique. Cette clé est un couple de valeurs : $(name, deriv)$. Où `name` est le nom de la fonction et `deriv`, son ordre de dérivation. Ainsi y, y' et u ont pour clé, respectivement : $("y", 0)$ indiquant qu'il s'agit de la fonction y non dérivée ; $("y", 1)$ indiquant qu'il s'agit aussi de la fonction y mais dérivée une fois ; et $("u", 0)$ indiquant qu'il s'agit de la fonction u non dérivée. La clé conserve donc le lien entre y et y' tout en précisant tout de même qu'il ne s'agit pas tout à fait des mêmes fonctions. Cet objet est nommé `FunctionKey`, où Clé De Fonction (CDF). Une fois ces structures définies, nous pouvons définir les structures représentants l'ensemble des paramètres.

- Les conditions initiales : en automatique (et en physique, de manière générale), les conditions initiales sont les valeurs de y et ses dérivées éventuelles à l'origine (à t_0 souvent égal à zéro). Nous utiliserons une liste, dont les éléments sont de la forme : $(CDF : valeur)$. Pour l'équation 4.3 les conditions initiales seront représentés par une liste d'un élément : $[(("y", 0) : 0.0)]$ signifiant que la condition initiale de la fonction est $y(t_0) = 0$. Pour l'équation d'ordre deux de l'exemple 4.4, les conditions initiales étant $f(0) = 1$ et $f'(0) = 0$, elles pourront être représentées par la liste suivante : $[(("f", 0) : 1.0); (("f", 1) : 0.0)]$.
- Les valeurs des variables littérales : une variable est représentée par un couple ($clè : valeur$) et l'ensemble des variables est regroupé sous la forme d'une liste de couples ($clè : valeur$). La clé étant le nom de la variable et la valeur, la valeur réelle de la variable, pour l'équation donnée.
- Les valeurs numériques des fonctions : il s'agit d'une liste de séries chronologiques, chacune associée à une fonction, via une CDF. Les valeurs numériques sont donc une liste d'éléments de la forme : $(CDF : sèrie)$.

Exemple 4.5
Pour représenter l'ensemble des paramètres de l'équation 4.3, nous aurons les éléments suivants :

- *Les conditions initiales :* $[(("y", 0) : 0)]$ *;*
- *Les variables littérales :* $[("T" : 25); ("K" : 2, 4)]$ *;*
- *Les valeurs numériques des fonctions :* $[(("u", 0) : [(t_n)_{n\in[\![0,n_u-1]\!]} : u])]$*. Ici, nous notons u la série chronologique des valeurs de la fonction nommée « u ». Aussi, la série u possédant un nombre n_u d'élément, l'ensemble de ces indices n se situent dans l'intervalle d'entiers $[\![0,\ n_u\ -\ 1]\!]$.*

Remarque La série u et la fonction u sont notée de la même manière, bien qu'il s'agisse d'objets mathématiques distincts. Nous nous permettons, en effet, de confondre les notations

1. `https://forge.lias-lab.fr/projects/mathmouse/files`

des fonctions et des séries chronologiques, sachant toutes les fonctions (non-usuelles) sont (dans le cadre de cette thèse) toujours définies par une série chronologique.

4.2.2 Modéliser une formule littérale par un arbre binaire

En 1924, le logicien Jan Lukasiewicz a introduit la notation polonaise (dite polonaise, car Jan Lukasiewicz était polonais) [RMA15, Wal70, McI18]. Cette notation consistait à écrire les opérations en écrivant l'opérateur, puis les termes, contrairement à la notation classique (aussi appelée notation infixe [RMA15, Wal70]), qui place les termes de par et d'autre de l'opérateur.

Exemple 4.6
L'opération $7 + 2$ *(notation classique, ou infixe), se note* $+\ 7\ 2$*, en notation polonaise.*

La notation polonaise inverse (NPI), quant à elle, a été introduite en 1954 et nommée ainsi, car elle consiste a noté l'opérateur après les termes, contrairement à la notation polonaise. Cette notation est aussi appelée notation post-fixe [RMA15, Wal70].

Exemple 4.7
L'opération $7 + 2$*, se note donc* $7\ 2\ +$*, en notation polonaise inverse.*

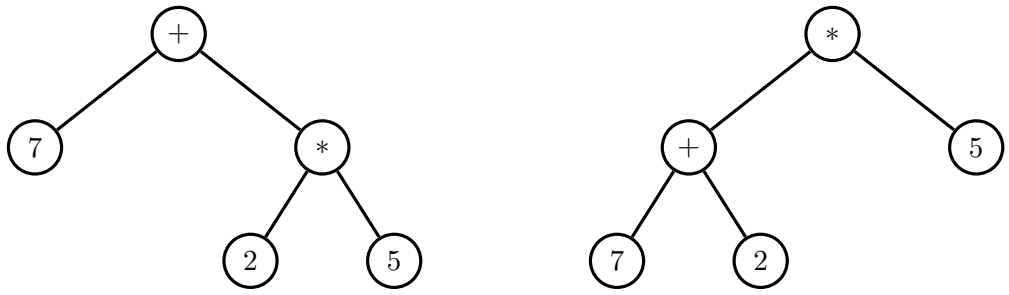
Elle a été réintroduite (réinventée sans le savoir) par Bauer et Dijkstra, en 1960. Ils ont introduit également un algorithme nommé « shunting yard algorithm » (littéralement l'algorithme de la gare de triage, que l'on pourrait traduire par l'algorithme d'aiguillage), permettant de traduire une expression classique (infixe) en un équivalent en polonaise inverse (post-fixe) Le but était alors d'optimiser le traitement des opérations, par les ordinateurs. En effet, cette notation offre l'avantage de ne pas nécessiter de parenthèses, qui servent à indiquer les exceptions aux priorités opératoires.

Exemple 4.8
Prenons les calculs suivant, comme exemple :

$$7 + 2 * 5 \tag{4.5}$$

$$(7 + 2) * 5 \tag{4.6}$$

La priorité opératoire revient à la multiplication dans 4.5 et à l'addition dans 4.6. En notation polonaise inverse, ces calculs se notent, respectivement, $7\ 2\ 5\ *\ +$ *et* $5\ 7\ 2\ +\ *$*. On remarque que la multiplication apparaît avant l'addition, dans l'ordre des opérateurs, pour le premier calcul, où elle est prioritaire. À l'inverse, elle apparaît en second, dans le second calcul, où l'addition lui est prioritaire. L'ordre des opérateurs suit donc toujours l'ordre de leur priorité, sans exception possible. Il n'est donc pas utile d'avoir recourt à des parenthèses. De plus, la régularité de lecture de la notation, la rende beaucoup plus simple à automatiser dans un programme informatique. C'est la notation utilisée par certaine calculatrice, qui convertissent*



(a) Exemple d'arbre pour le calcul 4.5 (b) Exemple d'arbre pour le calcul 4.6

FIGURE 4.1 – Exemple d'arbre de représentant les NPI de 4.5 et 4.6

la notation classique en polonaise inverse [McI18].

L'interprétation par un programme est assez simple et pourtant très efficace, d'où son succès en informatique [RMA15]. Il suffit de lire les éléments de gauche à droite, un à un [Wal70, Chr14, McI18]. On mémorise les termes que l'on rencontre et l'on effectue l'opération, lorsque l'on rencontre l'opérateur.

Exemple 4.9
Pour 7 2 5 $*$ + *: (i) on lit 7, on le sauvegarde dans une liste [7] ; (ii) on lit 2, on le sauvegarde dans notre liste (les nouveaux éléments sont ajoutés en première position dans la liste) [2,7] ; (iii) on lit 5, on ajoute à notre liste [5,2,7] ; (iv) on lit l'opérateur de multiplication, on multiplie les deux derniers termes lus (les deux premiers éléments de la liste) en les retirant de la liste, qui deviens [7] ; (v) le résultat de l'opération précédente donne* $5 * 2 = 10$*, que l'on ajoute à la liste [10,7] ; (vi) puis on lit l'opérateur plus, on additionne donc les deux derniers éléments de la liste :* $10 + 7 = 17$.

Remarque La notation polonaise inverse n'est pas unique. En effet, les calculs 4.5 et 4.6 peuvent aussi s'écrire, respectivement : 2 5 $*$ 7 + et 7 2 + 5 $*$. Cependant, quelle que soit la forme de la notation, l'ordre des opérateurs (et donc les priorités opératoires) est toujours respecté. Ainsi, l'application de l'algorithme de lecture sur l'une ou l'autre des notations possible d'un même calcul, donnera le même résultat.

Si cette représentation se prête bien à une structure de tableau linéaire, les termes infixe et post-fixe sont à rapprocher des algorithmes de parcours d'arbre binaire. Le professeur Brailsford de l'université de Nottingham a fais une vidéo [Bra14] sur la représentation des calculs sous forme d'arbre binaire et comment retrouver la notation polonaise inverse, depuis un arbre. La figure 4.1 est un exemple de représentation des calculs 4.5 et 4.6 sous forme d'arbre binaire. Pour retrouver la NPI de chacune des expressions, il faut lire l'arbre selon le principe du parcours en profondeur post-fixe (postfix deep first search). L'idée de cet algorithme, c'est, à partir d'un nœud, de toujours chercher à inspecter d'abord son sous arbre gauche, puis le sous-arbre droit, avant de traiter le nœud en question. L'algorithme se

base sur l'hypothèse, que si un nœud n'a qu'un enfant, il s'agit de son fils gauche. Donc un nœud qui n'a pas de fils gauche, n'a pas de fils droit non plus. Cependant, un nœud ayant un fils gauche, n'a pas forcément de fils droit. Appliquons cet algorithme à l'arbre de la figure 4.1b :

- On part de la racine, ici le nœud multiplication et il a un fils gauche, le nœud plus ;
- on s'intéresse donc au nœud plus, il possède également un fils gauche, le nœud 7 ;
- le nœud 7 n'a pas d'enfant, comme on ne peut effectuer de calcul pour l'instant, on le garde en mémoire ;
- on a fini de traiter le nœud 7, on passe au traitement du fils droit, le nœud 2, que l'on garde aussi en mémoire ;
- on a fini de traiter les deux enfants, on peut traiter le nœud en cours, le nœud plus et comme on a deux termes (7 et 2) en mémoire, on peut les additionner ce qui donne 9, que l'on garde en mémoire ;
- on a fini de traiter le nœud plus, on peut revenir au nœud multiplication et s'intéresser à son fils droit, le nœud 5 ;
- le nœud 5 n'a pas d'enfant, on peut garder en mémoire le terme 5 ;
- on a fini de traiter le nœud 5, on peut revenir au nœud multiplication et comme on a 5 et 9 en mémoire, on peut les multiplier, ce qui donne 45.

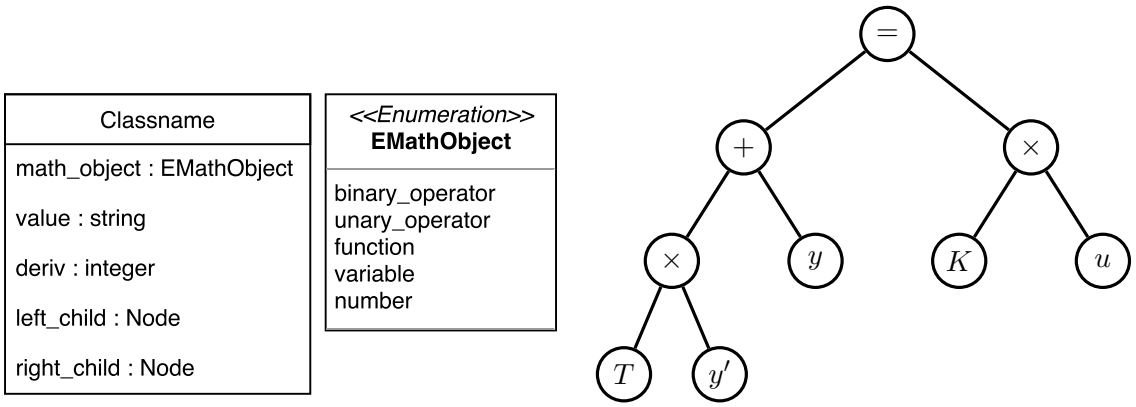
On peut vérifier que le résultat du calcul 4.6 est bien 45, nous ne sommes pas trompé. L'algorithme 1 est le pseudo-code de l'algorithme décrit ci-avant.

Algorithm 1 fonction lire_arbre_postfixe(Nœud nœud)

```
if nœud a un fils gauche then
  lire_arbre(nœud.fils_gauche)
  if nœud a un fils droit then
    lire_arbre(nœud.fils_droit)
  end if
end if
traiter(nœud)
```

Remarque Un algorithme de lecture infixe (qui consiste à lire le nœud gauche, puis à traiter le nœud en cours, avant de lire le nœud droit) permet de retrouver la notation infixe (aux parenthèses près).

Nous nous sommes inspiré de la notation polonaise inverse et de sa représentation sous forme d'arbre pour représenter des équations différentielles. En effet, la NPI n'utilise que des nombres et des opérateurs, mais en autorisant d'autres objets mathématiques (variables littérales et fonctions), il est possible de représenter la formule littérale d'une équation différentielles. La figure 4.2 détaille la structure d'un nœud dans la sous-figure 4.2a et donne un exemple d'arbre représentant l'équation 4.3 dans la sous-figure 4.2b.



(a) Description d'un nœud (b) Arbre binaire de l'équation 4.3

FIGURE 4.2 – Description de la structure d'arbre et exemple

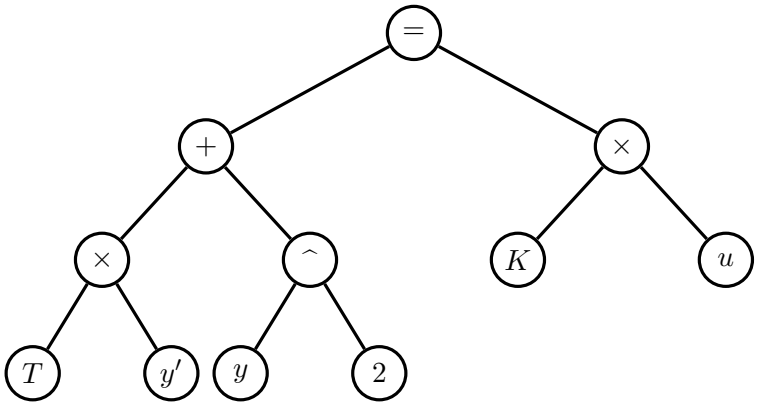


FIGURE 4.3 – Exemple d'arbre non-linéaire

Un nœud de l'arbre peut être d'un des types suivant : opérateur, fonction, nombre, variable. Notons que, comme indiqué sur la figure 4.2a, les opérateurs sont divisés en deux catégories `binary_operator` et `unary_operator` : (i) `binary_operator` (« opérateur binaire ») : les opérateurs qui nécessitent deux termes (l'addition, la multiplication, puissance d'un nombre...) ; (ii) `unary_operator` (« opérateur unaire ») : les opérateurs qui ne s'appliquent qu'à un terme (opposé d'un nombre, factorielle, racine carré...). Un nœud contient donc une variable `EMathObject` indiquant son type d'objet mathématique. Un attribut `valeur`, qui est la valeur du nœud (le nom de la variable, de l'opérateur...). Un attribut `Deriv`, qui, pour un objet de type fonction, indique son ordre de dérivation, sinon il vaut zéro par défaut. Puis les attributs `left_child` et `right_child`, qui indiquent les enfants du nœud. Ainsi, il est possible de construire une structure d'arbre, comme sur la figure 4.2b. Il s'agit de la représentation sous forme d'arbre binaire de l'équation 4.3. De même que pour les (NPI), une lecture de l'arbre sous forme infixe permet de retrouver (aux parenthèses près) la forme infixe de l'équation, une lecture postfixe permets d'interpréter l'arbre sous sa forme polonaise inverse.

Remarque La représentation sous forme d'arbre décrite ici permet de représenter des équations différentielles linéaires et non-linéaire. Par exemple, l'arbre donné sur la figure 4.3 n'est pas linéaire. En effet, sous forme infixe, l'équation s'écrit : $Ty' + y^2 = Ku$ et l'on remarque que le terme y est élevé au carré (l'opérateur puissance est indiqué par le symbole $\hat{}$ dans l'arbre). Nous noterons cependant que : (i) dans la suite des travaux, nous ne considérons, que des équations différentielles linéaires, pour simplifier l'implémentation des algorithmes de résolution des équations, le cas linéaire étant le plus simple ; (ii) la structure ne permets pas le stockage d'équations aux dérivées partielles, car elle a été construite en supposant que les fonctions utilisées sont des fonctions d'une seule variable, donc, si une fonction possède plusieurs variables, il ne sera pas possible (à moins de modifier la structure actuelle) de préciser selon quelle variable elle doit être dérivée, le cas échéant.

4.2.3 Diagramme E/A

Nous avons vu (section 3.2.1) que les bases de données relationnelles n'étaient pas des plus adaptées au stockage de séries chronologiques. En revanche, les équations différentielles se prêtent plus facilement à un stockage relationnel. En effet, si les BDR ne conviennent pas aux séries chronologiques, c'est entre autre pour le fait que le langage SQL ne possède pas un certains nombres de fonctionnalités spécifiques et nécessaires à la manipulation et à l'analyse de séries chronologiques. À l'inverse, une équation différentielle ne contiens pas de données séquentielles et se prêtent donc plus facilement au stockage dans une base de données relationnelle. Notons tout de même qu'il s'agit d'un avantage théorique, qui n'est pas toujours vrai en pratique. En effet, cet avantage est limité, dans notre cas, par la méthode de stockage des fonctions d'entrées. Dans notre cas, nous les représentons sous formes de séries chronologiques, car c'est ainsi qu'elles nous ont été fournies pour les tests et cela a l'avantage d'être une représentation « simple » dans le sens où il s'agit d'un tableau de valeur à stocker. Mais cela annihile l'avantage en termes de volume de stockage, des équations différentielles. Par contre, les fonctions d'entrées sont généralement connues et ne sont pas l'objet des études et des analyses. Il n'y a donc pas d'enjeu en termes d'optimisation du traitement des données.

Remarque Cette possibilité n'a pas été abordé ici, mais il pourrait être possible de s'affranchir de la contrainte du volume de stockage des fonctions d'entrées en les représentant sous forme de formule mathématique. De plus, la (NPI) pourrait être une bonne représentation de ces-dernières, car leurs expressions pourraient être insérées telle quelle dans les arbres des équations, avant de passer aux étapes de calculs numériques.

Un des avantages dont nous voulions tirer parti était le langage SQL. En effet, nous avons voulu conserver la possibilité de réaliser des requêtes « classiques », pour réaliser des opérations de sélection des équations, qui pourront venir compléter le système de « requête par les données » en réalisant une pré-sélection sur les équations, évitant ainsi d'avoir à traiter toutes les équations, mais uniquement un sous-ensemble, que l'utilisateur aura jugé pertinent. Le type de requête sélective pourront être les suivantes :

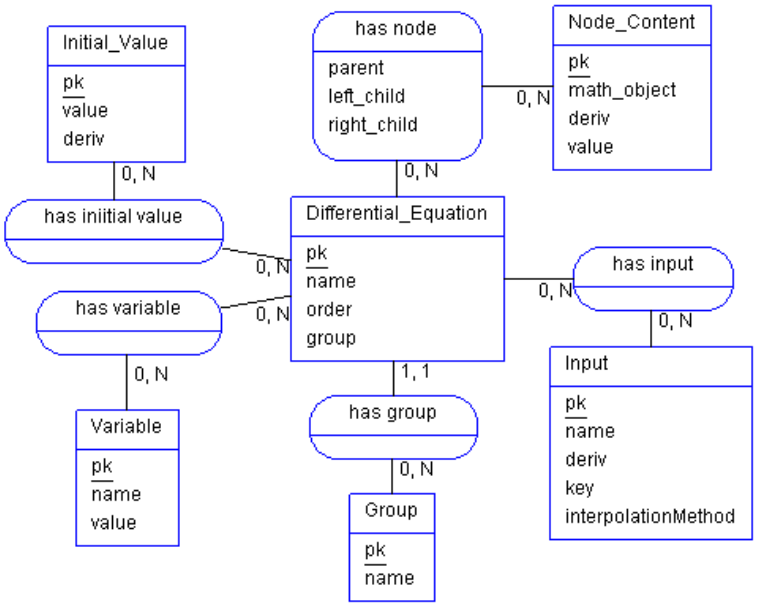


FIGURE 4.4 – Diagramme (E/A) représentant une équation différentielle

Exemple 4.10 *1. Sélectionner des modèles dont une variable en particuliers est située dans un intervalle de valeurs. Par exemple, les équations qui possèdent un paramètre T, dont la valeur est entre* 10 *et* 25 *secondes.*

2. Sélectionner les équations d'un ordre donné.

Le second objectif était de fournir un schéma qui soit facilement extensible, de sorte que l'utilisateur puisse ajouter les informations qu'il souhaite, concernant les équations. Nous avons donc voulu proposer un schéma inspiré de la modélisation dimensionnelle et sa représentation sous forme de schéma en étoile utilisé pour construire les entrepôts donnés à l'aide de bases de données relationnelles (voir section 2.2.3). L'objectif était de fournir un schéma permettant de stocker des équations et un ensemble de données contextuelles liées à ces-dernières. Par exemple, le chercheur/expérimentateur, qui a rentré l'équation dans la base, l'expérience réalisée pour obtenir l'équation, la date d'insertion de l'équation... Tous ceci rappelle le système de dimension dans la modélisation dimensionnelle (voir section 2.2.2). Aussi, l'utilisation d'un schéma en étoiles avec des dimensions permets d'utiliser la notion de hiérarchie dans les dimensions. Ainsi, en prenant par exemple une dimension pour préciser le chercheur qui rentré l'équation, ce dernier peut faire partie d'une équipe de chercheur, elle-même rattachée à un département, définissant ainsi la hiérarchie suivante : département > équipe > chercheur. Il est alors possible de tirer avantage des opérateurs OLAP, tels que Drill-Down ou Roll-Up, pour rapidement visualiser l'activité d'un laboratoire à l'échelle d'un département ou d'une équipe. En incluant la date d'insertion, il est aussi possible d'organiser les équations selon diverses périodes temporelles (mois, semestre, année...).

La figure 4.4 montre le schéma, que nous avons conçu. Nous avons défini une entité centrale (`Differential_Equation`) dont le rôle est similaire à une table de faits (il ne s'agit cependant

pas d'une table de faits et le schéma n'est pas un schéma en étoile, voir le paragraphe remarque, un peu plus bas). Elle est à un rôle central dans le schéma et toutes les associations lui sont reliées, comme des extensions à la table centrale, à la manière des dimensions. Et de même que les dimensions, il n'y a pas de communication entre les différentes extensions de la table centrale. Elle nécessite quatre extensions de base, qui servent à préciser la formule littérale et les différents éléments de l'ensemble des paramètres. Les différentes tables présentes sur le schéma permettent de représentation une équation différentielle (formule + paramètres).

- L'entité `Differential_Equation` joue le rôle d'entité centrale dans le schéma, à l'instar de la table de faits dans un schéma en étoile. Elle liste l'ensemble des équations différentielles (à travers leur clé primaire `pk`, pour « primary key ») stockées dans la base de données. Elle possède un attribut `name`, qui est un nom donné par l'utilisateur, à l'équation. Ce dernier peut servir à indiquer ce que représente l'équation. Elle possède également un attribut `ordre`. Cet attribut est redondant, car il peut être retrouvé à partir de la formule littérale (donc de l'arbre) en cherchant l'ordre de dérivation le plus élevé de la fonction inconnues. Nous avons choisi d'indiquer l'ordre en tant qu'attribut dans la table `Differential_Equation`, car l'extraction en SQL de l'ordre de dérivation depuis la formule est une requête très complexe à écrire et nécessite l'utilisation d'une vue matérialisée.
- L'entité `Node_Content`, représente un nœud de l'arbre tel que défini dans la section 4.2.2. En reprenant l'arbre de la figure 4.2b, nous pouvons remarquer qu'il y a deux nœuds contenant l'opérateur de multiplication. Les nœuds ont le même contenu, la table `Node_Content` ne contiendra qu'un seul tuple représentant un nœud de multiplication, mais ce dernier sera référencé plusieurs fois, pour indiquer qu'il sert à plusieurs endroits de l'arbre.
- L'entité `Initial_Value` contient les conditions initiales (valeurs à t_0 des dérivées de la fonction inconnue de l'ordre zéro à l'ordre $n-1$, voir section 4.1).
- L'entité `Variable` contient les valeurs réelles des variables littérales contenues dans l'équation.
- L'entité `Input` contient les fonctions d'entrées (fonctions connues) de l'équation. Les attributs `name` et `deriv` sont les attributs composants la `FunctionKey` (voir section 4.2.1). L'attribut `key` est une clé automatiquement générée par le programme d'insertion des équations. Les fonctions étant définie par des séries chronologiques, la liste des valeurs n'est pas stockée directement dans les SGBD, car nous avons vu que les bases de données relationnelles classique n'était pas une solution correcte. La quantité de séries que nous utilisons n'étant pas très grande pour le moment, nous pouvons utiliser le stockage dans des fichiers. Ainsi, la clé générée est le nom du fichier dans lequel la série correspondante a été écrite. Cependant, ce système a été créé pour offrir la possibilité d'utiliser un meilleur système de stockage (une base de donnée NoSQL ou un SGSC), sans avoir à modifier le schéma relationnel. Par exemple, la clé pourrait servir à extraire la série chronologique désirée dans magasin de donnée de type clé-valeur (voir section 2.3.1). L'attribut `interpolation_method` sert à indiquer quel type d'interpolation est le mieux adapté à la série chronologique (voir section 5.2.1).
- L'entité `Group` n'est pas nécessaire à la modélisation d'une équation. Il s'agit d'un exemple de dimension (telle que définie dans un schéma en étoile). Les groupes permettent à l'utilisateur de regrouper les équations selon des groupes qu'il définira.

Table	Attributes
Differential_Equation	PK; name; order; #group
Initial_Value	PK; value; deriv_input
Node_Content	PK; math_obj; deriv; value
Variable	PK; name; value
Input	PK; name; deriv; key; interpolation_method
Group	PK; name
has_node	PK; #diff_eq; #node_content; #parent; #left_child; #right_child
has_input	PK; #diff_eq; #input
has_variable	PK; #diff_eq; #variable

TABLE 4.1 – Modèle relationnel (Schéma logique) de la base de modèles

Le tableau 4.1 sont les différentes relations construites par la traduction du schéma entités-associations de la figure 4.4 en un modèle relationnelle. Chaque entité est associée à une table du même nom et possédant les mêmes attributs et quatre relations ont été ajoutées pour implémenter les associations (N :M) :

- `has_node` : fait le lien entre tous les nœuds de l'arbre. Pour un nœud donné, les attributs `parent`, `left_child` et `right_child` sont clés étrangères vers la table `Node_Content` et indiquent, respectivement, le parent et les enfants gauche et droit du nœud en cours. La clé peut être nulle, si le nœud n'a pas de parent (racine) ou d'enfants (feuilles). Cette méthode de stockage d'arbre (de structure récursive, de manière générale) est inspirée du design « Nested Sets » (« Ensembles imbriqués ») de [Kar10, pp. 34-53] et [Cel04, pp. 45-99].
- `has_input`, `has_variable` et `has_initial_value` : les tables d'associations, permettent d'indiquer quels éléments appartiennent à quelle équation.

Remarque sur le schéma de la base de modèle Le schéma de la figure 4.4 n'est pas un schéma en étoile. En effet, dans un schéma en étoile, les associations entre la table de faits et ses dimensions sont dites (1 :N). Par exemple, si le fait étudier sont les ventes d'une entreprise. La date de la vente sera contenue dans une dimension. L'association (1 :N) impose qu'une vente ne soit associée qu'à une seule date (autrement dit, chaque vente est un évènement unique localisé dans le temps). En revanche, plusieurs ventes peuvent avoir été réalisées à la même date. Or, sur le schéma de la figure 4.4, les associations entre l'entité `Differential_Equation` et ses extensions sont de type (N :M), car une équation peut avoir plusieurs conditions initiales, plusieurs fonctions d'entrées, plusieurs variables littérales et l'arbre représentant sa formule est composée de plusieurs nœuds. Et une même fonction d'entrées, peut servir à plusieurs équations différentes. Les nœuds représentant les opérateurs vont généralement servir plusieurs fois dans une équation et vont généralement servir aussi dans plusieurs équations. Non seulement, les extensions de l'entité `Differential_Equation`, qui apparaissent sur le schéma 4.4, ne sont pas des dimensions, mais les données qu'elles contiennent ne sont pas contextuelles et leurs présences sont nécessaires. En effet, elles servent à représenter les différents éléments qui définissent une équation (formule + paramètres). Cependant, même s'il n'était pas possible d'avoir un véritable schéma en étoile, nous avons tout de même choisi une structure qui en conserve au maximum les propriétés. En définissant une entité centrale,

qui peut admettre des extensions indépendantes entre elles. La seule exception est l'entité `Group`, qui possède bien une liaison (1 :N) avec l'entité centrale. Il est donc toujours possible d'ajouter des dimensions, qui respectent les propriétés d'une dimension d'un modèle multidimensionnelle, afin d'ajouter des données (informations) supplémentaires.

Nous allons détailler les tuples nécessaires, afin de stocker l'équation 4.3, selon le schéma de la figure 4.4.

- La table 4.2 est la table centrale. Elle défini une clé primaire, qui permettra de désigner l'équation 4.3. Ici, la clé vaut 1. L'équation possède un nom (« équation 1 ») et un groupe (« first order », ici, l'exemple n'a pas d'intérêt, puisque les équations du premier ordre peuvent être déterminée par l'attribut `order`) donné par l'utilisateur. L'attribut `order` indique l'ordre de l'équation.
- La table 4.3 contiens les différents nœuds utilisés dans l'arbre de l'équation 4.3. Les nœuds qui apparaissent plusieurs fois dans l'arbre (le nœud multiplication, dans ce cas) n'apparaissent qu'une fois dans la table, pour éviter les redondances.
- La table 4.4 permet d'indiquer l'agencement des nœuds dans l'arbre. Pour une équation, la racine est le seul nœud à ne pas avoir de parent, ici le tuple 1. L'attribut `content` en bout de ligne, indique le contenu du nœud. Ici, l'attribut vaut 1, il faut donc lire le tuple de clé primaire 1 dans la table `Node_Content`. On y voit, que le nœud racine est l'opérateur binaire d'égalité. Les attributs `#left_child` et `#right_child` indiquent que pour connaître les enfants de la racine, il faut lire les lignes dont les clés primaires sont, respectivement, 2 et 3. En lisant la ligne dont la clé primaire vaut deux, on obtient un tuple dont l'attribut `parent` indique bien 1 (la racine, ici). Aussi, le nœud a lui-même deux enfants (4 et 5) et sont contenu est l'élément de clé primaire 2 dans la table `Node_Content`. En consultant la table `Node_Content`, on peut voir qu'il s'agit de l'opérateur binaire d'addition. En procédant ainsi, on peut récupérer l'ensemble des nœuds de l'arbre, ainsi que la structure de ce-dernier.
- La table 4.5 contient les conditions initiales de l'équation 4.3. Ici, il n'y a qu'une condition initiale. L'attribut `deriv` indique qu'il s'agit de la valeur initiale de la fonction inconnue (y) et que la valeur est 0.0.
- La table 4.6 contient les valeurs numériques des variables T et K, qui interviennent dans l'équation 4.3. On y lit donc que T vaut 25 et que K vaut 2.4. Ainsi, si la variable K, par exemple, est utilisée dans plusieurs équations, avec plusieurs valeurs différentes, elle apparaîtra autant de fois, qu'elle a de valeurs différentes.
- La table 4.7 permet d'indiquer l'ensemble des variables de la table `Variable`, qui appartiennent à l'équation. Ici, la première ligne indique que l'équation 1 possède la variable 1 de la table `Variable`, qui est T, la seconde ligne, indique que la variable 2 (donc K) appartient aussi à l'équation. Ainsi, si la variable K, par exemple, intervient dans plusieurs équations, mais avec la même valeur, il suffit d'une seule ligne dans la table `Variable`, mais une ligne par équation dans `has_variable`.
- La table 4.8 contient la fonction d'entrée u de l'équation 4.3. Les attributs `name` et `deriv` constituent la CDF de u. L'attribut `key` est automatiquement générée de manière unique, au moment de l'insertion de la fonction dans la base de donnée. L'attribut `interpolation_method`

Differential_Equation			
pk	name	order	group
1	« équation 1 »	1	« first order »

TABLE 4.2 – `Differential_Equation` table pour l'équation 4.3

Node_Content			
pk	math_object	value	deriv
1	binary_operator	$=$	null
2	binary_operator	$+$	null
3	binary_operator	$\times$	null
4	function	y	0
5	variable	K	null
6	function	u	0
7	variable	T	null
8	function	y	1

TABLE 4.3 – `Node_Content` table pour l'équation 4.3

vaut « none », ce qui indique qu'il n'y a pas de méthode d'interpolation associée à cette fonction (voir section 5.2.2, pour plus de détails).

Écriture des requêtes sélectives de l'exemple 4.10

Exemple 4.11 *1. Sélectionner des modèles dont une variable en particuliers est située dans un intervalle de valeurs. Par exemple, les équations qui possèdent un paramètre T, dont la valeur est entre 10 et 25 secondes :*

```
SELECT Variable.name, Variable.value
FROM Variable, has_variable, Differential_Equation
WHERE Variable.pk = has_variable.variable
AND Differential_Equation.pk = has_variable.diff_eq
```

has_node				
pk	parent	#left_child	#right_child	content
1	null	2	3	1
2	1	4	5	2
3	1	6	7	3
4	2	8	9	3
5	2	null	null	4
6	3	null	null	5
7	3	null	null	6
8	4	null	null	7
9	4	null	null	8

TABLE 4.4 – `has_node` table pour l'équation 4.3

Initial_Value		
pk	value	deriv
1	0.0	0

TABLE 4.5 – `Initial_Value` table pour l'équation 4.3

Variable		
pk	name	value
1	T	25
2	K	2.4

TABLE 4.6 – `Variable` table pour l'équation 4.3

has_variable		
pk	#diff_eq	#variable
1	1	1
2	1	2

TABLE 4.7 – `has_variable` table pour l'équation 4.3

```
AND Variable.name = 'T'
AND Variable.value > 10
AND Variable.value < 25
```

2. *Sélectionner les équations d'un ordre donné (l'utilisation redondante de l'attribut* `order` *dans la table* `Differential_Equation`*, permet d'écrire très simplement cette requête) :*

```
SELECT * FROM Differential_Equation WHERE order = n
```

Sans cet attribut, extraire l'ordre depuis la formule nécessite (en SQL) :

(i) d'identifier les nœuds contenant la fonction inconnue : il faut lister les différentes fonctions qui interviennent dans la formule et y enlever toutes les fonctions connues, stockée dans la table `Input` *;*

(ii) de chercher l'ordre de dérivation maximale : en isolant les nœuds de l'équation qui contiennent la fonction inconnue, pour récupérer la valeur maximale des attributs `deriv` *;*

(iii) et enfin de sélectionner ou non l'équation, en fonction de l'ordre obtenue.

Il est possible de réaliser cette requête, à l'aide de tables intermédiaires (vues), pour réaliser les étapes une et deux. En stockant l'ordre directement dans la table `Differential_Equation`*, on économise le coût de création des vues, pour réaliser directement la sélection, rendant la requête à la fois plus simple à écrire pour l'utilisateur et plus efficiente.*

Input				
pk	name	deriv	key	interpolation_method
1	u	0	autogen_key	none

TABLE 4.8 – `Input` table pour l'équation 4.3

4.2.4 Représentation XML et définition d'un schéma XML

Nous avons cherché à proposer une représentation textuelle des équations différentielles. Le but étant de fournir un moyen d'échange entre la base de données et des outils/applications externes existantes. Cela pourrait permettre à un utilisateur d'extraire les données de la base de modèles pour les charger dans un outil/logiciel de calcul/analyse numérique (Matlab, R, Python avec diverses librairies de calcul numériques...) auquel il est habitué. Cela peut surtout permettre l'opération inverse, après avoir extrait un modèle à l'aide d'un outil/logiciel d'analyse numérique classique, il peut l'exporter dans un fichier texte, afin de l'ajouter à la base de modèles. Les fichiers XML peuvent également permettre d'automatiser la communication entre le logiciel gérant la base de modèles et d'autres briques logicielles, qu'elles soient intégrées (fournie avec) ou externes au gestionnaire de la base de modèles. Le langage XML (eXtended Markup Language) est un standard utilisé entre autre pour la transmission de données entre application [Car15].

Un avantage du langage XML est la possibilité de personnaliser le langage, permettant de l'adapter à un besoin particulier. Pour ce faire, il faut définir un schéma XML, qui permet de définir la structure XML désirée. La définition d'un schéma XML, permet également de vérifier la conformité d'un fichier XML au schéma. Ce qui permet aux programmes écrivains de validée le contenu du futur fichier avant de l'écrire et aux programmes lecteurs de vérifier la conformité du fichier avant d'essayer d'en extraire les informations. Nous avons défini un schéma XML (fourni en annexe B) et la figure 4.5 est un exemple de document XML, respectant la structure que nous avons définie. Dans l'exemple donné, c'est l'équation 4.3, qui est représentée.

Pour la représentation d'une formule (balise `formula`), nous tirons avantage de la possibilité d'imbriquer les balises, pour représenter à la fois les nœuds de l'arbre de la formule et sa structure. Dans l'exemple de la figure 4.5, la première balise est une balise `binary-oprator`. Cette-dernière possède un attribut `value`, qui vaut « = ». Cette balise représente donc l'opérateur d'égalité de l'équation. À l'intérieur de la balise, une seconde balise `binary-operator`, possède cette fois le signe « + » comme valeur et représente l'opérateur d'addition, qui est l'enfant gauche de l'opérateur « = » (voir figure 4.2b). Nous décrivons les différentes balises dans les paragraphes qui suivent.

- `differential-equation` : le schéma XML de cette balise (voir figure 4.6) oblige cette balise à ne contenir qu'une (et une seule) balise `formula` et une (et une seule) balise `parameters-sets` (voir plus loin). Elle contient également les attributs `name` et `group`, qui caractérisent l'équation. Le schéma XML précise que l'attribut `name` est obligatoire, dû à l'attribut `use` égale à « required », contrairement à l'attribut `group`, qui, si aucune valeur n'est fournie, sera égal à « Default », par défaut.
- `formula` : le XML schéma (il n'est pas fourni directement dans le texte, mais le fichier complet est disponible dans l'annexe B) oblige cette balise à contenir trois attributs : `numberOfInputFunctions` qui doit être un entier et représente le nombre de fonctions d'entrées que possède l'équation (1 pour l'équation 4.3) ; `numberOfVariables` également un entier le nombre de variables littérales contenues dans l'équation (2, pour l'équation 4.3)

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<differential-equation name="equation1" group="First Order">
<formula numberOfInputFunctions="1" numberOfVariables="2" outputName="y">
  <binary-operator value="=">
    <binary-operator value="+">
      <binary-operator value="*">
        <variable value="T"/>
        <function deriv="1" value="y"/>
      </binary-operator>
      <function deriv="0" value="y"/>
    </binary-operator>
    <binary-operator value="*">
      <variable value="K"/>
      <function deriv="0" value="u"/>
    </binary-operator>
  </binary-operator>
</formula>
<parameters-sets numberOfParametersSets="1">
  <parameters-set>
    <initial-values numberOfInitialValues="1">
      <initial-value deriv="0" name="y" value="0.0"/>
    </initial-values>
    <variables-values numberOfVariables="2">
      <variable-value name="T" value="25.0"/>
      <variable-value name="K" value="2.4"/>
    </variables-values>
    <input-functions numberOfInputFunctions="1">
      <input-function deriv="0" name="u">
        <time-series interpolationMethod="none">
          <time-value time="0.0" value="1.0"/>
          <time-value time="1.0" value="1.0"/>
          ...
          <time-value time="998.0" value="-1.0"/>
          <time-value time="999.0" value="-1.0"/>
        </time-series>
      </input-function>
    </input-functions>
  </parameters-set>
</parameters-sets>
</differential-equation>
```

FIGURE 4.5 – Représentation XML de l'équation 4.3

et `outputName` qui est le nom de la fonction inconnue (y pour l'équation 4.3).

- `binary-operator` : il existe aussi une balise `unary-operator`. Ces deux types de balises permettent de représenter les opérateurs. Le signe de l'opérateur étant stocké dans l'attribut `value` de la balise. Le schéma XML de la balise `binary-operator` (voir figure 4.7) lui impose de contenir exactement deux sous-balises. En effet, les attributs `minOccurs` et `maxOccurs` tous les deux égaux à deux imposent qu'il y ait au minimum deux sous-balises, mais également au maximum deux sous-balises (dans le cas de la balise `differential-equation` plus haut, ces deux attributs ne sont pas précisés, mais sont par défaut égaux à un). Le schéma XML permet aussi de restreindre les valeurs permises dans l'attribut `value` aux différents signes des opérateurs binaires et permet aussi de rendre l'attribut obligatoire. La figure 4.8 montre la définition du type `binaryOperatorEnum`, qui restreint le type `xs:token` (chaîne de caractères) aux différents caractères énumérés. La balise `unary-operator` est, elle, obligée d'avoir une et une seule balise et les possède aussi un attribut `value`, dont les valeurs sont restreintes à l'ensemble des signes des opérateurs unaires.
- `variable` : représente une variable, dont le nom est stocké dans l'attribut `value`. Le schéma XML de la balise (disponible dans le fichier complet en annexe B), l'attribut `value` est obligatoire et doit être une chaîne de caractères.
- `nombre` : n'apparaît pas sur l'exemple, car l'équation 4.3 ne contient pas de nombre réel fixé dans sa formule. Une équation telle que : $2y' + y = 0$ fera appelle à deux balise `nombre` pour représenter 2 et 0.
- `function` : représente une fonction, avec un nom et un ordre de dérivation contenu dans les attributs `name` et `deriv` (les mêmes attributs qui définissent la CDF, voir section 4.2.1). Le schéma XML rend obligatoire les deux attributs et impose également, que la valeur de l'attribut `deriv` soit entière.
- `parameters-set` : Elle contient toutes les informations sur l'ensemble des paramètres de l'équation. le XML schéma oblige la balise à être incluse dans une balise `parameters-sets`. Bien que ça ne soit pas encore supporter dans le code et dans la base de données, la structure a été pensée pour qu'à l'avenir, il soit possible d'avoir plusieurs jeux de paramètres, associées à une formule. Ainsi, une formule qui serait associée à plusieurs jeux de paramètres, définira autant d'équations, qu'il n'y a de jeux de paramètres.
- `initial-value` : le schéma XML lui impose d'être toujours incluse dans une balise `initial-values`. Les attributs `value` et `deriv` sont obligatoires et l'attribut `deriv` doit être un entier naturel, tandis que l'attribut `value` doit être une valeur réelle.
- `variable-value` : le schéma XML lui impose d'être toujours incluse dans une balise `variable-value`. L'attribut `value` est obligatoire et doit être une valeur réelle.
- `input-function` : le schéma XML lui impose d'être toujours incluse dans une balise `input-functions`. Les attributs `name` et `deriv` sont obligatoires et l'attribut `deriv` doit être un entier. Les valeurs de la fonction (sa série chronologique) sont inscrites dans la sous-balise `time-series`. Cette-dernière est obligatoire et doit être unique (une par balise `input-function`).
- `time-series` : la construction de cette balise est très fortement inspirée du langage PMML. PMML (pour Predictive Model Markup Language) est un standard lui aussi basé sur le langage XML [Pec09, GZLW09]. Selon [PBG16], il s'agit du standard d'échange le plus répandu, dans le domaine de la fouille de données. En effet, il a été développé pour permettre

```
<xs:complexType name="differentialEquationType">
  <xs:sequence>
    <xs:element ref="formula" />
    <xs:element ref="parameters-sets" />
  </xs:sequence>
  <xs:attribute name="name" type="xs:NMTOKEN" use="required" />
  <xs:attribute name="group" type="xs:token" use="optional" default="Default" />
</xs:complexType>
```

FIGURE 4.6 – Schéma XML de la balise `differential-equation`

```
<xs:complexType name="binaryOperatorType">
  <xs:sequence>
    <xs:choice minOccurs="2" maxOccurs="2">
      <xs:element ref="binary-operator"></xs:element>
      <xs:element ref="unary-operator"></xs:element>
      <xs:element ref="function"></xs:element>
      <xs:element ref="number"></xs:element>
      <xs:element ref="variable"></xs:element>
    </xs:choice>
  </xs:sequence>
  <xs:attribute name="value" type="binaryOperatorEnum" />
</xs:complexType>
```

FIGURE 4.7 – Schéma XML de la balise `binary-operator`

l'échange de modèles prédictifs utilisée en fouille données. Il permet également de modéliser les donner d'entrées des modèles, un des types de données d'entrées disponibles étant les séries chronologiques. La balise contient trois attributs, dont deux sont optionnels `startTime` et `endTime` et un est obligatoire `interpolationMethod`. Cette dernière permet d'indiquer la méthode d'interpolation adaptée à la série (voir fin de la section 5.2.1). Enfin, la balise peut contenir une liste de sous-balise de type `time-value`. Le schéma XML la présence d'au moins une balise `time-value`, mais n'impose pas de limite sur le nombre maximum. Il peut donc (en théorie, bien entendu) y avoir une infinité de valeurs.

```
<xs:simpleType name="binaryOperatorEnum">
  <xs:restriction base="xs:token">
    <xs:enumeration value="-" />
    <xs:enumeration value="+" />
    <xs:enumeration value="*" />
    <xs:enumeration value="/" />
    <xs:enumeration value="=" />
    <xs:enumeration value="&radic;" />
    <xs:enumeration value="^" />
  </xs:restriction>
</xs:simpleType>
```

FIGURE 4.8 – Restriction par le schéma XML des valeurs d'un attribut

4.3 Conclusion

Ce chapitre a permis de définir les structures de données nécessaires à la base de modèles. Tout d'abord, nous avons vu qu'une équation différentielle, pour être exploitable (solvable numériquement) dans le cas de la base de modèles, doit être définie à l'aide d'une formule explicite (à l'instar de l'équation 4.3) et d'un ensemble de paramètres. La formule explicite est vue comme un arbre binaire. Cet arbre peut être interprété de manière infixe, de sorte à retrouver la formule sous sa forme usuelle, soit de manière post-fixe, qui permet de l'écrire en notation polonaise inverse (NPI), cette-dernière étant plus simple à exploiter pour les calculs par ordinateur. Les nœuds dudit arbre sont les divers opérateurs, tandis que les feuilles sont les termes, qui peuvent être des fonctions, des variables et des nombres réelles. Quant à l'ensemble des paramètres, il contient les éléments suivants :

- les conditions initiales (la liste des valeurs initiales des fonctions dérivées, sauf la dérivée d'ordre n où n est l'ordre de l'équation) ;
- les valeurs réelles des variables ;
- les valeurs des fonctions d'entrées (sous formes de séries chronologiques associées à une méthode d'interpolation appropriées).

En plus de l'implémentation en Java, de la structure d'équation différentielle et dans le but de permettre le stockage dans une base de données relationnelle, nous avons également proposé un schéma entité/association, qui permet de stocker la structure d'équation différentielle définie ci-avant. Le schéma a été conçu selon une philosophie proche de celle du schéma en étoile d'un entrepôt de données, dans le but d'obtenir un schéma facilement extensible, pour permettre le stockage d'information supplémentaire, si les utilisateurs finaux en éprouvent le besoin. Enfin, pour faciliter l'échange de données entre la base de modèles et des applications externes, nous avons proposé un formalisme XML définie à l'aide d'un schéma XML (le fichier xsd du schéma est en annexe B). Maintenant que nous avons la structure de données et ses différentes représentations, la prochaine partie s'attachera à définir les différents programmes nécessaires à leur exploitation, afin de poursuivre le développement de la base de modèles.

Chapitre 5

Exploitation des Modèles Mathématiques

Le chapitre précédent s'est concentré sur la définition de la structure de données et ses diverses représentations nécessaires à la manipulation d'équations différentielles. L'objectif de ce chapitre sera donc de définir les processus nécessaires pour réaliser les fonctionnalités principales de la base de modèles :

- insérer/récupérer des équations dans la base de modèles ;
- effectuer des *requêtes par les données*, i.e retrouver une équation, à partir d'une série chronologique contenant des mesures expérimentales, prises par un utilisateur/expérimentateur.

Les fonctionnalités principales sus-mentionnées se découpent en plusieurs sous-processus. (i) Pour l'insertion/récupération de données, il faut pouvoir lire une équation depuis une source externe (ici, des fichiers XML) et transformer les données, pour les mettre au format définit par le schéma (E/A) de la base de modèles. Il faut également pouvoir réaliser l'opération inverse. On parlera de conversion XML-Tuple (et inversement). Les méthodes de conversion serviront ensuite à définir les processus d'insertion et d'extraction, qui permettent le chargement de données dans la base de modèles (insertion) ou la récupération de données de cette dernière (extraction). (ii) Pour effectuer des *requêtes par les données*, il faut :

- construire une structure d'équation différentielle en Java, à partir des données, sous forme de tuples extraites de la base de données ;
- une fois l'équation reconstruite, il faut la comparer à la série donnée par l'utilisateur.

L'étape de comparaison sera également divisée en deux sous-étapes. (i) La résolution numérique des équations, pour en obtenir des séries chronologiques, qui est basée sur l'algorithme de Runge-Kutta d'ordre quatre ; (ii) comparer les séries générées depuis les équations, avec celle fournit par l'utilisateur.

5.1 Les processus de conversion

5.1.1 Conversions XML-Arbre

Afin de lire les fichiers XML, nous utilisons le DOM (Document Object Model), un standard du W3C, qui consiste à représenter un document XML, sous la forme d'un arbre d'objets définis par le standard [w3s]. Le standard défini un ensemble de méthodes permettant de naviguer dans l'arbre et d'accéder aux divers éléments.

En java, une bibliothèque native du langage implémente le standard W3C, ainsi que les fonctions nécessaires pour créer le DOM à partir d'un fichier XML et écrire un fichier à partir du DOM. Le langage contient également une bibliothèque capable de certifier (« validate »), que le DOM est conforme à une spécification donnée sous la forme de schéma XML. Ainsi, il est possible de vérifier qu'un document est conforme au schéma, avant d'essayer de le lire, pour éviter les bugs de lecture dus à un fichier corrompu ou non-conforme. Il est aussi possible de certifier que le DOM est conforme à la spécification du schéma, avant d'écrire, pour éviter de générer un document non-conforme.

Une fois le DOM créé par appel des fonctions de la bibliothèque, nous pouvons l'utiliser pour construire l'équation.

Lecture de l'ensemble des paramètres La lecture de l'ensemble des paramètres se fait en récupérant les différentes balises des différents éléments qui compose l'ensemble. Par exemple, les valeurs réelles des variables se lisent en récupérant les balises `variable` (réalisé par une fonction de manipulation du DOM) et en lisant les valeurs des attributs (également une fonction d'accès au DOM). Le code correspondant en Java est disponible dans l'annexe C.3.1.

Lecture de la formule La lecture de la formule se fait à l'aide d'un algorithme de parcours d'arbre infixe. Le DOM étant une structure elle-même arborescente, les deux structures sont similaires. Il suffit donc de « cloner » la structure de la balise `formula` du DOM, en créant les nœuds appropriés. La procédure est décrite en pseudo-code, dans l'algorithme 2. Elle consiste, depuis la racine, à :

- lire le nœud du DOM en cours et créer le type de nœud de l'arbre correspondant ;
- si le nœud du DOM possède un enfant gauche : créer le sous-arbre gauche (en rappelant la fonction de lecture) et affecter le sous-arbre gauche créé en tant que fils gauche du nœud de l'arbre créé ;
- si le nœud du DOM possède un enfant droit : appliquer la même procédure qu'avec le sous-arbre gauche, en affectant le sous-arbre créé en tant que fils droit du nœud de l'arbre créé.

Dans le cas où un nœud n'a pas d'enfant, la récursion s'arrête. L'implémentation Java de l'algorithme est disponible en annexe C.3.2. Les algorithmes de conversions inverses Arbre-XML sont très similaires et sont également en annexe C.3.3 et C.3.4.

5.1.2 Conversion XML-Tuple

5.1.2.1 Les « Flat Objects » intermédiaires

Une équation peut être traduite dans une structure similaire aux tables relationnelles, mais représentée en objets Java, pour pouvoir être facilement manipulée dans un programme. Ces objets sont l'équivalent d'un langage universel, qui n'a pas pour objectif d'être utile à une

Algorithm 2 `Fonction` lire_equation(noeud_courant_xml)

Require: $noeud\_courant\_xml = balise\_racine(" = ")$ {Au premier appel uniquement}
  $noeud\_courant\_arbre \leftarrow creer\_noeud(noeud\_courant\_xml)$
  if $noeud\_courant\_xml$ possède un enfant gauche **then**
    $noeud\_xml\_gauche \leftarrow get\_enfant\_gauche(noeud\_courant\_xml)$
    $left\_children \leftarrow lire\_equation(noeud\_xml\_gauche)$ {Appel récursif de lire_equation}

    $set\_enfant\_gauche(noeud\_courant\_arbre, left\_children)$
    if $noeud\_courant\_xml$ possède un enfant droit **then**
      $noeud\_xml\_droit \leftarrow get\_enfant\_droit(noeud\_courant\_xml)$
      $right\_children \leftarrow lire\_equation(noeud\_xml\_droit)$ {Appel récursif de lire_equation}
      $set\_enfant\_droit(noeud\_courant\_arbre, right\_children)$
    end if
  end if

application particulière, mais à faciliter le passage d'une structure à une autre. En effet, leur structure est assez simple, ce qui permet de facilement les traduire dans (ou de les construire à partir d')un autre formalisme. Les objets sont conçus comme décrit ci-après.

- `Variable` (Table 5.1) : cet objet contient les deux attributs représentants une variable, ainsi qu'un attribut `pk`, qui est sa clé primaire. Cette-dernière est lue dans la base de données lorsque l'objet est construit à partir du résultat d'une requête sur la base de données. Lorsqu'il est construit à partir d'un arbre ou d'un fichier XML, cet attribut est automatiquement généré, en évitant de prendre une valeur déjà présente dans la base de données.
- `InitialValue` (Table 5.2) : cet objet contient les trois attributs définissant une condition initiale d'une équation (un élément de la liste des conditions initiales, i.e de la liste des valeurs initiales de la fonction inconnue et ses dérivées, voir l'exemple 4.3). Il contient aussi un attribut `pk`, qui fonctionne comme pour l'objet `variable`.
- `Input` (Table 5.3) : cet objet contient quatre attributs, qui représente une fonction d'entrées, plus l'attribut `pk`. Parmi les quatre attributs, il y a l'attribut `series`, de type « série chronologique », qui est l'ensemble des valeurs connues de la fonction. Le type « série chronologique » contiens, en plus des valeurs, un attribut `interpolation_method`, qui indique la méthode d'interpolation adéquat pour la série. L'attribut `serial_key` est traité de la même façon que l'attribut `pk`. Cet attribut est une clé, qui indique dans quels fichiers sont stockées les valeurs de la série chronologique, ces-dernières n'étant pas stockées dans la base de données, contrairement aux autres attributs. Ainsi, si l'objet est construit depuis la base de données, le processus d'extraction (voir plus loin, section 5.1.2.3) va récupérer la clé et lire les valeurs dans le fichier correspondant. Dans le cas d'une traduction depuis un arbre ou un fichier XML, les valeurs ne seront écrites dans un fichier et associées à une clé, au moment de l'insertion des données dans la base de données. Dans ce cas, c'est le processus d'insertion (section 5.1.2.3), qui s'occupera : (i) de créer une clé unique ; (ii) affecter la valeur de la clé à l'attribut `serial_key` ; (iii) de créer un fichier nommé avec la clé ; (iv) et d'y écrire les valeurs.
- `Node` (Table 5.4) : cet objet contient les différents éléments qui caractérisent un nœud d'une

Variable
pk : entier
name : string
value : réel

Table 5.1 – Description de l'objet `Variable`

InitialValue
pk : entier
name : string
deriv : entier
initialValue : réel

Table 5.2 – Description de l'objet `InitialValue`

Input
pk : entier
name : string
deriv : entier
series : série chronologique
serial_key : string

Table 5.3 – Description de l'objet `Input`

Node
pk : entier
mathObjet : enumération
name : string
deriv : entier
parent : entier
left : entier
right : entier
equation : entier
pk_content : entier

Table 5.4 – Description de l'objet `Node`

formule (voir section 4.2.2). Notons que le nœud et son contenu sont stockés dans le même objet et non séparé en deux, comme dans la base de données. Cela simplifie grandement la traduction depuis un arbre, ou un fichier XML. Aussi, l'attribut `pk_content` est affecté à une valeur non-utilisé dans la base de données, durant sa lecture. Pour l'attribut `pk`, il est absolument nécessaire de lui affecter une valeur adéquate (non-utilisée dans la base de données), car dans le cas d'une traduction depuis un XML où un arbre, il est nécessaire d'initialiser cet attribut, dès la phase de traduction. Car c'est à l'aide cet attribut, que sont conservés les liens parents enfants, entre les nœuds. Si les clés ne sont pas correctement choisies, de sorte qu'une mise à jour de la clé soit nécessaire, alors il faut mettre à jour les clés en veillant à ne pas utiliser une valeur déjà utilisée dans la base de données, ni dans la table que l'on essaie d'ajouter, car cela risquerait d'introduire une incohérence dans les données, qui pourrait être impossible à résoudre, sans le XML ou l'arbre, original.

- `Formula` : représente une formule, à l'aide de la liste des nœuds qui la compose. Un exemple est détaillé dans la section 5.1.2.2.
- `FlatDifferentialEquation` : représente une équation à l'aide d'une formule (objet `Formula`) et d'une liste de `Variable` d'une liste de `InitialValue` d'une liste de `Input`.

5.1.2.2 Conversion XML-Flat

La lecture des paramètres depuis un XML, consiste à récupérer la liste des balises de l'ensemble des paramètres et à créer les éléments correspondants en « Flat Object » et à les ajouter à la bonne table. Le code Java est disponible en annexe C.4.1.

FlatDifferentialEquation
pk : entier
formula : list of Node
variables : list of Variable
inputs : list of Input
initialValues : list of InitialValues
order : entier
name : string
idGroup : entier
group : string

TABLE 5.5 – Description de l'objet `FlatDifferentialEquation`

Formula									
nodes	1	2	3	4	5	6	7	8	9
mathObject	b-op	b-op	b-op	var	func	func	b-op	var	func
name	$=$	$+$	$\times$	T	y'	y	$\times$	K	u
deriv	null	null	null	null	1	0	null	null	0
left	2	3	4	null	null	null	8	null	null
right	7	6	5	null	null	null	9	null	null
pk_equation	1								
pk_content	1	2	3	3	4	5	6	7	8

TABLE 5.6 – Table formula à partir de l'arbre 4.2b

La création de la table formula se fait par lecture de la structure de l'arbre à l'aide d'un algorithme de parcours infixe. Le parcours et la création se déroule selon les étapes suivantes, pour un nœud de l'arbre, en partant de la racine :

- créer un objet `Node` à partir des informations du nœud en cours, en lui donnant un identifiant arbitraire (une valeur entière positive, dans l'ordre croissant) et l'ajouter dans la table `formula` ;
- regarder la présence d'enfants gauche (le cas échéant, d'un enfant droit) et rappeler la procédure, récursivement ;
- affecter les clés des enfants aux attributs `left` et `right` de l'objet `Node` en cours.

La table 5.6 montre ce à quoi ressemble la table `formula`, pour l'équation 4.3. Le code Java est en annexe C.4.2.

Les conversions inverses : Flat-XML La conversion de l'ensemble des paramètres se fait de manière relativement simple par rapport à la conversion de la formule (code en annexe C.4.4). La conversion de la formule, en revanche, nécessite de reconstruire la structure d'arbre, à partir de la structure linéaire de la table `formula`. Pour cela, on parcourt la table, en suivant les liens entre les éléments, indiqués par les attributs `left` et `right`. Là aussi l'algorithme est récursif et l'on peut décomposer les étapes comme suit (en partant da la racine) :

- créer un noeud du DOM correspondant au nœud en cours (représenter par sa clé dans la table `formula`) ;
- lire les clés du fils gauche (et le cas échéant du fils droit) ;
- rappeler la fonction avec chacune des deux clés.

Cet algorithme ne parcourt donc pas la table de manière linéaire, mais de manière cohérente avec la nature arborescente de la représentation d'une formule. L'algorithme 3 décrit en pseudo-code les étapes décrites ci-dessus. Le code Java de l'algorithme est en annexe C.4.3.

Algorithm 3 `Fonction` ecrire_nœud(table,pk_courante)

Require: $pk\_courante = pk\_racine$ {Au premier appel}
  $noeud\_courant\_xml \leftarrow creer\_noeud(table[pk\_courante])$
  if $table[pk\_courant].pk\_gauche \neq 0$ nœud courant possède un fils gauche **then**
    $noeud\_gauche\_xml \leftarrow ecrire\_noeud(table, table[pk\_courant].pk\_gauche)$ {Appel récursif de ecrire_nœud}
    $set\_enfant\_gauche(noeud\_courant\_xml, noeud\_gauche\_xml)$
    if $table[pk\_courant].pk\_droit \neq 0$ nœud courant possède un fils droit **then**
      $noeud\_droit\_xml \leftarrow ecrire\_noeud(table, table[pk\_courant].pk\_droit)$ {Appel récursif de ecrire_nœud}
      $set\_enfant\_droit(noeud\_courant\_xml, table[pk\_courante].pk\_gauche)$
    end if
  end if

5.1.2.3 Conversions Flat-Tuple

Les conversions Flat-Tuple seront nommées processus d'insertion, pour la conversion des objets « flat » en tuple et processus d'extraction, pour la conversion de tuple en objet flat.

Le processus d'insertion il consiste à écriture les requêtes SQL (voir section 2.1.2) adéquates permettant d'ajouter les éléments contenus dans les objets « Flat ». Il s'agit d'un processus dont le rôle est essentiel, car il doit assurer qu'il n'y aura pas d'ajout d'information déjà présente. Par exemple, deux contenus de nœuds identiques, dans la table `Node_Content`. Il doit aussi assurer que les nœuds, qui constituent la formule d'une équation, restent correctement liés entre eux. Cette tâche est contrainte par la nécessité de s'assurer que les clés temporaires (voir section 5.1.2.1) n'existent pas déjà et, le cas échéant, en générer de nouvelles uniques et mettre à jour la table, sans perdre les liens entre les nœuds. Le processus doit aussi affecter aux différents éléments de l'ensemble des paramètres, des clés uniques entre elles et par rapport à l'ensemble des clés déjà utilisées dans la base de données.

Tout d'abord, les identifiants sont des entiers de type « long » en Java. Il s'agit d'un type d'entier codé sur 64 bits. Ainsi, ils peuvent servir à représenter des entiers positifs, de zéro à $2^{64} - 1$. Les clés sont générées dans l'ordre croissant, en partant de 1.

Lors de l'insertion d'une nouvelle équation, les étapes suivantes sont réalisées :

1. Récupérer les listes des identifiants : pour les variables, l'exemple 5.1 montre la requête permettant de récupérer toutes les clés utilisées pour les variables. Pour l'exemple, on cherche donc à récupérer les clés de la table `Variable`, dont on a besoin que de la colonne `pk`. Le tout est trié dans l'ordre croissant, par l'instruction `ORDER BY`.

   Exemple 5.1

   ```
   SELECT Variable.pk FROM Variable ORDER BY pk
   ```

2. Vérification de l'existence d'une variable : pour une variable dont les attributs sont $(\text{"}var\_name\text{"}, var\_value)$, la requête de l'exemple 5.2 est exécutée, afin de vérifier la présence d'une variable ayant le même contenu. Si la requête renvoie zéro résultat, la nouvelle variable sera ajoutée, sinon, sa clé sera mise à jour avec la clé du tuple obtenu et marquée comme « existante », de façon à ce qu'elle ne soit pas insérée dans la base. En revanche, il faudra tout de même ajouter une ligne supplémentaire dans la table `has_variable` (voir dernière étape).

   Exemple 5.2

   ```
   SELECT * FROM Variable
   WHERE name = "var\_name"
   AND value = var\_value
   ```

3. insertion : si la variable n'existe pas dans la base de données, il faut l'ajouter dans la table `Variable`, en utilisant la première requête de l'exemple 5.3. La seconde requête permet de faire le lien entre la variable et l'équation, car la table `has_variable` indique quelle variables appartiennent à quelle équation. Dans le cas où la variable existe déjà dans la base de donnée, la première requête est ignorée, mais il faut réaliser la seconde, où `var_pk` est la clé de la variable, lue dans la base de données. L'attribut `link_pk` est la clé de la ligne ajoutée dans la table `has_variable`.

   Exemple 5.3

   ```
   INSERT INTO Variable VALUES (var\_pk,"var\_name",var\_value)
   ```

   ```
   INSERT INTO has\_variable VALUES (link\_pk,var\_pk,equation\_pk)
   ```

Les fonctions réalisant ce processus en Java sont en annexe C.5.1.

Le processus d'extraction il consiste à utiliser les requêtes adéquates pour

```
-- Fusion des tables node, node_content et has_node
SELECT Differential_Equation.id AS id_eq,
    Node.id AS id_node,
    Node_Connection.node_depth,
    Node_Content.math_object,
    Node_Content.name,
    Node_Content.deriv,
    Node_Connection.id_parent,
```

```
    Node_Connection.id_left,
    Node_Connection.id_right
FROM Differential_Equation,
    Node,
    Node_Content,
    Node_Connection
WHERE Differential_Equation.id = Node_Connection.id_eq
AND Node_Connection.id_node = Node.id
AND Node.id_content = Node_Content.id;

-- Extraction des variables
SELECT Differential_Equation.id AS id_eq,
    Variable_Value.id AS id_var,
    Variable_Value.name,
    Variable_Value.var_value
FROM Variable_Value,
    Equation_Variable_Value,
    Differential_Equation
WHERE Differential_Equation.id = Equation_Variable_Value.id_eq
AND Variable_Value.id = Equation_Variable_Value.id_var;

-- Extraction des fonctions d'entrees
SELECT Differential_Equation.id AS id_eq,
    Input_Function.id AS id_input_function,
    Input_Function.name,
    Input_Function.deriv,
    Input_Function.serial_key,
    Input_Function.interpolation_function
FROM Input_Function,schema_mmw.Equation_Input,
    Differential_Equation
WHERE Input_Function.id = Equation_Input.id_input_function
AND Differential_Equation.id = Equation_Input.id_eq;

-- Extraction des valeurs initiales
SELECT Initial_Value.id AS id_initial,
    Differential_Equation.id AS id_eq,
    Initial_Value.name,
    Initial_Value.deriv,
    Initial_Value.init_value
FROM Initial_Value,
    Equation_Initial_Value,
    Differential_Equation
WHERE Initial_Value.id = Equation_Initial_Value.id_initial
AND Differential_Equation.id = Equation_Initial_Value.id_eq;

-- Extraction de la table Differential_Equation
SELECT * FROM Differential_Equation;

-- Extraction de la table Differential_Equation, avec le groupe
SELECT Differential_Equation.id,
    Differential_Equation.name,
    Group_Name.group_name
FROM Differential_Equation, Group_Name
WHERE Differential_Equation.id_group = Group_Name.id;
```

```
--
SELECT Differential_Equation.id AS id_eq,
    Group_Name.id,
    Group_Name.group_name
FROM Group_Name,Differential_Equation
WHERE Group_Name.id = Differential_Equation.id_group;
```

5.2 La génération de données théoriques

5.2.1 Résolution numérique par la méthode de Runge-Kutta d'ordre quatre

Il existe diverses méthodes de résolution numérique d'équations différentielles [PTVF02, pp. 710-714], qui ont pour but de calculer des valeurs de la fonction solution à partir d'une solution initiale, notée y_0 à l'instant t_0. L'écart temporel entre les valeurs calculées est fixé avant l'application de la méthode et est noté h. Ainsi on aura : $t_n = t_0 + n \times h$. Nous nous limiterons ici aux équations différentielles linéaires d'ordre 1, qui peuvent être écrites sous la forme :

$$y'(t) = f(t, y(t)) \tag{5.1}$$

Parmi les plus répandues, les méthodes de Runge-Kutta, dont la plus répandue est la méthode de Runge-Kutta, d'ordre quatre. Elle consiste à calculer les valeurs numériques d'une équation différentielles, à l'aide d'une formule définissante une suite récurrente (la valeur d'un élément, dépend de celle d'un ou plusieurs éléments préalablement connus). Par exemple, la méthode de Runge-Kutta d'ordre 4 est l'application de la formule suivante :

$$y_{n+1} = y_n + \frac{1}{6} \times [k_1 + 2 \times k_2 + 2 \times k_3 + k_4] \tag{5.2}$$

avec

$$\begin{cases} k_1 &= h \times f(t_n, y_n) \\ k_2 &= h \times f(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\ k_3 &= h \times f(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\ k_4 &= h \times f(t_n + h, y_n + k_3) \end{cases}$$

Cette méthode se généralise à des équations d'ordre n, grâce à outil mathématique (les vecteurs et les matrices), permettant de ramener une équation d'ordre n à une équation d'ordre 1 via une petite transformation algébrique [RDO01, pp. 207-208]. En effet, une équation différentielle linéaire d'ordre n peut s'écrire sous la forme :

$$y^{(n)}(t) + \alpha_{n-1} y^{(n-1)}(t) + \alpha_{n-2} y^{(n-2)}(t) + \cdots + \alpha_1 y^{(1)}(t) + \alpha_0 y^{(0)}(t) = b(t) \tag{5.3}$$

On transforme artificiellement cette équation en le système d'équation suivant :

$$\begin{cases} y^{(1)}(t) &= y^{(1)}(t) \\ \dots &= \dots \\ y^{(n)}(t) &= -\alpha_{n-1}y^{(n-1)}(t) - \dots - \alpha_0 y^{(0)}(t) - b(t) \end{cases} \tag{5.4}$$

Il s'agit d'un système de n équation linéaire, qui peut s'écrire sous forme matricielle. Pour cela, on commence par poser :

$$Y(t) = \begin{bmatrix} y^{(0)}(t) \\ y^{(1)}(t) \\ \dots \\ y^{(n-2)}(t) \\ y^{(n-1)}(t) \end{bmatrix} \tag{5.5}$$

Les majuscules indiquent des vecteurs.

On a donc :

$$Y'(t) = \begin{bmatrix} y^{(1)}(t) \\ y^{(2)}(t) \\ \dots \\ y^{(n-1)}(t) \\ y^{(n)}(t) \end{bmatrix} \tag{5.6}$$

Le système 5.4 s'écrit alors sous la forme :

$$Y'(t) = AY(t) + B(t) = F(t, Y(t)) \tag{5.7}$$

Où A est une matrice définie comme suit :

$$A = \begin{bmatrix} 0 & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & \dots & 0 & 1 \\ -\alpha_0 & -\alpha_1 & \dots & -\alpha_{n-2} & -\alpha_{n-1} \end{bmatrix} \tag{5.8}$$

et $B(t)$ est le vecteur suivant :

$$B(t) = \begin{bmatrix} 0 \\ 0 \\ \dots \\ 0 \\ b(t) \end{bmatrix} \tag{5.9}$$

L'équation 5.7 est une équation matricielle d'ordre un, où la fonction inconnue Y est une fonction vectorielle, de taille n. Comme il s'agit d'une équation d'ordre un, on peut généraliser la méthode de Runge-Kutta en utilisant une version vectorielle des formules :

$$Y'(t) = F(t, Y(t)) \tag{5.10}$$

$$Y_{n+1} = Y_n + \frac{1}{6}[K_1 + 2 \times K_2 + 2 \times K_3 + K_4] \tag{5.11}$$

$$\begin{cases} K_1 &= h \times F(t_n, Y_n) \\ K_2 &= h \times F(t_n + \frac{h}{2}, Y_n + \frac{1}{2} \times K_1) \\ K_3 &= h \times F(t_n + \frac{h}{2}, Y_n + \frac{1}{2} \times K_2) \\ K_4 &= h \times F(t_n + h, Y_n + K_3) \end{cases} \tag{5.12}$$

Prenons comme exemple, l'équation :

$$y^{(2)}(t) + 2 \times y'(t) + y(t) = t + 1 \tag{5.13}$$

D'abord, on identifie $\alpha_0 = 1$ et $\alpha_1 = 2$.

Ensuite, d'après 5.8, on a donc :

$$A = \begin{bmatrix} 0 & 1 \\ -\alpha_0 & -\alpha_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & -2 \end{bmatrix} \tag{5.14}$$

Puis, en posant :

$$Y(t) = \begin{bmatrix} y^{(0)}(t) \\ y^{(1)}(t) \end{bmatrix} \tag{5.15}$$

On a donc :

$$Y'(t) = \begin{bmatrix} 0 & 1 \\ -1 & -2 \end{bmatrix} \times Y(t) + \begin{bmatrix} 0 \\ t+1 \end{bmatrix} = F(t, Y(t)) \tag{5.16}$$

On prendra, comme conditions initiales :

$$Y(0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{5.17}$$

Ainsi, on peut calculer les différents K_n, pour $t_0 = 0$ et $Y_0 = Y(0)$ (et on prendra un pas h égal à 1) :

$$\begin{cases} K_1 &= F(t_0, Y_0) \\ K_2 &= F(t_0 + \frac{1}{2}, Y_0 + \frac{1}{2} \times K_1) \\ K_3 &= F(t_0 + \frac{1}{2}, Y_0 + \frac{1}{2} \times K_2) \\ K_4 &= F(t_0 + 1, Y_0 + K_3) \end{cases} \tag{5.18}$$

Ce qui donne :

$$K_1 = \begin{bmatrix} 0 & 1 \\ -1 & -2 \end{bmatrix} \times Y_0 + \begin{bmatrix} 0 \\ t_0 + 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & -2 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 + 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{5.19}$$

$$K_2 = \begin{bmatrix} 0 & 1 \\ -1 & -2 \end{bmatrix} \times [Y_0 + \frac{1}{2} \times K_1] + \begin{bmatrix} 0 \\ t_0 + \frac{1}{2} + 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & -2 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0.5 \end{bmatrix} + \begin{bmatrix} 0 \\ 1.5 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \tag{5.20}$$

En appliquant ainsi les formules, pour K_3 et K_4, on obtient :

$$\begin{cases} K_3 &= \begin{bmatrix} 0.25 \\ 0.25 \end{bmatrix} \\ K_4 &= \begin{bmatrix} -0.25 \\ 0.25 \end{bmatrix} \end{cases} \tag{5.21}$$

Et on obtient :

$$Y_1 = Y_0 + \frac{1}{6}[K_1 + 2 \times K_2 + 2 \times K_3 + K_4] = \frac{1}{6}[\begin{bmatrix} 0 \\ 1 \end{bmatrix} + 2 \times \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} + 2 \times \begin{bmatrix} 0.25 \\ 0.25 \end{bmatrix} + \begin{bmatrix} -0.25 \\ 0.25 \end{bmatrix}] = \begin{bmatrix} 0.2083 \\ 0.4583 \end{bmatrix} \tag{5.22}$$

Et d'après la définition de Y 5.15, on a :

$$Y_1 = \begin{bmatrix} y_1 \\ y_1' \end{bmatrix} \tag{5.23}$$

D'où : $y_1 = 0.2083$. En réitérant, on peut ainsi récupérer la suite des valeurs de y. L'algorithme 4 décrit en pseudo-code l'implémentation de l'algorithme de Runge-Kutta.

Algorithm 4 Algorithme de Runge-Kutta

$t\_courant \leftarrow get\_initial\_time(equation)$
$Y_n \leftarrow get\_initial\_values(equation)$
$add\_data\_point(results\_series, t\_courant, Y\_n[0])$
while $t\_courant \in time\_values(equation)$ **do**
  $Y_n \leftarrow compute\_next\_Y(Y_n, t\_courant, get\_functions\_values(equation, t\_courant))$
  $add\_data\_point(results\_series, t\_courant, Y\_n[0])$
  $t\_courant \leftarrow next\_time\_value(equation)$
end while

Remarque sur l'ordre des erreurs L'ordre de la méthode de résolution (ici quatre), correspond à l'ordre de l'erreur commises sur les valeurs calculées. Autrement dit, une méthode d'ordre quatre, signifie que pour tout n l'erreur commise $e_n = x_n^{th} - x_n$ (où x_n^{th} est la valeur que prendrait x au point t_n s'il était possible de résoudre l'équation et d'obtenir la formule explicite de x sur son domaine de définition) est un $O(h^2)$. De manière générale, une méthode est dite d'ordre N, lorsque l'erreur e_n est un $O(h^{N+1})$. Donc la méthode de Runge-Kutta est un $O(h^5)$. Mais dans le cas de l'algorithme de Runge-Kutta, augmenté l'ordre, nécessite d'augmenter la quantité de calcul, par étape. En effet, la méthode de Runge-Kutta d'ordre 1 (plus connue sous le nom de méthode d'Euler explicite) ne fait appel qu'à $F(t_n, Y_n(t))$, à chaque pas de résolution.

Remarque sur l'interpolation Si nous reprenons l'exemple de l'équation 4.3, le terme $b(t)$ est égal à $Ku(t)$. Dans l'exemple développé plus haut, nous avons pris : $b(t) = t + 1$. Ainsi, nous avons la formule explicite de $b(t)$. Or, dans notre cas général, nous ne connaissons pas la formule explicite des fonctions. Dans notre exemple 4.3, nous n'avons pas la formule explicite de $u(t)$, nous avons seulement les valeurs de u aux instants $(t_n)_{n\in\mathbb{N}}$. Ce qui signifie que pour $n \in \mathbb{N}$ nous connaissons $u(t_n) = u_n$ et $u(t_{n+1}) = u_{n+1}$. Par définition, le pas h est l'écart entre t_n et t_{n+1}, autrement dit $h = t_{n+1} - t_n$ et donc $t_{n+1} = t_n + h$.

Si nous observons les formules de calcul des K_n 5.12, nous remarquons que K_1 dépend de t_n et K_4 dépend des $t_n + h = t_{n+1}$. Et la fonction F contient $B(t)$ 5.10. Appliqué à notre exemple 4.3, cela signifie que pour calculer K_1 et K_4, nous avons besoin de $u(t_n) = u_n$ et de $u(t_n + h) = u(t_{n+1}) = u_{n+1}$. Ainsi, le calcul de K_1 et K_4 ne pose pas de problème, puisque nous avons les valeurs de u_n et u_{n+1}.

En revanche, les calculs de K_3 et K_4 dépendent tous les deux de $t_n + \frac{h}{2}$, qui n'est pas une valeur de la suite $(t_n)_{n\in\mathbb{N}}$. Ce qui signifie que nous ne connaissons pas la valeur de $u(t_n + \frac{h}{2})$. C'est pourquoi, nous avons besoin d'interpoler cette valeur. C'est-à-dire, en déterminer une valeur approchée, à partir de la connaissance de u_n et u_{n+1}.

Le principe de l'interpolation consiste à émettre une hypothèse sur la fonction mathématique que suit u, sur l'intervalle $[t_n, \ldots, t_{n+1}]$. Nous pouvons, par exemple, supposer que les valeurs évoluent de façon linéaire sur l'intervalle cité plus tôt. Ainsi, nous émettons l'hypothèse qu'entre t_n et t_{n+1}, nous avons :

Timestamp	Value
0.0000000e+00	0.00000000
1.0000000e+00	0.094105346
2.0000000e+00	0.18452077
3.0000000e+00	0.27139095
4.0000000e+00	0.35485491
5.0000000e+00	0.43504619
6.0000000e+00	0.51209313
7.0000000e+00	0.58611902
8.0000000e+00	0.65724231
...	...

FIGURE 5.1 – Série chronologique correspondant à l'équation 4.3

Iteration	Value
1	0
2	0.094105340000000
3	0.184520764762767
4	0.271390946244791
5	0.354854899263603
6	0.435056183944733
7	0.512093123446397
8	0.586119009303915
9	0.657242298722463
...	...

FIGURE 5.2 – Valeurs régénérées à l'aide de la méthode de Runge-Kutta d'ordre 4

$$u(t) = at + b = \frac{u_{n+1} - u_n}{t_{n+1} - t_n} \times (t - t_n) + u_n = \frac{u_{n+1} - u_n}{h} \times (t - t_n) + u_n \tag{5.24}$$

En remplaçant t, par t_n et t_{n+1}, successivement on retrouve que $u(t_n) = u_n$ et $u(t_{n+1}) = u_{n+1}$ et pour $t = t_n + \frac{h}{2}$, on trouve :

$$u(t_n + \frac{h}{2}) = \frac{u_{n+1} - u_n}{h} \times (\frac{h}{2}) + u_n = \frac{u_{n+1} + u_n}{2} \tag{5.25}$$

Nous sommes maintenant en mesure de proposer une valeur pour $u(t_n + \frac{h}{2})$ et de terminer le calcul de K_3 et K_4.

Selon les séries chronologiques, l'hypothèse de l'évolution linéaire n'est pas toujours le meilleur choix. C'est la raison pour laquelle, les séries chronologiques sont associées à une méthode d'interpolation linéaire adaptée. Ainsi, l'algorithme de résolution peut faire appel à la bonne fonction d'interpolation, pour pouvoir faire ses calculs.

5.2.2 Le processus de génération

Le processus de génération consiste à appliquer l'algorithme de Runge-Kutta, décrit ci-avant, à une ou plusieurs des équations de la base de modèles. Le tableau 5.1 donne les dix premières valeurs de l'ensemble des valeurs expérimentales, qui, après analyse, donnent l'équation 4.3. Le tableau 5.2 donne les dix premières valeurs rendues en appliquant l'algorithme de Runge-Kutta à l'équation 4.3. La génération de valeurs, constitue une des fonctionnalités fondamentales de la base de modèles, car il permet ensuite de comparer les valeurs numériques générées avec les valeurs expérimentales, apportées par l'expérimentateur/utilisateur.

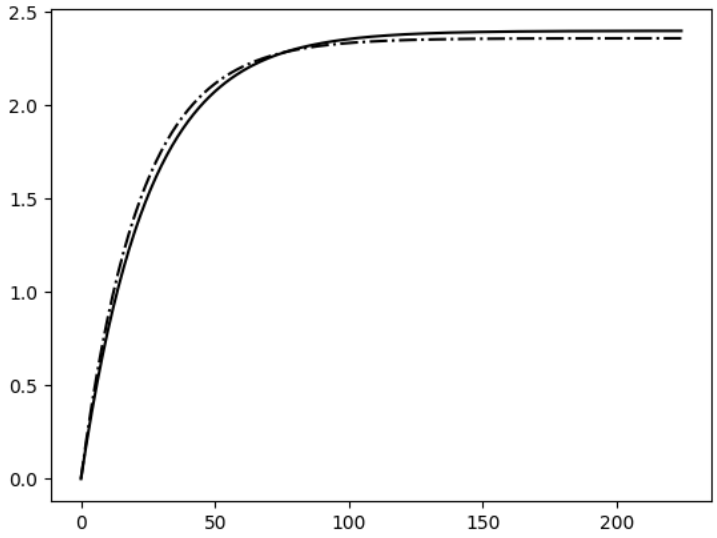


FIGURE 5.3 – Exemples de séries chronologiques, représentées graphiquement

5.3 La comparaison de données théoriques et expérimentales (comparaison série-équation)

Maintenant, que nous pouvons générer une série chronologique, à partir d'une équation différentielle, il faut pouvoir les comparer avec les séries chronologiques fournies par l'utilisateur. Les séries chronologiques générées depuis les modèles théoriques, seront dites séries théoriques, tandis que les séries chronologiques, généralement issues d'une expérimentation, fournies par l'utilisateurs, seront dites séries expérimentales.

5.3.1 Comparaison de séries chronologiques

La comparaison d'une série expérimentale avec équation différentielle (avec sa série théorique associée) aboutie à trois cas de figures :

- l'équation correspond aux données expérimentales, i.e. les séries expérimentale et théorique sont suffisamment similaires et l'équation sera alors annotée `ModelAccepted` ;
- l'équation ne correspond pas aux données expérimentales, i.e. les séries expérimentales et théorique sont trop différentes l'une de l'autre et l'équation sera alors annotée `ModelRejected` ;
- et le cas d'indécision, i.e. les séries expérimentale et théorique sont à la fois, pas assez similaires pour affirmer que l'équation correspond aux données expérimentales, mais pas assez différentes, pour être sûr qu'elle ne correspond pas et l'équation sera alors annotée `Undetermined`.

La comparaison de séries chronologiques est une tâche complexe. En effet, il faut comparer des valeurs réelles continues, ce qui rend inefficace, la comparaison exacte, valeur par valeur. Il faut donc employer des méthodes, qui puissent prendre en compte les approximations et la forme globale des séries chronologiques. Par exemple, sur la figure 5.3, la courbe pleine représente

la solution de l'équation 4.3, au cours du temps et la courbe en trait interrompue est une courbe dont on a légèrement changé les paramètres. On voit que les courbes ne correspondent pas exactement, mais leur comportement sont similaires et leurs profils restent malgré tout proches. Il n'est donc pas évident d'affirmer que les deux courbes sont différentes, en effet, en fonction des usages, un utilisateur pourra vouloir considérer qu'elles sont suffisamment proches pour être considérées égales, là où un autre utilisateur ne voudra/pourra pas se permettre autant d'approximation.

Afin de comparer des séries chronologiques, nous avons dans un premier temps, choisit une méthode qui s'appuie sur la série des erreurs en valeurs absolue. Considérons une série expérimentale $(y_n)_{n\in\mathbb{N}}$ et une série théorique $(x_n)_{n\in\mathbb{N}}$. La série des erreurs en valeur absolue sera notée $(e_n)_{n\in\mathbb{N}}$ où : $\forall n \in \mathbb{N}, e_n = |y_n - x_n|$. L'utilisation de cette série permet de considérer la moyenne des erreurs commises, ainsi que son écart type. L'utilisation de la valeur absolue a pour but d'éviter qu'un ensemble d'erreurs négatives (des valeurs de (x_n) sont plus grandes que celles de y_n) ne puissent être compensées par un ensemble d'erreurs positives (des valeurs de (x_n) plus petites que celles de (y_n)). Par exemple, sur la figure 5.3, la courbe interrompue est d'abord au-dessus de la première, avant de passer en dessous, il s'agit d'un cas où la série des erreurs possèdent des valeurs positives et négatives, qui pourraient s'annuler mutuellement.

On notera μ_{n_e-1} la moyenne de la série (rappelons que n_e est le nombre d'élément de la série e). Classiquement, cette moyenne ce calcul avec la formule suivante :

$$\mu_{n_e-1} = \frac{1}{n_e} \sum_{n=0}^{n_e-1} e_n \tag{5.26}$$

et l'écart type sera calculée, comme étant la racine carrée de la variance, à l'aide de la formule suivante :

$$\sigma_{n_e-1} = \sqrt{\frac{1}{n_e} \sum_{n=0}^{n_e-1} (e_n - \mu_{n_e-1})^2} = \sqrt{var_{n_e-1}(e)} \tag{5.27}$$

Le problème de ces formules, c'est qu'elles nécessitent une somme d'éléments de e, qui en plus sont, par définition toutes positives, donc la somme ne peut que croître. Ainsi, si la somme contient beaucoup d'éléments, il y a un risque qu'elle grossisse au-delà de la capacité de représentation des réelles en informatique. Ainsi, nous avons préféré utiliser les versions incrémentales de ces formules [Fin09].

La moyenne sera calculée avec la formule suivante :

$$\begin{cases} \mu_0 = e_0 \\ \forall n \in [\![1..n_e - 1]\!], \mu_n = \mu_{n-1} + \frac{1}{n+1}(e_n - \mu_{n-1}) \end{cases} \tag{5.28}$$

La fraction $\frac{1}{n+1}$ est un ajustement de la formule, car l'indexation commence à $n = 0$. La

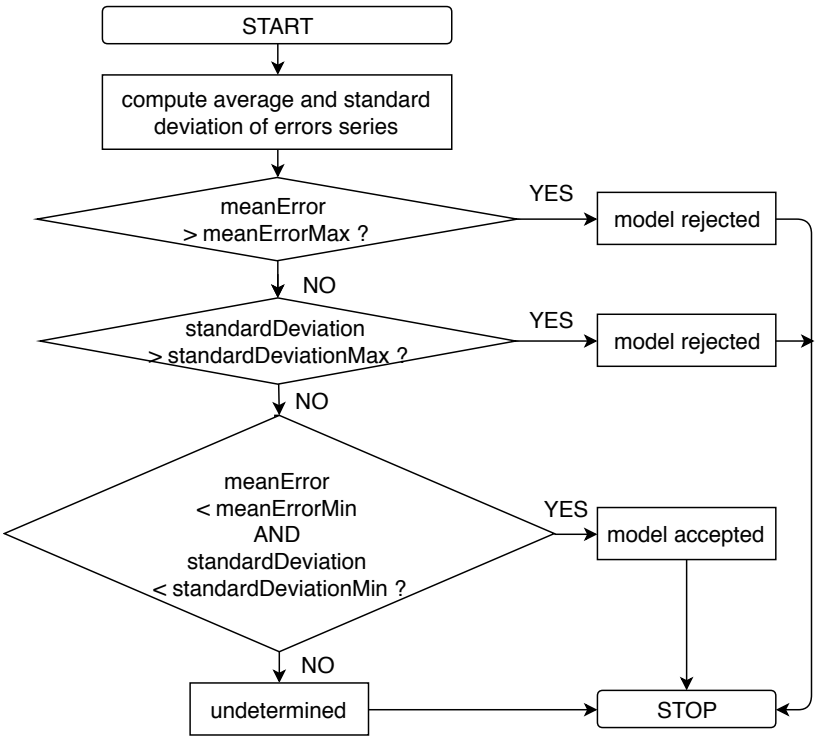


FIGURE 5.4 – Diagramme de l'algorithme de comparaison

variance sera calculée à l'aide de la formule suivante :

$$\begin{cases} S_0(e) = (e_0 - \mu_{n_e})^2 \\ \forall n \in [\![1..n_e - 1]\!], S_n(e) = S_{n-1}(e) + (e_n - \mu_{n-1}(e_n - \mu_n)) \end{cases} \tag{5.29}$$

avec : $S_n(e) = (n+1)var_n(e)$.

L'écart type sera finalement : $\sigma_{n_e-1} = \sqrt{S_{n_e-1}(e)/(n+1)}$.

Amélioration Sur le conseil d'un automaticien (Pr. Thierry POINOT, Université de Poitiers), nous avons ensuite considéré le carré de l'erreur. Car les méthodes d'analyse numérique classiques en automatique, repose souvent sur la méthode des moindres carré, qui consiste à minimiser la somme des carrés des erreurs. La méthode et les formules restent les mêmes, mais la série des erreurs devient : $\forall n \in \mathbb{N}, e_n = (y_n - x_n)^2$. Notons que l'élévation au carré permet, elle aussi, d'éviter le problème de compensation des erreurs positives et négatives.

5.3.2 le processus de comparaison

Après le calcul de ces valeurs nous définissons des valeurs seuils, afin de décider si une équation est acceptable, au vu des donnés. Nous utilisons un ensemble de quatre valeurs seuils, un minimum et un maximum pour la moyenne et pour l'écart type. Ces valeurs sont paramé-

trables, dans l'objectif à terme de fournir à l'utilisateur expert (le scientifique) la possibilité de donner des valeurs, qu'il juge cohérentes avec ses objectifs. La disjonction des cas présentée plus haut plus, se fait selon les critères suivants :

- `ModelAccepted` : il faut que la moyenne et l'écart type soient tous les deux inférieurs à leurs seuils minimums respectifs.
- `ModelRejected` : il faut que la moyenne ou l'écart type soit supérieur à son seuil maximum respectifs.
- `Undetermined` : aucun des cas précédents n'est vérifié.

La figure 5.4 résume sous forme de diagramme l'algorithme de comparaison basé sur les explications faîtes ci-avant. L'implémentation Java de l'algorithme décrit ci-avant est dans l'annexe C.2.2.

5.4 Définition d'une architecture fonctionnelle générique pour la base de modèles

Afin de pouvoir comparer une série chronologique aux équations contenues dans la base de modèles, nous avons défini un processus de génération et un processus de comparaison. Le processus de génération consiste à transformer une équation théorique en une série chronologique (alors dite théorique également), à l'aide de l'algorithme de Runge-Kutta d'ordre quatre, qui permet de résoudre numériquement une équation différentielle, pour en retrouver les valeurs numériques. Le processus de comparaison, quant à lui, consiste à comparer une série chronologique expérimentales, données par l'utilisateur/expérimentateur, aux séries chronologiques théoriques, générées à partir des équations de la base de modèles. Le processus global de traitement numérique des séries chronologiques consiste donc à relier ses deux processus. La figure 5.5 montre comment les deux processus s'enchaînent. Le processus de génération vient en premier interroger la base de données, afin de récupérer les modèles qu'elle contient, puis les résouts, pour générer un ensemble de séries chronologiques théoriques. Ensuite, le processus de comparaison itère sur l'ensemble des séries chronologiques théoriques, pour les comparer à la série expérimentale en cours. Pour chaque série, l'opération `notify` consiste à créer une notification, qui contient :

- l'équation correspondant à la série théorique en cours ;
- les valeurs des variables statistiques (moyenne des erreurs et écart type) calculées au cours de la comparaison ;
- la classification de l'équation (`ModelAccepted`, `ModelRejected` ou `Undetermined`).

Ces notifications pourront être capturées par une application cliente, pour permettre à l'utilisateur de les consulter. Et le processus prend fin, quand toutes les séries théoriques ont été traitées.

Sur la base de cette séparation de fonctionnalités, entre comparaison et génération, nous proposons une architecture fonctionnelle générique, pour une base de modèle, représentée sur la figure 5.6. Cette architecture contient les éléments suivants :

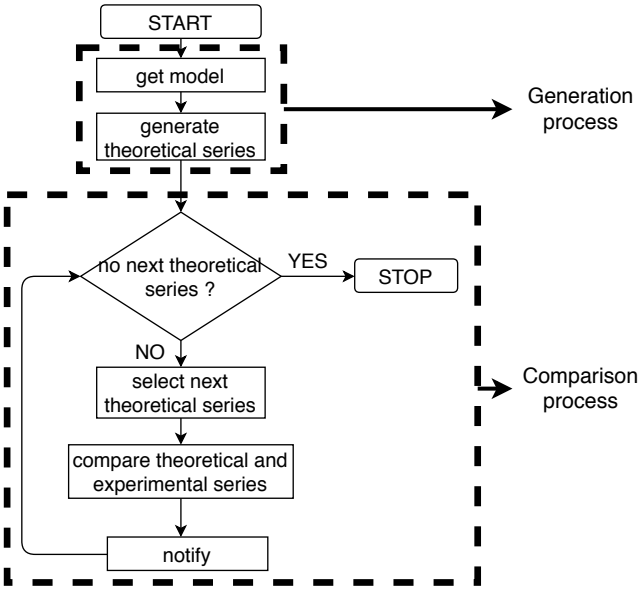


FIGURE 5.5 – Processus global de gestion des séries chronologiques (Comparaison/Génération)

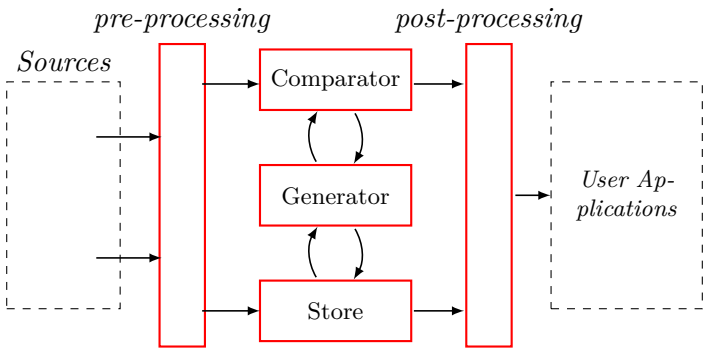


FIGURE 5.6 – Schéma générique du gestionnaire de modèles

- La base de données (`Store`) : il s'agit de la couche de persistance de la base de modèles. Dans notre cas, elle contient donc les équations différentielles et il s'agit d'une base de données relationnelles, dont le schéma est celui définie sur la figure 4.4, dans la section 4.2.3.
- Le module de génération (`Generator`) : il implémente le processus de génération. Il peut communiquer avec la base de données, mais uniquement pour y extraire des équations.
- Le module de comparaison (`Comparator`) : il implémente le processus de comparaison. Il communique donc avec le module de génération, mais ne communique pas avec la base de données. Lorsque l'utilisateur réalise une requête de comparaison, ce dernier va consulter le générateur, pour avoir des séries théoriques et c'est le générateur, qui ira interroger la base de données, pour générer les séries théoriques et les fournir au processus de comparaison.
- Les modules de pré et post-traitement (`pre-processing` et `post-processing`) : ils implémentent, notamment, les algorithmes de conversions nécessaires, pour mettre les données au format de la base de données, dans le cas du pré-traitement et pour formater les données, en XML où dans un autre format, pour communiquer des données (les équations, mais aussi les notifications du module de comparaison) avec des applications externes.
- Les sources : elles peuvent être soit des séries chronologiques, que l'utilisateur souhaite comparer aux modèles présents dans la base de modèles, soit des nouveaux modèles, que l'utilisateur souhaite ajouter dans la base. Les différents formats de données sont reconnus et « aiguiller » au niveau de la couche de pré-traitement.
- Les applications clientes : ce sont toutes les applications qui pourront s'intégrer avec la base de modèles, pour que l'utilisateur puisse l'exploiter, selon ses besoins.

5.5 Conclusion

Afin de représenter, stocker et gérer les équations différentielles, nous avons dû définir des processus de conversion entre les différents formats de données (XML, tuples et objets java). Ainsi que les besoins en traitement numérique (génération, comparaison). Enfin, l'architecture fonctionnelle générique permet de modéliser les différents processus définies, afin de définir la structure d'une base de modèles.

Ladite structure est inspirée de celle des entrepôts de données, qui consiste en une chaîne linéaire de traitement des données. Où les sources, généralement hétérogènes sont traitées, via le processus ETL (Extract, Transfom, Load), dont le rôle est d'extraire les données utiles et les « formatter » pour pouvoir les stocker dans l'entrepôt. Une fois stockées, les données sont mises à disposition de l'utilisateur, via des applications métiers ou clientes. Dans notre cas, les sources de données sont les modèles théoriques générés par les scientifiques. Mais la chaîne de traitement de base à due être enrichie, pour pouvoir également prendre en compte des données numériques. La nouvelle chaîne n'est plus linéaire et possède « deux têtes », deux entrées, une pour chaque type de données d'entrées (modèles ou séries chronologiques) et les applications métiers ou clientes peuvent prendre en compte, les fonctionnalités supplémentaires, offertes par la base de modèles.

Les fonctionnalités supplémentaires permettent de tirer avantage d'un système de « requête par les données », qui permet de réaliser des requêtes non supportées par le langage SQL classique. En effet, le langage SQL ne supporte pas la notion de comparaison de données numériques brutes avec des modèles théoriques. En SQL, les comparaisons se font généralement sur des valeurs précises d'attributs, qui sont en générales de type discret ou une combinaison de type discret (entier, caractères, chaînes de caractères...). Mais les méthodes de comparaison dont nous avons besoin avec les séries chronologiques doivent être suffisamment souple, pour ne pas être trop sensibles aux approximations.

Pour l'instant, les fonctionnalités de requêtes non-gérée par le langage SQL sont implémentées directement en Java. L'élaboration d'une extension du langage permettant d'écrire plus simplement ces requêtes, de sorte que l'utilisateur puisse les réaliser, sans avoir à se soucier de leur implémentation, pourra faire l'objet de futurs travaux.

Chapitre 6

Prototype et expérimentations

À partir de la description fonctionnelle générale d'une base de modèle, résumée sur le schéma 5.6, nous avons implémenté un prototype, afin de démontrer la faisabilité de la base de modèles décrites dans les deux chapitres précédents. Les principaux éléments du code sources sont en annexes C et ont été mentionnés dans le courant du chapitre 5. Le code complet est disponible sur la forge du laboratoire : **`https://forge.lias-lab.fr/projects/mathmouse/files`**. L'objectif du prototype était de donner un exemple concret et fonctionnel, d'implémentation de base de modèles et de pouvoir en faire la démonstration, à travers une interface graphique. Nous avons donc cherché à implémenter les fonctionnalités principales, qui sont les suivantes :

- le stockage de modèles, avec la possibilité d'ajouter ses propres modèles à la base de modèles ;
- un système de requête « par les données », permettant de reconnaître une équation de la base de modèles, à partir d'un jeu de données expérimentales.

Ainsi, dans ce chapitre, nous détaillerons l'implémentation du prototype, qui comprends : (i) une base de données PostgreSQL ; (ii) des modules de code (sous la forme de microservices, voir 6.1.1, qui reprennent les différents éléments de l'architecture logicielle données dans la section 5.4 ; (iii) et une interface graphique, qui fait office d'application clientes.

6.1 Architecture logicielle et détails d'implémentation

Le schéma 6.1 montre l'architecture logicielle du prototype. Cette dernière est composée de plusieurs modules, dont les rôles et l'agencement est, en grande partie, basée sur le découpage fonctionnel de la figure 5.6 (voir section 5.4).

- `DBMS` : le module nommé `DBMS` correspond au SGBD, il s'agit du module correspondant à la fonction `Store` du schéma générale 5.6. Ce module est implémenté à l'aide de Postgresql.
- `DBService` : Le module `DBService`, sert d'interface entre le module `DBMS`, qui est une base de données relationnelles et le reste des modules programmées en Java. Ce module est responsable de l'écriture des requêtes, qui permettent d'interagir avec la base de données. Il implémente donc, les processus d'extraction et d'insertion. Par rapport au schéma principal, il correspond aux modules de pré-traitement des modèles, pour les opérations d'insertion et de post-traitement, pour les opérations d'extractions.
- `Generators` : ici, le module est en pointillé et aux pluriels, car en réalité, il peut y avoir plusieurs générateurs différents. Que ce soit des générateurs implémentant des méthodes

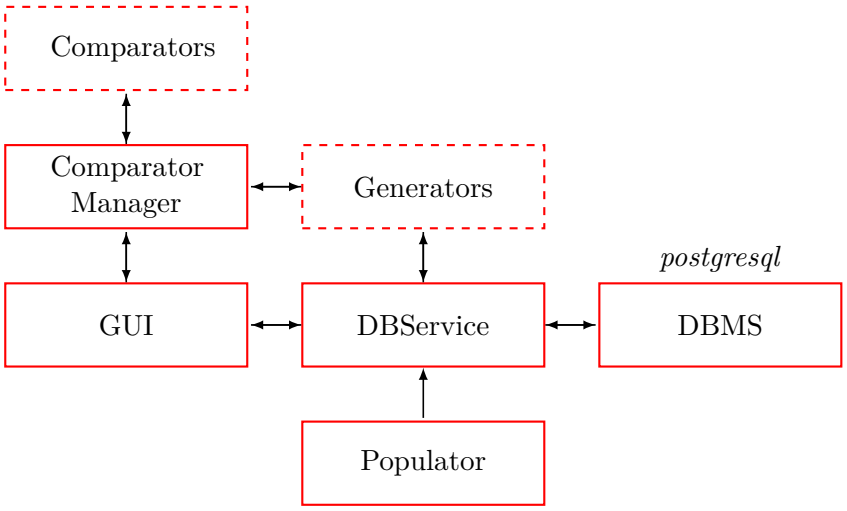


FIGURE 6.1 – Architecture des services

différentes (autre que Runge-Kutta), ou plusieurs instances d'une méthode commune, afin de paralléliser les calculs et optimiser les performances de calcul. Dans notre cas, c'est surtout pour le second scénario, que nous avons introduit la possibilité d'avoir plusieurs générateurs.

- `Comparators` : ici aussi, il est possible d'avoir plusieurs comparateurs, pour les mêmes raisons que les générateurs. Les modules générations/comparaisons, concentrent à eux deux la majorité des besoins en performance de calcul et nécessite d'être optimisés. Nous pensons cependant, que pour des volumétries importantes, de nombreuses optimisations pourront venir d'une heuristique de comparaison plus évoluée. En effet, l'algorithme de comparaison que nous avons mis en place, peut être qualifié de « brute force », dans le sens, où il compare tous les modèles un par un. Une piste intéressante pourrait être d'ajouter des métadonnées sur les modèles, de sorte à les grouper en catégories ou modèles similaire. Ainsi, la discrimination d'un modèle, pourrait entraîner la discrimination des modèles qui lui sont similaires, en économisant le coût de génération des données et le coût de comparaison.
- `Comparator Manager` : il permet de gérer la distribution des opérations de comparaison. La parallélisation des modules de comparaison nécessite un manager, car il faut que chaque comparateur se souviennent de la série expérimentale, à laquelle il doive comparer la série théorique et pour cela, il faut un module, qui coordonne les différentes opérations de comparaison à réaliser, avant de les déléguer. Ce n'est pas le cas du générateur, qui n'a qu'à générer les séries théoriques, à partir des données d'une équation différentielle, sans se préoccuper des séries expérimentales. Ainsi, chaque générateur peut fonctionner de manière totalement indépendante.
- `Populator` : ce module n'est techniquement pas utile et il ne correspond à aucune fonctionnalités du schéma générique. Ici, il sert simplement à insérer automatiquement un ensemble d'équations différentielles dans la base de modèles, pour pouvoir rapidement peupler la base de données, avant de réaliser les tests.
- `GUI` : voir 6.1.2.

6.1.1 Détails supplémentaires à propos de l'implémentation

L'architecture logicielle de la figure 6.1 est une architecture en microservices. Chaque module est un programme totalement indépendant du reste du code et expose une API, pour permettre aux autres éléments d'interagir avec lui. Les microservices sont exécutés à l'intérieur de conteneurs docker [1] (un conteneur = un microservice). Pour faire communiquer les services entre eux, nous avons choisi le broker RabbitMQ [2]. Il existe un conteneur RabbitMQ [3] disponible dans le dépôt de conteneurs officiels de docker. Ledit, dépôt officiel contient également un conteneur pour Postgresql [4]. Tous les autres microservices ont été développées en Java et communique avec le serveur(/conteneur) RabbitMQ, à l'aide de la librairie cliente Java de rabbitMQ [5]. Le service `DBService` implémente, également la librairie JDBC, qui permet d'interagir avec une base de données relationnelle, ce qui lui permet de communiquer avec le service `DBMS`.

Nous avons choisi de travailler avec une architecture microservices, pour plusieurs raisons. (i) Forcer la séparation du code des différentes fonctionnalités (blocs de code indépendants) et ainsi, avoir plus de facilités à tester chaque bloc de code, de manière indépendante. (i) Nous pensons que cela peut-être utile pour développer les services de comparaison et génération, de manière indépendante et modulaire. Une perspective d'amélioration de est permettre à l'utilisateur de choisir une méthode de génération et de comparaison, selon ce qui paraît le plus approprié à ces besoins. Cela demande d'implémenter plusieurs modules ayant des méthodes différentes. L'utilisation des microservices est donc pensée pour faciliter la création, le test et l'intégration de ces modules supplémentaires. Notamment, car cela permet d'ajouter des modules qui ne sont pas obligés d'être codé en Java, ainsi, l'utilisateur peut utiliser le langage avec lequel il est à l'aise et surtout utiliser un langage, tel que le langage Python, qui très utilisé par les programmes de calculs scientifiques. (i) La génération et la comparaison étant des opérations lourdes en calculs. Nous pensons que pour des volumétries importantes, un passage à l'échelle horizontale (distribution des charges de calculs sur plusieurs machines en réseaux) est à prévoir. Les microservices permettent de facilement créer de nouvelle instance (nouveau conteneur) d'un même code et de paralléliser les calculs (il faut tout de même que le(s) service(s) ait été pensé(s) pour être répliqué(s), comme vu plus haut, la réplication du module de comparaison, par exemple, n'est pas évidente, car elle nécessite une supervision). De plus, docker permet la création et la destruction dynamique de conteneurs, signifiant que le nombre de conteneur pourra être dynamiquement adapté à la charge effective de calcul.

6.1.2 `GUI`

Le module `GUI` est un exemple d'application cliente. Elle ne fait pas partie, à proprement dit, de la base de modèle, elle sert simplement à montrer les différentes interactions possibles

1. `https://www.docker.com/`
2. `https://www.rabbitmq.com/`
3. `https://hub.docker.com/_/rabbitmq/`
4. `https://hub.docker.com/_/postgres/`
5. `https://www.rabbitmq.com/java-client.html`

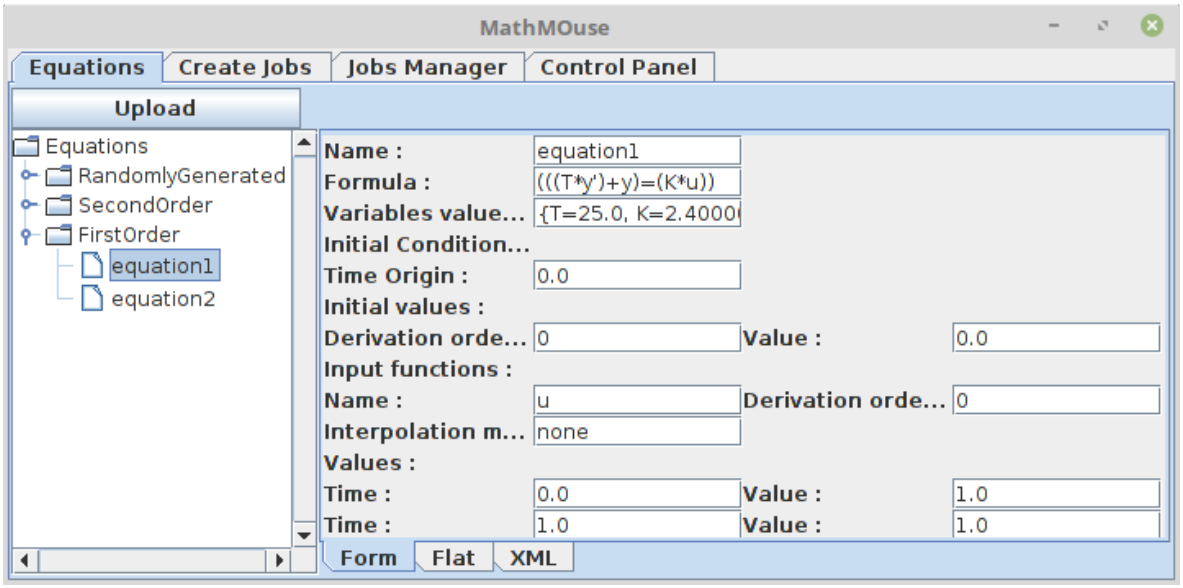


FIGURE 6.2 – Aperçu de la fenêtre d'exploration/affichage

avec cette-dernière. L'interface graphique, que nous avons implémentée permet de réaliser les opérations suivantes :

- visualiser les modèles présents dans la base de données ;
- ajouter un nouveau modèle, à partir d'un fichier XML ;
- sélectionner des séries chronologiques expérimentales, contenues dans des fichiers textes au format csv, afin de les comparer aux modèles présents dans la base de données ;
- visualiser en temps-réel la progression des processus de comparaison et les résultats obtenus.

Visualisation des modèles et ajout d'un modèle La figure 6.2 est un aperçu de l'outil de visualisation des équations présentes dans la base de modèles. La partie de gauche, permet d'afficher les équations. Cette zone est organisée en arborescence, similaire à un explorateur de fichiers, qui permet d'accéder à la liste des équations, identifiées par leurs noms et réparties dans les groupes (définies par l'utilisateur). Cette zone est nommée, zone d'exploration ou explorateur. À droite, l'espace d'affichage du contenu de l'équation sélectionnées. Cette zone est nommée, zone d'affichage.

Pour ajouter un modèle, l'utilisateur peut cliquer sur le bouton « Upload » situé juste au-dessus de l'explorateur. Ce faisant, une sous-fenêtre s'ouvre, comme on peut le voir sur la figure 6.3. Cette-dernière permet à l'utilisateur de choisir, dans son système de fichier local, le fichier XML correspondant à l'équation, qu'il souhaite ajouter. Une fois le fichier sélectionné, il peut cliquer sur « Open ». Le logiciel va alors ouvrir le fichier et le lire et une autre fenêtre apparaîtra (en remplaçant la fenêtre de choix du fichier), pour afficher le contenu du fichier à l'utilisateur. La visualisation du contenu dans la nouvelle fenêtre est similaire à celle de la zone d'affichage de la fenêtre principale. Cette visualisation permet à l'utilisateur de s'assurer que le contenu qu'il s'apprête à ajouter est bien celui auquel il s'attend. Dans la nouvelle fenêtre de validation,

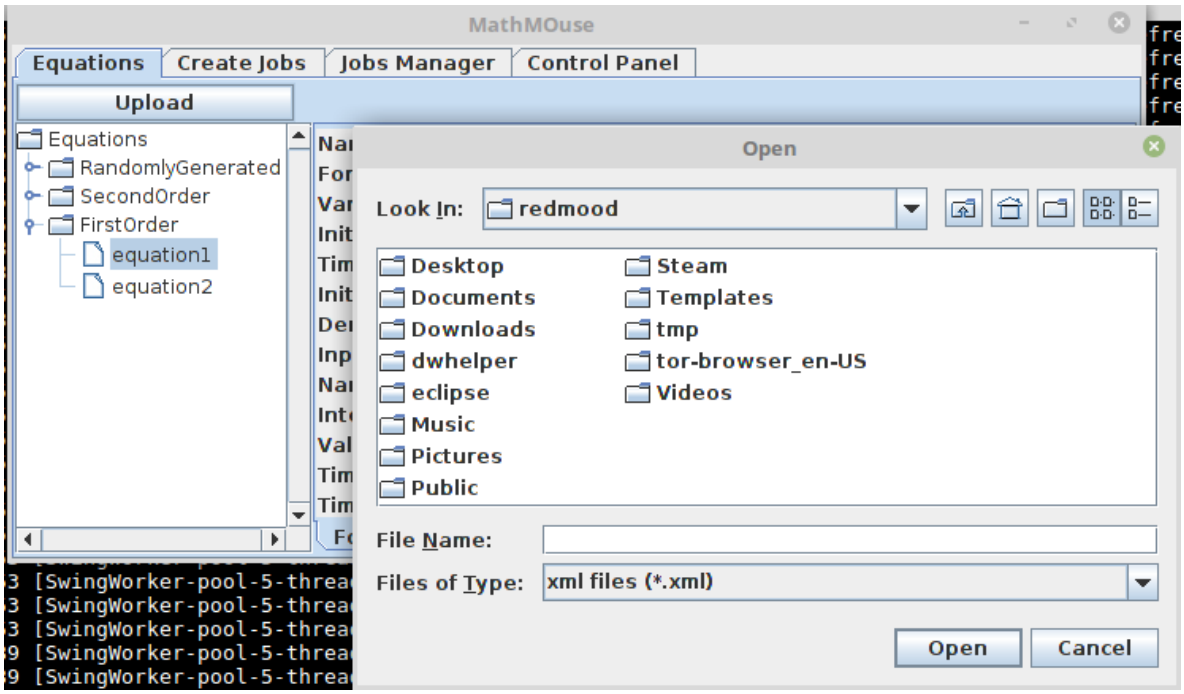


FIGURE 6.3 – Ajout d'une équation dans la base de modèles

l'utilisateur aura la possibilité d'annuler l'opération, à l'aide d'un bouton « Cancel », ou de valider l'opération, à l'aide d'un bouton « Validate ». La validation provoquera l'ajout de l'équation dans la base de modèles. Une fois l'équation ajoutée, le module `DBService` renvoie une notification à vers l'interface, pour que cette dernière mette à jour l'explorateur, avec la nouvelle équation ajoutée. À l'heure actuelle, il n'y a pas de gestion de l'erreur en cas d'échec de l'ajout, mais si l'opération échoue, il n'y a pas de notification retour du module `DBService` et donc pas de mise à jour de l'explorateur.

Requêtes de comparaison, visualisation de la progression et des résultats La figure 6.4 est un aperçu de la fenêtre de préparation des requêtes de comparaison. Cette fenêtre est initialement vide. L'utilisateur peut alors cliquer sur le bouton « Open », ce qui va ouvrir une nouvelle fenêtre (que l'on peut voir sur la figure 6.5, lui permettant de choisir un fichier csv. Après validation du fichier à ouvrir, il pourra, là-aussi, visualiser son contenu, avant de valider. Lorsque c'est fait, une ligne apparaît, telle que sur la figure 6.4. Cette dernière contient quatre colonnes.

- La colonne `Jobs Names` assigne un nom, à chaque ligne de requête, basé sur le nom du fichier.
- La colonne `Files` rappelle les chemins vers les fichiers qui ont été chargés, afin que l'utilisateur puisse vérifier, qu'il ne s'est pas trompé (malgré la vérification juste après la sélection du fichier).
- La colonne `Parameters` contiens un bouton « configure ». Ce bouton n'a pas encore été configuré, il ne fais donc rien pour l'instant. Il a été placé là, dans le but de servir à une

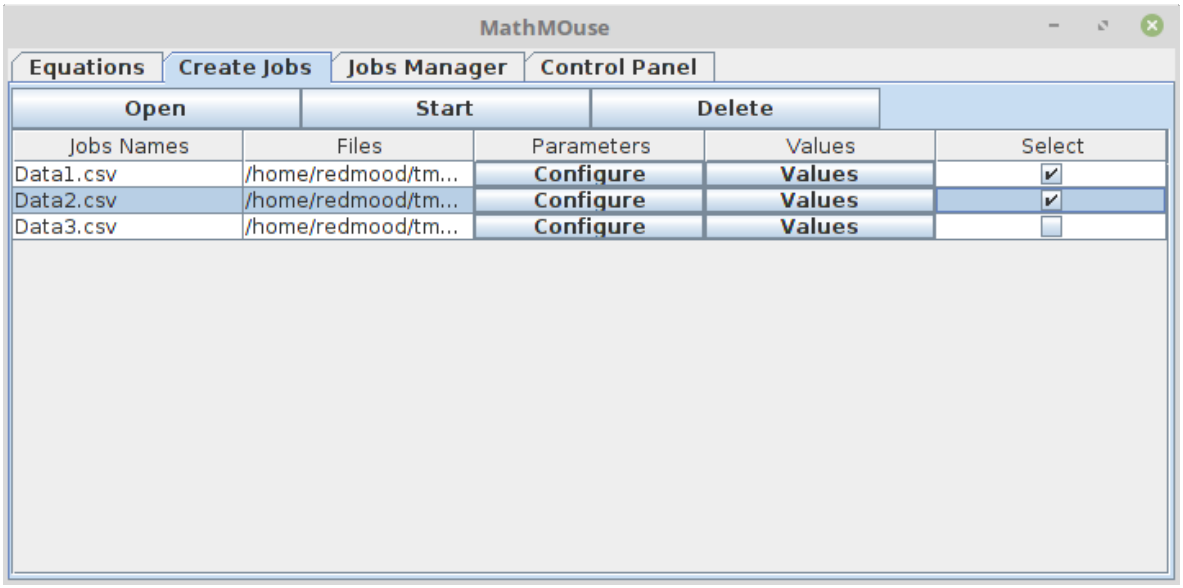


FIGURE 6.4 – Aperçu de la fenêtre de préparation de requêtes de comparaison

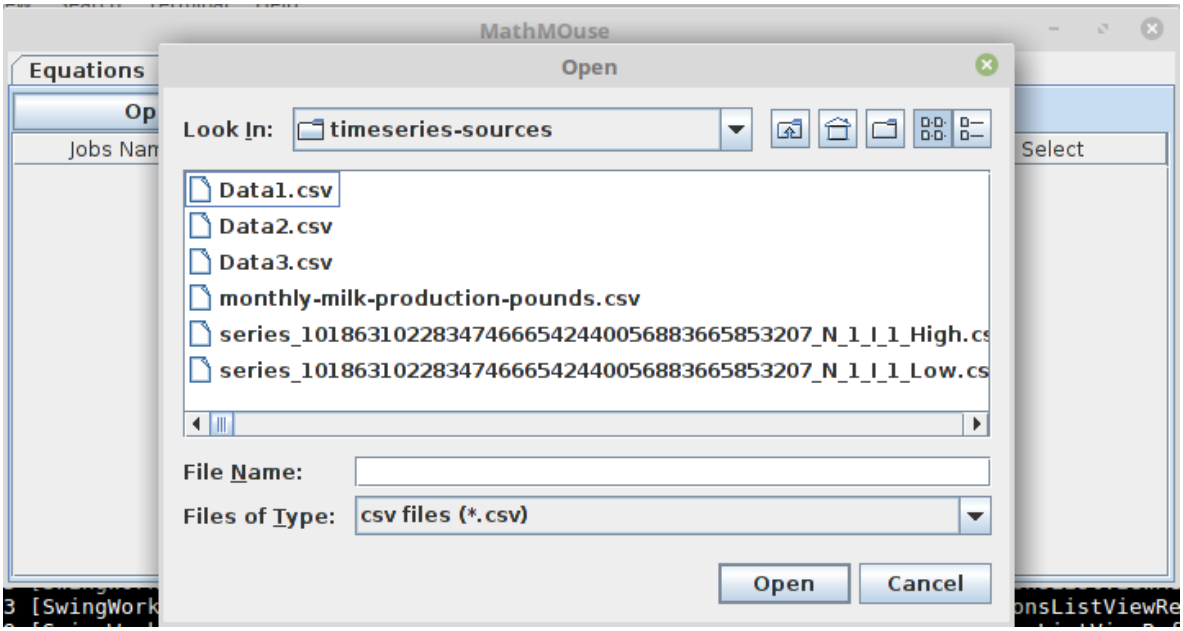


FIGURE 6.5 – Sélecteur de série chronologique

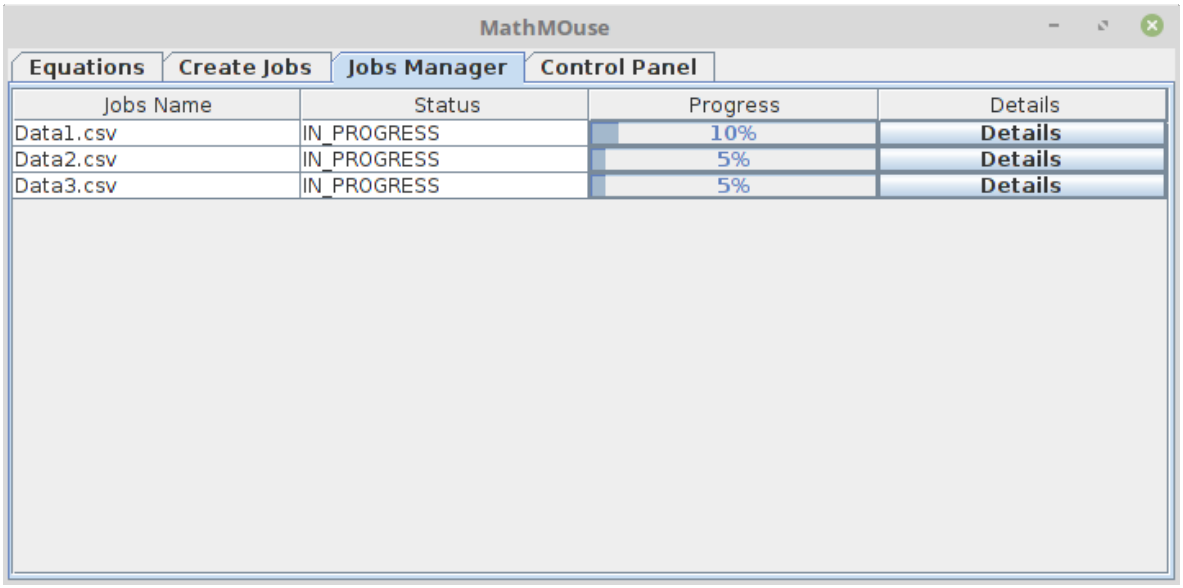


FIGURE 6.6 – Aperçu de la fenêtre d'avancement des opérations de comparaison

éventuelle amélioration, qui consisterait à permettre à l'utilisateur de configurer sa requête de comparaison. Cela lui permettrait, par exemple de choisir les algorithmes de comparaison et/ou de génération, qu'il souhaite utiliser leur de l'opération. Ou encore de préciser un sous-ensemble d'équation avec lesquelles il veut faire sa comparaison (par exemple, les équations d'ordre un, ou les équations d'un groupe en particuliers, ou les équations sans second membre, i.e dont la fonction d'entrée est la fonction nulle...).

- La colonne `Values` permet de ré-afficher les valeurs (en cas de doute de dernier instant).
- La colonne `Select` doit être cochée, pour que la requête soit effectivement envoyée à la base de modèles.

Et afin de réaliser cet envoi effectif, l'utilisateur devra cliquer sur le bouton « Start ». Le bouton « Delete », quant à lui, permet, au contraire, de supprimer les opérations cochées.

Une fois les opérations lancées, l'utilisateur peut aller vérifier leur progression dans la fenêtre d'avancement. Chaque opération possède un statut (colonne `Status`).

- L'opération est en attente, `WAITING`, lorsque l'opération a été envoyée au module manager des opérations de comparaison et que l'interface attend un retour de ce dernier, lui confirmant le démarrage de l'opération.
- Elle passe en phase de démarrage, `STARTING`, lorsque le module manager envoie une demande de génération au module correspondant et attend que ce dernier renvoie les données. Le générateur envoie les séries expérimentales, au fur à mesure qu'il les génère. C'est-à-dire que, s'il récupère dix modèles dans la base de données, il génère la série correspondant au premier modèle, l'envoie au comparateur, puis génère la suivante et ainsi de suite.
- L'opération est dite en cours, `IN_PROGRESS`, lorsque la première série expérimentale, issue du générateur, est reçue. Ainsi, les opérations de comparaison sont réalisées au fur et à mesure que les séries sont reçues, pour tirer avantage de la parallélisation des calculs.

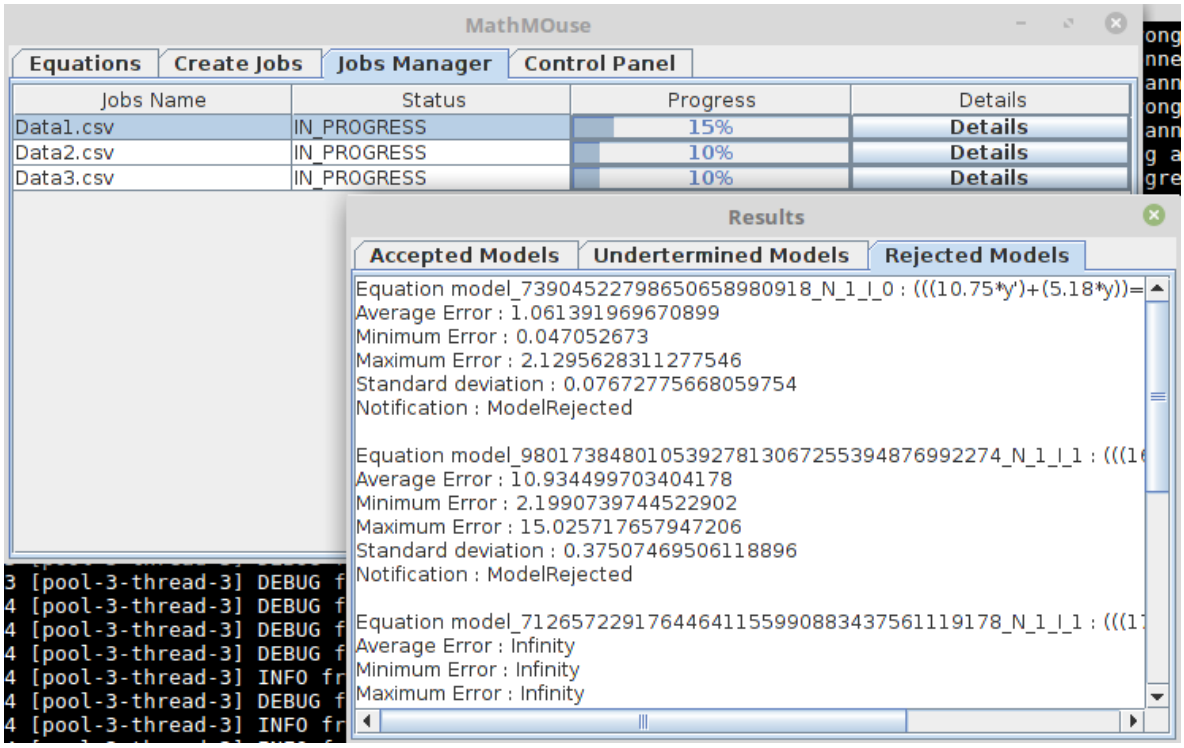


FIGURE 6.7 – Affichage des résultats d'une comparaison

- L'opération est terminée, `OVER`, lorsque toutes les comparaisons sont effectuées et que tous les résultats ont été reçus par l'interface.

La colonne `Progress` permet de visualiser l'état d'avancement de chacune des opérations et le bouton « Details » dans la colonne du même nom, permet, comme on peut le voir sur la figure 6.7, d'ouvrir une fenêtre, dans laquelle l'utilisateur peut voir les résultats des comparaisons, qui arrivent eux aussi, au fur et mesure que les comparaisons sont effectuées. L'utilisateur n'a pas à attendre la fin des opérations, pour commencer à regarder les résultats. La fenêtre des résultats est divisée en trois onglets : `Accepted Models` ; `Undetermined Models` ; et `Rejected Models`. Ainsi, les modèles sont regroupés selon s'ils ont été acceptés, rejetés, ou s'ils sont dans le cas d'indétermination. Les résultats affichés permettent de connaître : (i) le nom de l'équation concernée et sa formule ; (ii) l'erreur moyenne (ici, la moyenne du carré des erreurs) ; (iii) la valeur minimale de la série des erreurs, ainsi que sa valeur maximale ; (iv) l'écart type ; (v) et la notification associée, bien que techniquement redondante.

6.2 Tests expérimentaux

6.2.1 Expérience de contrôle

Pour vérifier que la base de modèle est bien en mesure de détecter des séries chronologiques correspondant à un certain modèle et de rejeter, les autres modèlent, nous avons utilisé un

set de trois exemples simples, fournit par l'équipe d'automatique du laboratoire.

Exemple 6.1
Il s'agit de l'équation que nous avons utilisée en exemple, tout le long de cette thèse. L'équation est la suivante :

$$Ty' + y = Ku \tag{6.1}$$

Où :

- $T = 25s$;
- $K = 2,4$ *(K est un gain sans unité)* ;
- *la fonction u, observable sur la figure 6.8a, alterne entre les valeurs* 1 *et* -1 *(bien que la courbe de la figure 6.8a soit continues, les valeurs sont de u sont bien discrètes et égales à uniquement à* 1 *ou* -1*)* ;
- *le pas de temps est de* $1s$;
- *la série expérimentale contiens milles valeurs de* $0s$ *à* $999s$.

La figure 6.8b contiens deux courbes (bien que ça ne voit pas très, car elles sont superposées), qui représentent (en ligne discontinue) la solution de l'équation calculées avec l'algorithme de Runge-Kutta, décrit dans la section 5.2.1 et en ligne continue (sous la première) les données expérimentales d'exemple fournies par l'équipe d'automatique. L'exemple étant parfaitement calibré, pour que les courbes correspondent quasi-parfaitement, d'où leur superposition.

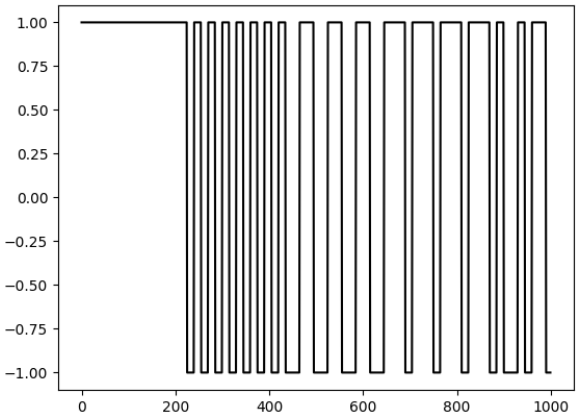
Exemple 6.2
Cet exemple est basé sur la même formule que l'exemple 6.1, mais avec des paramètres différents :

- $T = 2s$;
- $K = 1,3$ *(K est un gain sans unité)* ;
- *la fonction u, observable sur la figure 6.9a, possède des variations plus complexes que dans l'exemple 1* ;
- *le pas de temps est de* $0,1s$;
- *les données expérimentales contiennent mille valeurs de* $0s$ *à* $99,9s$.

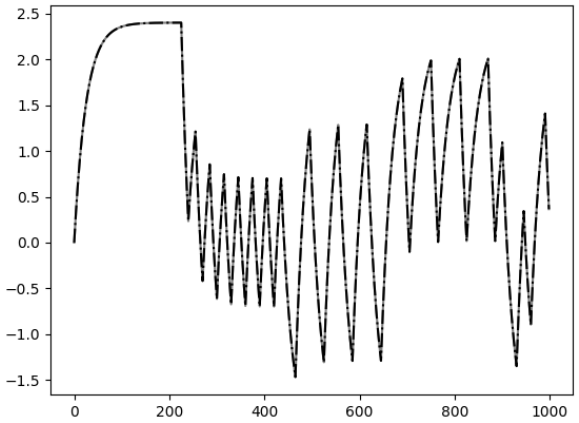
Comme pour l'exemple 1, les courbes expérimentales et théoriques sont représentées sur la figure 6.9b et la proximité des valeurs, fait que les courbes sont superposées sur la figure.

Exemple 6.3
Cet est une équation du second ordre :

$$y^{(2)} + 2m\omega_0 y^{(1)} + \omega_0^2 y = Ku \tag{6.2}$$

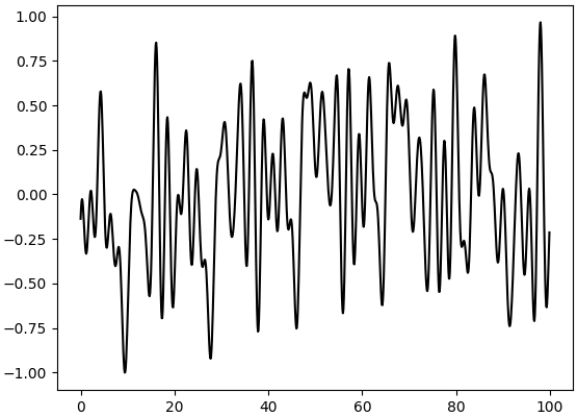


(a) Fonction u

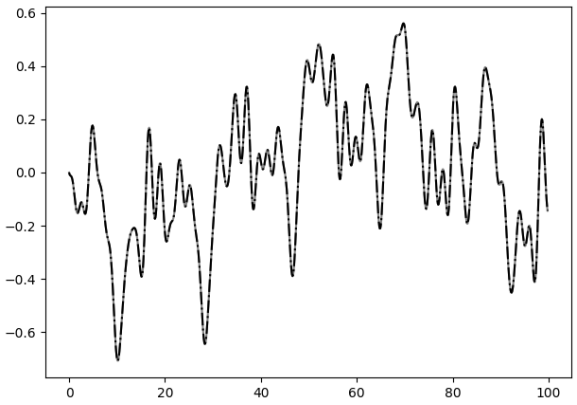


(b) Séries expérimentales et théoriques

FIGURE 6.8 – Exemple 1

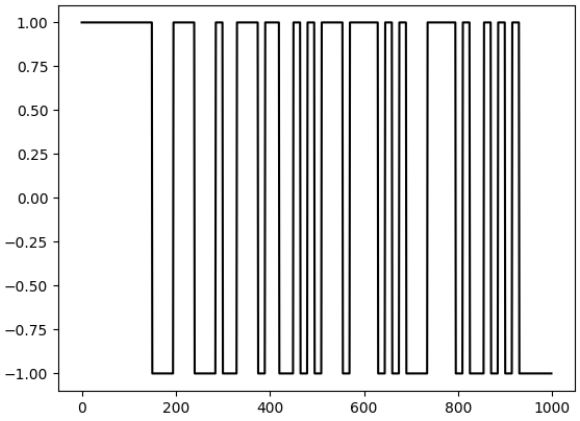


(a) Fonction u

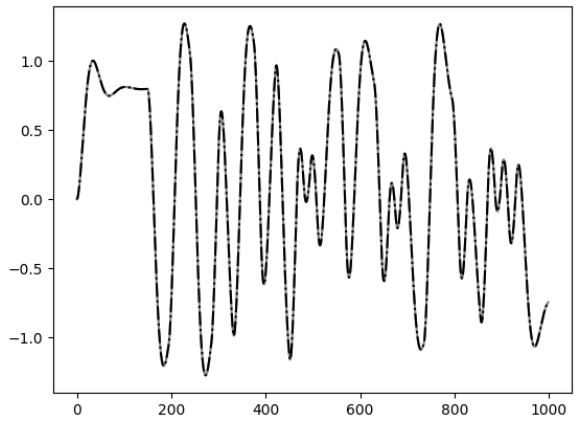


(b) Séries expérimentales et théoriques

FIGURE 6.9 – Exemple 2



(a) Fonction u



(b) Séries expérimentales et théoriques

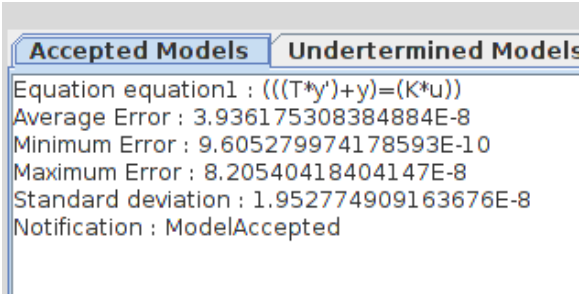
FIGURE 6.10 – Exemple 3

Avec :

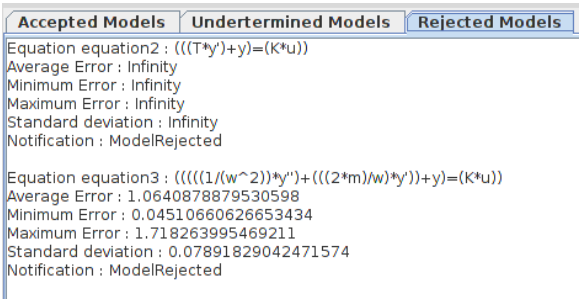
- $K = 0,8$ *;*
- $m = 0,4s$ *;*
- $\omega_0 = 0,1$ *rad/s ;*
- *la fonction u alterne également entre des valeurs égales à* 1 *ou* -1*, observable sur la figure 6.10a ;*
- *le pas de temps est de* $1s$ *et la séries possède deux mille valeurs de* $0s$ *à* $1999s$*.*

Comme l'espace disponible pour tracer les courbes est limitée par la place disponible sur la page, nous n'avons tracé que les mille premières valeurs, aussi bien sur la figure 6.10a que sur la figure 6.10b. Sur la figure 6.10b, les courbes théoriques et expérimentales sont, encore une fois, superposées.

Après avoir ajouté les trois modèles dans la base, nous avons testé la comparaison de chacune des séries expérimentale, contre l'ensemble des trois modèles, pour s'assurer que la base de



(a) Acceptation de l'équation 1



(b) Rejet des équations 2 et 3

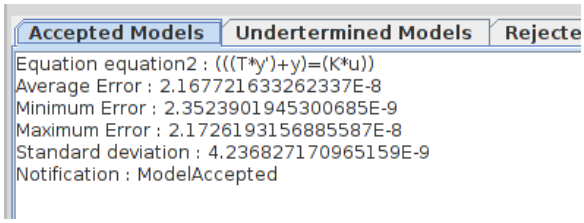
FIGURE 6.11 – Résultat pour l'équation 1

modèles renvoie des résultats cohérents. Les figures 6.11, 6.12 et 6.13 montrent les résultats obtenus pour chacun des exemples. Nous pouvons voir que chaque série a bien été associée à son équation correspondante, tandis que les deux autres ont été rejetées.

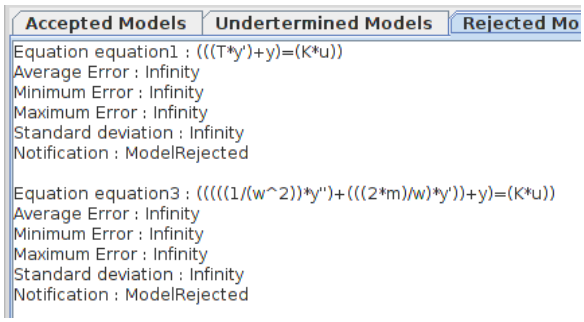
6.2.2 Test de reconnaissance, sur une base de 100 modèles

Création du jeu de données Pour avoir une base de cents modèles, nous avons choisi de les générer aléatoirement. À l'aide d'un script Python, nous pouvons choisir aléatoirement un ordre (limité à 5), puis les valeurs des coefficients de l'équation, puis le nombre de fonction d'entrées (de zéro à deux), puis les valeurs des conditions initiales. Enfin, un second programme, en Java, car il s'appuie sur notre implémentation Java de l'algorithme de Runge-Kutta, permet d'éliminer les équations qui s'avère non-solvable, ou constamment nulle, ou qui diverge à l'infini trop rapidement (nous avons pris une base de 5000 points maximum, pour un pas de temps de une seconde, les équations dont la résolution donnent des valeurs non-représentables informatiquement, avant d'atteindre les 5000 points, sont éliminées). Nous avons généré 800 équations, le processus d'élimination en as éliminé la moitié, puis nous avons gardé une centaine d'entre eux, choisit aléatoirement.

Les résultats observés Après avoir crée notre jeu de cents modèles, nous les avons ajoutés à la base de modèles, à l'aide du module `Populator`, qui avait été prévu à cet effet. Puis nous avons généré un ensemble de série expérimentales, à partir des modèles de la base de données. Ces séries sont les résolutions des modèles, que l'on a brouillé en introduisant plus ou moins

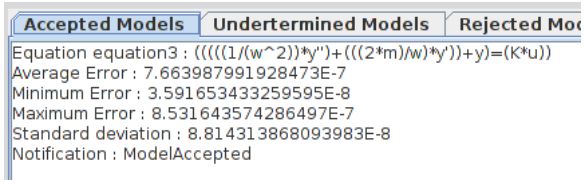


(a) Acceptation de l'équation 2

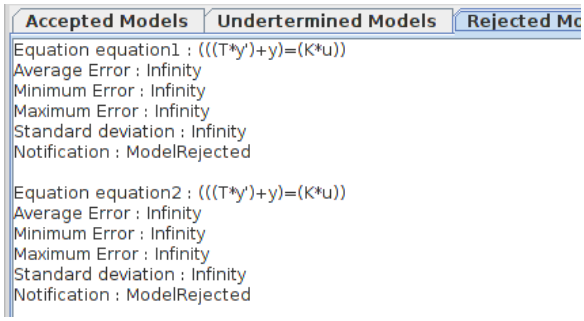


(b) Rejet des équations 1 et 3

FIGURE 6.12 – Résultat pour l'équation 2



(a) Acceptation de l'équation 3



(b) Rejet des équations 1 et 2

FIGURE 6.13 – Résultat pour l'équation 3

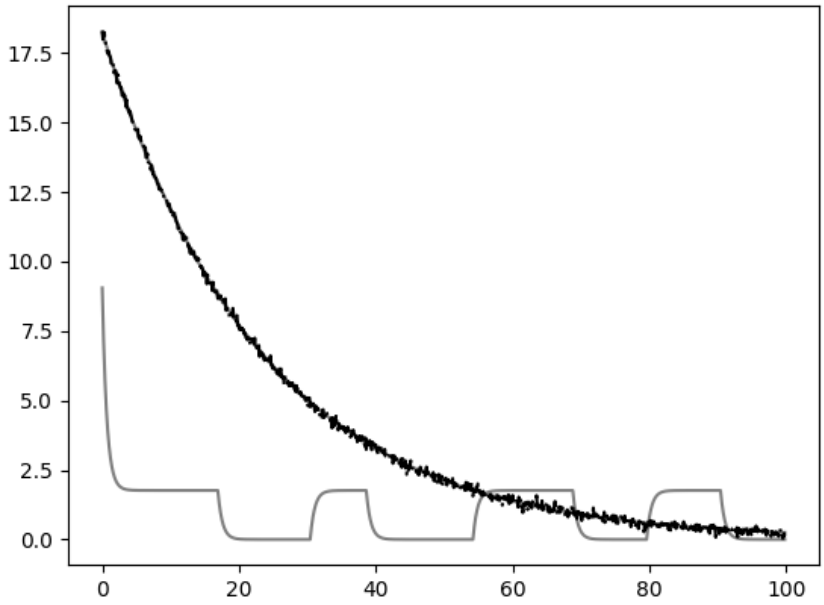


FIGURE 6.14 – Résultat du test sur cent modèles

de bruit dans les résultats. La figure 6.14 contiens une courbe en noires, qui représente une des séries expérimentales générées et deux courbes grises (l'une étant peu visible sous la courbe noire). Il s'agit de deux courbes tracées à partir des modèles. Celle qui se superpose à la courbe noire correspond à un modèle, qui a été rangé dans la catégorie `Undetermined`, l'autre à un modèle, qui a été rejeté.

Les paramètres d'acceptation étant :

- Seuil maximale de la moyenne des erreurs quadratiques (erreurs au carré, voir section 5.3.1) : 1 ;
- Seuil minimale de la moyenne des erreurs quadratiques : 1 ;
- Écart type maximale de la série des erreurs quadratiques : 10^{-3} ;
- Écart type minimale de la série des erreurs quadratiques : 10^{-3}.

Le modèle indéterminé possédait les caractéristiques suivantes :

- Nom et formule de l'équation : model_73909619612425749897640348103411570988_N_1_I_1 : $13,27y' + 0,57y = 3,95u_0$;
- Moyenne des erreurs quadratiques : $0,07746691034994661$;
- Écart type : $0,06364983292592044$.

Le modèle rejeté :

- Nom et formule de l'équation : model_869129233400038195239156636958373442 17_N_1_I_1 : $8,88y' + 13,53y = 15,49u_0$;
- Moyenne des erreurs quadratiques : $7,172225804499686$;
- Écart type : $0,457865380173926$.

Bien que le tracé de la courbe, montre que le modèle trouvé semble, malgré le bruit, correspondre au modèle, ce dernier n'a pas été accepté. Cela peut provenir des valeurs qui sont peut-être trop strictes. Ces dernières sont des paramètres de l'algorithme de comparaison, de sorte qu'il est possible d'implémenter une méthode permettant à l'utilisateur de préciser lui-même les paramètres qu'il juge acceptables.

6.2.3 Évaluation des temps de calculs et du potentiel d'optimisation

La troisième expérience consistait à évaluer les gains possibles en efficacité, en utilisant la parallélisation des calculs. Les modules de comparaisons et de générations étant les deux modules lourds en calculs, nous avons testé le système de multiplication d'instances de services, pour paralléliser les calculs. Nous avons testé neuf configurations :

- 1 comparateur, 1 générateur ;
- 1 comparateur, 3 générateurs ;
- 1 comparateur, 5 générateurs ;
- 3 comparateurs, 1 générateur ;
- 3 comparateurs, 3 générateurs ;
- 3 comparateurs, 5 générateurs ;
- 5 comparateurs, 1 générateur ;
- 5 comparateurs, 3 générateurs ;
- 5 comparateurs, 5 générateurs.

La figure 6.15 contient trois courbes, qui résument les temps mesurer. Précisons d'abord, que nous avons mesuré le temps d'exécution, dit de bout-en-bout. Autrement dit, le temps écoulé entre le moment où l'utilisateur appui sur le bouton « Start » de l'interface, pour lancer l'opération de comparaison et le moment où cette dernière bascule vers son état « Over », c'est-à-dire quand tous les modèles disponibles ont été comparés à la série expérimentale en cours et que tous les résultats ont été reçus par l'interface, de sorte que l'utilisateur puisse les consulter. Ensuite, la figure 6.15 contient trois courbes, qui montrent l'évolution du temps de calcul, en fonction du nombre de comparateurs, pour un nombre fixé de générateurs. Chacune des trois courbes correspond à un nombre de générateurs donné.

Nous pouvons remarquer que de manière générale, il y a beaucoup à gagner, à tirer avantage du calcul en parallèle. Avec une réduction par au moins trois, du temps de calcul, par ajout de générateurs et une réduction par quatre, lorsque pour un nombre de générateurs suffisant, on introduit plus de comparateurs. En effet, ici, pour un ou trois générateurs, par exemple, augmenter le nombre de comparateur n'apporte rien. À partir de cinq générateurs, passer de un à trois comparateurs entraîne une réduction du temps de calcul. En revanche, lorsque que l'on continue d'augmenter le nombre de comparateur, cela fini par ne plus rien apporter, à nouveau. Ceci met également en relief, l'utilité de pouvoir multiplier une fonctionnalité particulière (un microservice) du code, sans multiplier tout le code. L'exemple ici, nous montre, qu'il n'est pas forcément très intéressant d'avoir autant de comparateurs, que de générateurs.

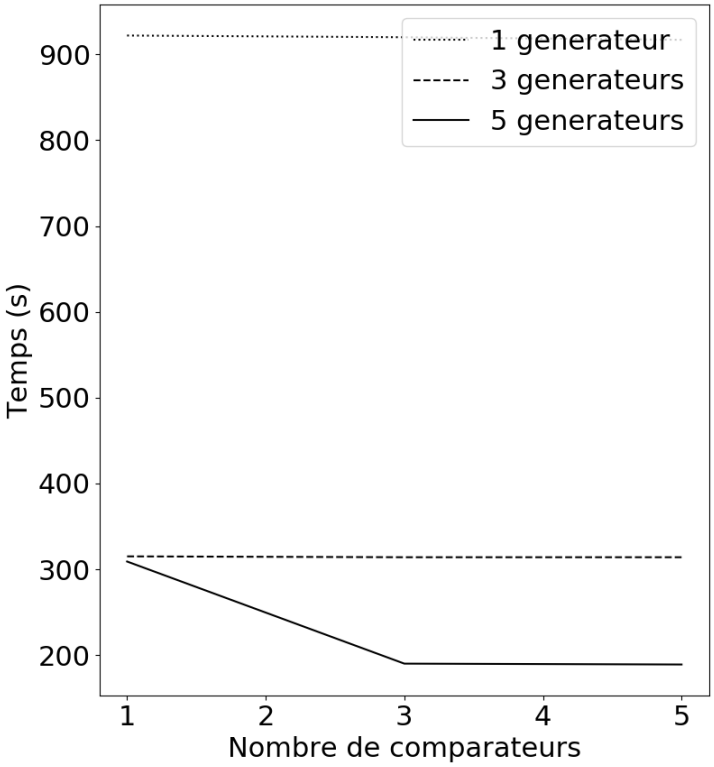


FIGURE 6.15 – Temps d'exécution pour cent modèles

Remarque Les temps d'exécution observés ont été obtenus sur une machine windows, avec tous les services tournant localement sur la même machine. La machine est équipée d'un processeur Intel i5-4690 CPU cadencé à 3.5 GHz, de 8 GB de RAM et un disque de 1 TB.

6.3 Conclusion

Dans ce chapitre, nous avons vu l'implémentation d'un prototype de base de modèles. Ce dernier a été implémenté en Java, avec une architecture microservices. L'architecture est supportée par la plate-forme docker, qui gére la création d'environnements unique et isolé, pour chaque service (conteneur) et possibilité de multiplier (scale) un même service et/ou de le déplacer vers une autre machine connectées. Une interface graphique jouant le rôle d'application clientes a également été implémentée. Son rôle est de démontrer l'accessibilités des différentes fonctionnalités de la base de modèles, ainsi que les propriétés du prototype implémenté.

Une première série de tests a été réalisée sur un jeu de trois équations, qui étaient des exemples classiques d'équations utilisées dans l'automatique. Puis nous avons construit un jeu de cents équations, de façon à tester le comportement du prototype sur jeu de données plus conséquent. Ce qui a également permis d'avoir un aperçu du potentiel d'optimisation et de passage à l'échelle, basée sur les fonctionnalités offertes par Docker et l'utilisation de microservices.

Cependant, le prototype présenté en est encore à une version très basique, qui pourrait être améliorée de bien des façons. Parmi les améliorations possibles, en voici quelques-unes, qui nous semblent intéressantes.

(i) Apporter plus de contrôle à l'utilisateur. En lui donnant notamment la possibilité de fournir son propre jeu de paramètre, ou de redéfinir partiellement certains paramètres et notamment les fonctions d'entrées. En effet, pour l'instant, les fonctions d'entrées utilisées sont celles fournies avec le modèle, lors de son insertion, mais ce comportement par défaut n'est pas le plus rationnel. En effet, si l'utilisateur sait qu'au cours de son expérience, il a utilisé une fonction d'entrées particulière, alors même s'il existe un modèle qui correspond à ses données, dans la base de modèles, ce-dernier ne sera pas détecté, si la fonction d'entrée par défaut, n'est pas la même que celle que l'utilisateur a utilisée. En plus de la fonction d'entrée, il serait intéressant de fournir à l'utilisateur le moyen de choisir les algorithmes de comparaison, qu'il souhaite utilisé et d'en définir les paramètres (tels que les valeurs seuils), le cas échéant. Il serait également intéressant, que l'utilisateur puisse choisir son algorithme de génération. Notamment, car cela permettrait d'envisager une plus grande ouverture du type de modèle.

(ii) La possibilité de stoker les fonctions d'entrées sous forme symbolique. Cela pourrait à la fois permettre d'utiliser moins d'espace de stockage et de pouvoir s'affranchir des méthodes d'interpolation (le cas échéant). Mais cela offrirait surtout plus de souplesse dans la résolution des équations. En effet, à l'heure actuelle, si une équation dépend d'une série u pour être résolue, alors nous ne pouvons résoudre l'équation, que pour u. C'est-à-dire, que si u est définie sur un intervalle de 1000s, nous ne pouvons résoudre sur un intervalle temporel qui dépasse 1000s. Aussi, si u est découpée en valeurs extraites toutes les 1s, alors nous ne pouvons affiner à 1 ms, par exemple, où choisir un pas de temps de 1.25s. C'est-à-dire que la série u impose de fait, un certains nombres de paramètres, que l'utilisateur pourrait vouloir modifier, mais ces modifications ne peuvent être supportées par l'implémentation actuelle.

(iii) Un dernier point intéressant, concerne la définition du format XML, pour la représentation des équations. Il s'agit d'une représentation « maison », bien que nous nous soyons intéressé de très près et donc inspiré du standard PMML [GZLW09, Pec09], lui aussi basé sur langage XML. Ce dernier supporte notamment la représentation des séries chronologiques, depuis sa version 4.0. L'intérêt ici, serait de s'appuyer sur le standard, pour permettre à l'utilisateur de plus facilement intégrer la base de données à d'autres logiciels ou librairies (tel que Scikit-Learn [6] et Sklearn [7] en Python).

6. `http://scikit-learn.org/stable/index.html`
7. `http://pypi.python.org/pypi/sklearn-pmml/0.1.0`

Troisième partie

Conclusion et perspectives

Chapitre 7

Conclusion & Perspectives

Dans ce chapitre, nous faisons un bilan des travaux effectuer durant cette thèse, puis nous donnerons un ensemble de perspectives pour poursuivre ces travaux.

7.1 Contributions

Nos travaux nous ont permit d'amener les contributions qui suivent.

États de l'art : Tout d'abord, le premier chapitre de l'état de l'art nous a permit de constater l'évolution dont ont bénéficié les bases de données en général. Dans un second chapitre, nous nous sommes davantage intéressé aux usages que font les scientifiques des bases de données, en prenant un type de données couramment utilisé en sciences, les séries chronologiques. Nous nous sommes donc penchés sur les bénéfices que les systèmes de stockage et gestion des séries chronologiques ont su tirer des autres technologies de stockages. Nous avons vu qu'il a fallu développé des outils spécifiques pour stocker et gérer correctement des séries chronologiques. Mais nous avons vu aussi, qu'il y a un besoin chez les scientifiques d'échanger et de manipuler des modèles mathématiques et de les mettre en lien avec des données expérimentales, c'est-à-dire avec des séries chronologiques. Nous nous sommes penchés sur les outils existants, qui ont pour objectif de couvrir ce besoin et nous avons constaté qu'il s'agit d'un terrain encore peu abordé dans la communauté base de données.

Conception d'une base de modèles et prototypage : Tout d'abord, dans le chapitre 4, nous nous somme attardés sur la définition des données nécessaires à notre base de modèles. Nous avons détaillé la définition d'une équation différentielle, pour en identifier les propriétés et en proposer une modélisation informatique. Nous avons également précisé que dans le cadre de notre projet, nous avons besoin que l'équation soit solvable d'un point de vue numérique et pas uniquement théorique (c'est-à-dire par calcul symbolique). Pour cela, nous avons défini un ensemble de paramètres nécessaires pour satisfaire ce besoin et nous les avons intégrés à notre modélisation d'une équation différentielle. Cela nous a permit de construire notre modélisation, en définissant les structures de données nécessaire à leur implémentation dans un programme informatique. Nous avons donc défini, une structure d'arbre pour représenter la formule d'une équation, ainsi que les structures nécessaires pour représenter les différents paramètres utiles aux calculs numériques. Puis nous avons défini un schéma entités-associations,

inspiré de la méthode de modélisation dimensionnelle utilisée dans les bases de données décisionnelles. Enfin, nous avons défini une représentation textuelle au format XML, inspiré du standard PMML utilisé dans le domaine de la fouille de données, afin de permettre d'échanger des équations entre plusieurs programmes.

Une fois que nous avons obtenu notre modèle de données, nous nous sommes intéressé à la couche logicielle, dans le chapitre 5. Nous avons donc défini les processus principaux à développer pour réaliser les différentes fonctionnalités (gestion des insertions/sélections de modèles et exécution de requêtes par les données), que nous souhaitions implémenter dans notre base de modèles. Nous avons identifié deux types de processus : (i) les processus de conversion : il s'agit des processus qui permettent de lire un fichier XML, pour récupérer l'arbre et l'ensemble des paramètres d'une équation et du processus permettant de traduire un arbre en tuples, que l'on peut ensuite insérer dans la base de modèles ; (ii) et les processus de calcul numérique : il s'agit du processus de génération de série chronologique par résolution numérique des équations contenues dans la base de données et du processus de comparaison de deux séries chronologique, qui permet de comparer une série expérimentale fournie par l'utilisateur et les séries, dites théoriques, générées par le biais du processus précédent.

Pour les processus de conversion, nous avons proposé un algorithme de lecture récursive pour les fichiers XML, car les fichiers XML permettent de conserver la structure d'arbre et se prête donc facilement à la récursivité. Pour les conversions vers des tuples, nous sommes passé par des objets intermédiaires, qui ont pour principal objectif « d'aplatir » l'arbre, afin d'avoir une représentation linéaire similaire à des tuples. Pour le processus de génération, nous avons utilisé l'algorithme de Runge-Kutta d'ordre quatre, car il s'agit d'un algorithme classique et largement utilisé pour résoudre des équations différentielles ordinaires. Pour le processus de comparaison entre deux séries, nous avons choisi d'utiliser la série des écarts (calcul de la différence, valeur par valeur). Pour éviter que les écarts négatifs ne se compensent avec les écarts négatifs, nous avons naïvement commencé par considérer la valeur absolue des écarts, mais nous avons ensuite préféré l'élévation au carré (en suivant le conseil du professeur Thierry POINOT, Université de Poitiers).

Après avoir défini les processus, nous avons également proposé une architecture générale pour la construction d'un gestionnaire de modèles. Ce schéma permet de définir les différentes parties principales, que doivent contenir le gestionnaire (une base de modèles, un module de comparaison, un module de génération, des modules de pré-traitement des données d'entrées et de post-traitement, pour préparer les données avant de les mettre à disposition du client), ainsi que l'ordre dans lequel ces modules interagissent entre eux.

Pour finir, dans le chapitre 6, nous avons décrit le prototype que nous avons implémenté. Nous commençons par décrire le découpage en microservice, que nous avons utilisé, inspiré de l'architecture générique proposée à la fin du chapitre précédent. Ce découpage comprend principalement les microservices suivants : (i) la base de données postgresql ; (ii) un microservices de liaison entre la base de données et les autres services ; (iii) un microservices de génération (résolution numérique des équations différentielles) ; (iv) un microservices de comparaison (comparaison entre deux séries chronologiques) ; (v) et un microservices qui gère l'interface utilisateur.

À travers l'interface utilisateur, il est possible d'insérer un nouveau modèle dans la base de données, en fournissant un fichier XML qui en contient les données. L'interface permet aussi de réaliser une requête d'identification d'un modèle à partir d'une série chronologique. Pour cela, il faut fournir la série chronologique, sous la forme d'un fichier csv, à comparer avec les modèles de la base. Ce prototype nous a permis de réaliser des tests à l'aide de trois exemples fournis par nos collègues automaticiens. Les tests ont montré que le prototype était capable de reconnaître chacun des modèles, à partir des séries chronologiques correspondantes, tout en discriminant les modèles non-adéquats. Puis nous avons créé un ensemble de cents modèles, à partir desquelles nous avons générées trois séries par modèles (en introduisant plus ou moins de bruits dans chaque série crée), ce qui donne un total de trois cents séries chronologiques. Cela nous a permit de tester la reconnaissance de modèles dans un ensemble de modèles disponible plus grand et sur des séries chronologiques plus ou moins déformées par du bruit. Nous avons pu constater que le prototype était toujours capable de déterminer un modèle correspondant à la série, bien que les résultats se dégradent, au fur et mesure que le bruit augmente (ce qui fait sens). Nous avons également mesuré le temps nécessaire pour obtenir un résultat, pour une série chronologique en entrée et cents modèles dans la base de modèles et nous avons mesuré la variation de ce temps de réponse du prototype en fonction du nombre d'instances des microservices de comparaison et de générations. Nous avons pu observer que, de manière générale, l'augmentation du nombre d'instances des deux services permettaient un gain de temps significatif, mais qu'il était inutile d'augmenter le nombre d'instances d'un seul service, car le second agit alors comme un goulot de bouteille, empêchant tout gain supplémentaire de performance.

7.2 Perspectives

7.2.1 Un spectre d'équation plus étendu

Dans nos travaux, nous nous sommes limités aux équations différentielles dites : (i) ordinaires : l'équation dépend d'une seule et même variable (en physique, il s'agit généralement du temps), mais il faut noter que des paramètres tel que la température peuvent dépendre du temps et de la position dans l'espace (donc des trois coordonnées x, y et z, par exemple), ce qui fait quatre variables, au lieu d'une seule ; (ii) et linéaire. La raison principale tient au fait qu'il s'agit du type d'équation le plus simple à résoudre numériquement et cela nous a permit de développer un prototype, sans avoir à implémenter des algorithmes de calcul numérique trop complexe. Comme nous l'avons montré sur la figure 4.3 (voir 4.2.2), il est possible de représenter des équations non-linéaires (mais toujours ordinaires), avec la structure de stockage que nous avons proposé. Cependant, pour représenter des équations non-ordinaires, dites aussi aux dérivées partielles, le problème réside dans l'opérateur de dérivation des fonctions. Pour une fonction d'une variable, la dérivation se fait, par défaut, par rapport à l'unique variable. Dans le cas des fonctions de plusieurs variables, il n'est pas possible de dériver une fonction sans préciser par rapport à quelle variable il faut dériver. Or, dans la structure que nous avons proposée, nous n'avons pas prévu de mécanisme de précision de la variable de dérivation.

7.2.2 Développer un langage « SQL-like » spécifique aux bases de modèles

La fonctionnalité de « requête par les données » (qui repose sur une opération de comparaison entre des séries chronologiques fournies par l'utilisateur et des séries chronologiques théoriques, générées à partir des modèles stockés dans la base de modèles), que nous avons développé dans ces travaux n'est pas prévu dans les fonctionnalités du langage SQL. Nous avons également vu, dans l'état de l'art, que le langage SQL n'est pas adapté à la manipulation de séries chronologique. Ainsi, le développement d'un langage basé sur le langage SQL ou un des langages développés pour les gestionnaires de séries chronologiques (voir section 3.1.3) pourra faciliter l'interaction entre l'utilisateur et la base de modèle.

7.2.3 Le passage à l'échelle : gestion de gros volumes de données

Dans la partie expérimentale, nous avons pu tester notre prototype, d'abord à l'aide de trois exemples simples, fournit par l'équipe automatique du LIAS, puis à l'aide de séries chronologiques et de modèles générés aléatoirement. Cependant, la quantité de modèles et séries chronologique utilisées (100 modèles et une centaine de séries d'entrées) représentent un total de $6,98$ Mo, ce qui est très face aux volumes mentionnées dans la littérature scientifique (voir section 2.3). Nous avons conçu le prototype à l'aide d'une architecture microservices, dans l'objectif de pouvoir le faire fonctionner dans un environnement distribué, ce qui devrait aider à gérer un volume de données beaucoup plus significatif, mais nous n'avons pas eu l'occasion de le vérifier. Il serait intéressant d'étudier le comportement de l'approche avec des volumes de données beaucoup plus lourds, mais aussi, avec un plus gros volume de données issues de cas réels et fournit par des spécialistes de disciplines diverses.

Quatrième partie

Annexes

Annexe A

Résolution d'une équation différentielle linéaire d'ordre 2

Nous avons pris comme exemple l'équation suivante :

$$\forall t \in \mathbb{R}, f(t) - f^{(2)}(t) = 0 \tag{A.1}$$

On peux réécrire l'équation de la manière suivante (en simplifiant un peu l'écriture) :

$$f^{(2)} - f = 0 \tag{A.2}$$

Il s'agit d'une équation différentielle linéaire d'ordre deux, dont la forme générale est la suivante :

$$af^{(2)} + bf^{(1)} + cf = 0 \tag{A.3}$$

Avec $a = 1$, $b = 0$, $c = -1$. D'après [PM08, pp. 35-36], on peut résoudre cette équation en se basant sur le calcul du déterminant Δ, tel que définit pour les équations du second degré, de la forme : $ax^2 + bx + c = 0$ où x est une inconnue réelle. Cette équation est dite équation caractéristique de l'équation A.2. Ici, on as donc : $\Delta = b^2 - 4ac = 0 - 4 \times 1 \times (-1) = 4$. On as $\Delta > 0$, donc l'équation caractéristique possède deux racines réelles distinctes, que l'on notera α et β et qui se calcul de la façon suivante :

$$\begin{cases} \alpha = \frac{-b+\sqrt{\Delta}}{2a} = \frac{0+\sqrt{4}}{2\times 1} = 1 \\ \beta = \frac{-b-\sqrt{\Delta}}{2a} = \frac{0-\sqrt{4}}{2\times 1} = -1 \end{cases} \tag{A.4}$$

Les racines ci-dessus permettent de calculer la solution générale de l'équation A.2 :

$$\forall t \in \mathbb{R}, f(t) = K_1 e^{\alpha t} + K_2 e^{\beta t} = K_1 e^{t} + K_2 e^{-t} \tag{A.5}$$

Si on pose $f(0) = 1$, on as alors :

$$\begin{array}{ll} & f(0) = 1 \\ \Longrightarrow & K_1 e^0 + K_2 e^0 = 1 \\ \Longrightarrow & K_1 + K_2 = 1 \qquad (1) \end{array} \tag{A.6}$$

Ainsi, on as besoin d'une seconde condition, qui consiste généralement à fixer une valeur d'origine pour les dérivées d'ordre supérieur. On peux prendre $f^{(1)}(0) = 0$, ce qui donne (sachant que $f^{(1)}(t) = K_1 e^t - K_2 e^{-t}$) :

$$\begin{array}{ll} & f^{(1)}(0) = 0 \\ \Longrightarrow & K_1 e^0 1 K_2 e^0 = 0 \\ \Longrightarrow & K_1 - K_2 = 0 \qquad (2) \end{array} \tag{A.7}$$

Ainsi :

$$\begin{array}{l} (1) - (2) \implies 2K_2 = 1 \implies K_2 = \frac{1}{2} \\ (1) + (2) \implies 2K_1 = 1 \implies K_1 = \frac{1}{2} \end{array} \tag{A.8}$$

Annexe B

XML Schema File

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="differential-equation" type="differentialEquationType" />
<xs:element name="formula" type="formulaType" />
<xs:element name="binary-operator" type="binaryOperatorType" />
<xs:element name="unary-operator" type="unaryOperatorType" />
<xs:element name="function" type="functionType" />
<xs:element name="number" type="numberType" />
<xs:element name="variable" type="variableType" />
<xs:element name="parameters-sets" type="parametersSetsType" />
<xs:element name="parameters-set" type="parametersSetType" />
<xs:element name="initial-values" type="initialValuesType" />
<xs:element name="initial-value" type="initialValueType" />
<xs:element name="variables-values" type="variablesValuesType" />
<xs:element name="variable-value" type="variableValueType" />
<xs:element name="input-functions" type="inputFunctionsType" />
<xs:element name="input-function" type="inputFunctionType" />

<xs:complexType name="differentialEquationType">
  <xs:sequence>
    <xs:element ref="formula" />
    <xs:element ref="parameters-sets" />
  </xs:sequence>
  <xs:attribute name="name" type="xs:NMTOKEN" use="required" />
  <xs:attribute name="group" type="xs:token" use="optional" default="Default"
      />
</xs:complexType>

<xs:complexType name="formulaType">
  <xs:sequence>
    <xs:element ref="binary-operator" />
  </xs:sequence>
  <xs:attribute name="numberOfVariables" type="xs:integer" use="required" />
  <xs:attribute name="numberOfInputFunctions" type="xs:integer" use="required
      " />
  <xs:attribute name="outputName" type="xs:NMTOKEN" use="required" />
```

```
</xs:complexType>

<xs:complexType name="binaryOperatorType">
  <xs:sequence>
    <xs:choice minOccurs="2" maxOccurs="2">
      <xs:element ref="binary-operator"></xs:element>
      <xs:element ref="unary-operator"></xs:element>
      <xs:element ref="function"></xs:element>
      <xs:element ref="number"></xs:element>
      <xs:element ref="variable"></xs:element>
    </xs:choice>
  </xs:sequence>
  <xs:attribute name="value" type="binaryOperatorEnum" />
</xs:complexType>

<xs:simpleType name="binaryOperatorEnum">
  <xs:restriction base="xs:token">
    <xs:enumeration value="-" />
    <xs:enumeration value="+" />
    <xs:enumeration value="*" />
    <xs:enumeration value="/" />
    <xs:enumeration value="=" />
    <xs:enumeration value="&radic;" />
    <xs:enumeration value="^" />
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="unaryOperatorType">
  <xs:sequence>
    <xs:choice>
      <xs:element ref="binary-operator"></xs:element>
      <xs:element ref="unary-operator"></xs:element>
      <xs:element ref="function"></xs:element>
      <xs:element ref="number"></xs:element>
      <xs:element ref="variable"></xs:element>
    </xs:choice>
  </xs:sequence>
  <xs:attribute name="value" type="unaryOperatorEnum" />
</xs:complexType>

<xs:simpleType name="unaryOperatorEnum">
  <xs:restriction base="xs:token">
    <xs:enumeration value="!" />
    <xs:enumeration value="&radic;" />
```

```
    <xs:enumeration value="-" />
    <xs:enumeration value="||" />
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="functionType">
  <xs:choice minOccurs="0">
    <xs:element ref="binary-operator" />
    <xs:element ref="unary-operator" />
    <xs:element ref="function" />
    <xs:element ref="number" />
    <xs:element ref="variable" />
  </xs:choice>
  <xs:attribute name="value" type="xs:NMTOKEN" use="required" />
  <xs:attribute name="deriv" type="xs:integer" use="required" />
  <xs:attribute name="role" type="roleEnum" use="optional" />
</xs:complexType>

<xs:simpleType name="roleEnum">
  <xs:restriction base="xs:NMTOKEN">
    <xs:enumeration value="equationInput" />
    <xs:enumeration value="equationOutput" />
    <xs:enumeration value="basicMathFunction" />
    <xs:enumeration value="basicUserDefinedFunction" />
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="numberType">
  <xs:attribute name="value" type="xs:double" use="required" />
</xs:complexType>

<xs:complexType name="variableType">
  <xs:attribute name="value" type="xs:NMTOKEN" use="required" />
</xs:complexType>

<xs:complexType name="parametersSetsType">
  <xs:sequence>
    <xs:element ref="parameters-set" maxOccurs="unbounded" />
  </xs:sequence>
  <xs:attribute name="numberOfParametersSets" type="xs:integer" use="required
      " />
</xs:complexType>

<xs:complexType name="parametersSetType">
```

```
122   <xs:sequence>
123     <xs:element ref="initial-values" />
124     <xs:element ref="variables-values" />
125     <xs:element ref="input-functions" />
126   </xs:sequence>
127 </xs:complexType>
128 
129 <xs:complexType name="initialValuesType">
130   <xs:sequence>
131     <xs:element ref="initial-value" maxOccurs="unbounded" />
132   </xs:sequence>
133   <xs:attribute name="numberOfInitialValues" type="xs:integer" use="required"
        />
134 </xs:complexType>
135 
136 <xs:complexType name="initialValueType">
137   <xs:attribute name="name" type="xs:NMTOKEN" />
138   <xs:attribute name="value" type="xs:double" />
139   <xs:attribute name="deriv" type="xs:integer" />
140 </xs:complexType>
141 
142 <xs:complexType name="variablesValuesType">
143   <xs:sequence>
144     <xs:element ref="variable-value" minOccurs="0" maxOccurs="unbounded" />
145   </xs:sequence>
146   <xs:attribute name="numberOfVariables" type="xs:integer" use="required" />
147 </xs:complexType>
148 
149 <xs:complexType name="variableValueType">
150   <xs:attribute name="name" type="xs:NMTOKEN" use="required" />
151   <xs:attribute name="value" type="xs:double" use="required" />
152 </xs:complexType>
153 
154 <xs:complexType name="inputFunctionsType">
155   <xs:sequence>
156     <xs:element ref="input-function" minOccurs="0" maxOccurs="unbounded" />
157   </xs:sequence>
158   <xs:attribute name="numberOfInputFunctions" type="xs:integer" use="required
        " />
159 </xs:complexType>
160 
161 <xs:complexType name="inputFunctionType">
162   <xs:sequence>
163     <xs:element ref="time-series" />
```

```
  </xs:sequence>
  <xs:attribute name="name" type="xs:NMTOKEN" use="required" />
  <xs:attribute name="deriv" type="xs:integer" use="required" />
</xs:complexType>

<xs:element name="time-series">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="time-value" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="startTime" type="REAL-NUMBER" use="optional" />
    <xs:attribute name="endTime" type="REAL-NUMBER" use="optional" />
  <xs:attribute name="interpolationMethod" type="INTERPOLATION-METHOD" use="
      required" />
  </xs:complexType>
</xs:element>

<xs:element name="time-value">
  <xs:complexType>
    <xs:attribute name="time" type="REAL-NUMBER" use="required"/>
    <xs:attribute name="value" type="REAL-NUMBER" use="required"/>
  </xs:complexType>
</xs:element>

<xs:simpleType name="INTERPOLATION-METHOD">
  <xs:restriction base="xs:string">
    <xs:enumeration value="none"/>
    <xs:enumeration value="linear"/>
    <xs:enumeration value="exponentialSpline"/>
    <xs:enumeration value="cubicSpline"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="REAL-NUMBER">
  <xs:restriction base="xs:double">
  </xs:restriction>
</xs:simpleType>
</xs:schema>
```

Annexe C

Code Source Java

C.1 Définition de la classe `FunctionKey`

```java
package fr.ensma.lias.timeseriesreductorslib.reductors.models.differentialequations;

public class FunctionKey {
    private String name;
    private int deriv;

    public FunctionKey( String name, int deriv ) {
        this.name = name;
        this.deriv = deriv;
    }

    public String getName() {
        return name;
    }

    public void setName( String name ) {
        this.name = name;
    }

    public int getDeriv() {
        return deriv;
    }

    public void setDeriv( int deriv ) {
        this.deriv = deriv;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + deriv;
        result = prime * result + ( ( name == null ) ? 0 : name.hashCode() );
        return result;
    }

    @Override
```

```
38      public boolean equals( Object obj ) {
39          if ( this == obj )
40              return true;
41          if ( obj == null )
42              return false;
43          if ( getClass() != obj.getClass() )
44              return false;
45          FunctionKey other = (FunctionKey) obj;
46          if ( deriv != other.deriv )
47              return false;
48          if ( name == null ) {
49              if ( other.name != null )
50                  return false;
51          } else if ( !name.equals( other.name ) )
52              return false;
53          return true;
54      }
55
56      @Override
57      public String toString() {
58          return "FunctionKey [name=" + name + ", deriv=" + deriv + "]";
59      }
60
61  }
```

C.2 Implémentation des algorithmes de traitement numérique

C.2.1 Implémentation Java de l'algorithme de Runge-Kutta

```
1   /**
2    *
3    * Generates a time series from an equation. It generates as many values as
         available in the
4    * inputs functions.
5    *
6    * @param function,
7    * the approximation function to use
8    * @return
9    * @throws Exception
10  */
11  public TimeSeriesDoubleDouble equationToTimeseriesRK4() throws Exception {
12      TimeSeriesDoubleDouble result = new TimeSeriesDoubleDouble();
13      // Double step = this.getDiffEqSyst().getStep();
14      // Double start = this.diffEqSyst.getAbscissa();
15
16      // gets the initial time t_0 from the input time series
17      FunctionKey firstKey = null;
```

```
18      Double currentTime = 0.0d;
19
20      // uses initial conditions to initialize
21      // Y_0=[y_0^(0),y_0^(1),...,y_0^(p-1)]^T, where p is the order of the
22      // equation. For j in [1,..,p-1], y_0^(j) = y(j)^(t_0).
23      List<Double> Y_n = this.initU1( parametersSet.getInitialValues(),
            unkownFunctionName );
24
25      if ( !inputFunctionsValues.isEmpty() ) {
26          Iterator<FunctionKey> inputKeysIterator = inputFunctionsValues.keySet().
                iterator();
27          firstKey = inputKeysIterator.next();
28          currentTime = inputFunctionsValues.get( firstKey ).firstKey();
29
30          // the first values of the numerical solution of the equation is
31          // y_0^(0).
32          result.put( currentTime, parametersSet.getInitialValues().get( new
                FunctionKey( unkownFunctionName, 0 ) ) );
33
34          // then we use the recurrence relation to calculate the y_i's values,
35          // 1<i<n
36          while ( inputFunctionsValues.get( firstKey ).higherKey( currentTime ) != null
                ) {
37              Y_n = computeYNP1( Y_n, currentTime, inputFunctionsValues.get( firstKey )
                    .higherKey( currentTime ) );
38              Double value = Y_n.get( 0 );
39              currentTime = inputFunctionsValues.get( firstKey ).higherKey( currentTime
                    );
40              result.put( currentTime, value );
41          }
42      } else {
43          // the first values of the numerical solution of the equation is
44          // y_0^(0).
45          result.put( currentTime, parametersSet.getInitialValues().get( new
                FunctionKey( unkownFunctionName, 0 ) ) );
46
47          double nextTime;
48
49          // then we use the recurrence relation to calculate the y_i's values,
50          // 1<i<n
51          for ( int i = 1; i < 5000; i++ ) {
52              nextTime = currentTime + 1.0;
53              nextTime = (double) Math.round( nextTime );
54              Y_n = computeYNP1( Y_n, currentTime, nextTime );
55              Double value = Y_n.get( 0 );
56              currentTime = nextTime;
57              result.put( currentTime, value );
58          }
59      }
60      return result;
```

```
61  }
62
63  /**
64   *
65   * Computes Y_(n+1) = Y_n + (k1 + 2*k2 + 2*k3 + k4) / 6, using the RK4 method
         formulas
66   *
67   * @param Y,
68   * Y_n
69   * @param t,
70   * the current time t_n
71   * @param tph
72   * = t_(n+1)
73   * @return
74  */
75  private List<Double> computeYNP1( List<Double> Y, Double t1, Double t2 ) {
76      // YNP1 is initialized to 0
77      List<Double> YNP1 = new ArrayList<Double>();
78
79      // vector and time to make computations with
80      List<Double> vector = new ArrayList<Double>();
81      double time;
82      double h = t2 - t1;
83
84      // computes the Ki values
85      List<Double> K1 = new ArrayList<Double>();
86      List<Double> K2 = new ArrayList<Double>();
87      List<Double> K3 = new ArrayList<Double>();
88      List<Double> K4 = new ArrayList<Double>();
89      List<Double> F = new ArrayList<Double>();
90
91      // K1
92      time = t1;
93      for ( int i = 0; i < Y.size(); i++ ) {
94          vector.add( Y.get( i ) );
95      }
96      F = computeF( vector, t1, t2, time );
97      for ( int i = 0; i < F.size(); i++ ) {
98          K1.add( F.get( i ) * h );
99      }
100
101     // K2
102     time = t1 + 0.5d * h;
103     for ( int i = 0; i < K1.size(); i++ ) {
104         vector.set( i, Y.get( i ) + 0.5d * K1.get( i ) );
105     }
106     F = computeF( vector, t1, t2, time );
107     for ( int i = 0; i < F.size(); i++ ) {
108         K2.add( F.get( i ) * h );
109     }
```

```
110
111     // K3
112     for ( int i = 0; i < K1.size(); i++ ) {
113         vector.set( i, Y.get( i ) + 0.5d * K2.get( i ) );
114     }
115     F = computeF( vector, t1, t2, time );
116     for ( int i = 0; i < F.size(); i++ ) {
117         K3.add( F.get( i ) * h );
118     }
119
120     // K4
121     time = t1 + h;
122     for ( int i = 0; i < K1.size(); i++ ) {
123         vector.set( i, Y.get( i ) + K3.get( i ) );
124     }
125     F = computeF( vector, t1, t2, time );
126     for ( int i = 0; i < F.size(); i++ ) {
127         K4.add( F.get( i ) * h );
128     }
129
130     // YNP1 = Y + (K1 + 2*K2 + 2*K3 + K4)/6
131     for ( int i = 0; i < Y.size(); i++ ) {
132         YNP1.add( Y.get( i ) + ( K1.get( i ) + 2 * K2.get( i ) + 2 * K3.get( i ) + K4
                .get( i ) ) / 6.0d );
133     }
134
135     return YNP1;
136 }
137
138 /**
139  *
140  * Computes the value of F(Y,t), where t1 and t2 are time keys of the input series
        that point
141  * out a segment of the series that will be approximated by a straight line in order
         to be able
142  * to evaluate the input function at any point t.
143  *
144  * @param Y
145  * @param t1
146  * @param t2
147  * @param t
148  * @return
149 */
150 private List<Double> computeF( List<Double> Y, Double t1, Double t2, Double t ) {
151     List<Double> F = new ArrayList<Double>();
152     List<NodeObject> DYclone = new ArrayList<NodeObject>();
153     // cloning DY to avoid altering it
154     for ( int i = 0; i < DY.size(); i++ ) {
155         DYclone.add( DY.get( i ).clone() );
156     }
```

```
157
158     // replace the value of the unknown by the values of Y
159     for ( int i = 0; i < Y.size(); i++ ) {
160         for ( int j = 0; j < DYclone.size(); j++ ) {
161             DYclone.set( j, DYclone.get( j ).replaceFunctionByVar2( new FunctionKey(
                    unkownFunctionName, i ),
162                     Y.get( i ) ) );
163         }
164     }
165
166     // replaces the input functions by their value at t, using a straight
167     // line approximation
168     for ( FunctionKey key : inputFunctionsValues.keySet() ) {
169         double value = inputFunctionsValues.get( key ).getInterpolationFunction()
170                 .interpolate( inputFunctionsValues.get( key ), t1, t2, t );
171         for ( int j = 0; j < DYclone.size(); j++ ) {
172             DYclone.set( j, DYclone.get( j ).replaceFunctionByVar2( key, value ) );
173         }
174     }
175
176     // computes the value of F
177     for ( int j = 0; j < DYclone.size(); j++ ) {
178         F.add( reversePolishEvaluation( DYclone.get( j ) ) );
179     }
180
181     return F;
182 }
183
184 private double reversePolishEvaluation( NodeObject expression ) {
185     // if the node is a number it returns its own value
186     if ( expression instanceof Number )
187         return ( (Number) expression ).getDoubleValue();
188     // if the node is a binary operator, it evaluates both of its children,
189     // do the operation and returns the value
190     else if ( expression instanceof BinaryOperator ) {
191         double left = reversePolishEvaluation( expression.getLeftPart() );
192         double right = reversePolishEvaluation( expression.getRightPart() );
193         try {
194             return BinaryOperatorEnum.compute( left, right, expression.getValue() );
195         } catch ( Exception e ) {
196             e.printStackTrace();
197         }
198         // if it is a unary operator, it evaluate its child, do the
199         // operation and returns the value
200     } else if ( expression instanceof UnaryOperator ) {
201         double right = reversePolishEvaluation( expression.getLeftPart() );
202         try {
203             return UnaryOperatorEnum.compute( right, expression.getValue() );
204         } catch ( Exception e ) {
205             e.printStackTrace();
```

```
206         }
207     }
208     // in case something went wrong (a node were not a number or an
209     // operator), it return an infinite value
210     return Double.POSITIVE_INFINITY;
211 }
```

C.2.2 Implémentation de l'algorithme de comparaison

```
1  package fr.ensma.lias.timeseriesreductorslib.comparison;
2
3  import java.util.HashMap;
4  import java.util.Map;
5  import java.util.Set;
6
7  import org.apache.log4j.Logger;
8
9  import fr.ensma.lias.timeseriesreductorslib.timeseries.TimeSeriesDoubleDouble;
10
11 /**
12  * The class implements an algorithm used to perform comparison between two series.
13  *
14  * @author cyrille ponchateau (cyrille.ponchateau@ensma.fr)
15  *
16  */
17 public class CompareAlgorithm
18 {
19     // elements keys
20     public static final String MEAN_DELTA = "meandelta";
21     public static final String MEAN_DELTA_MIN = "meandeltamin";
22     public static final String MEAN_DELTA_MAX = "meandeltamax";
23
24     public static final String STANDARD_DEVIATION = "standarddeviation";
25
26     // the statistics values are stored into a map
27     private static Map<String, Double> statsMap;
28
29     /**
30      * Compute statistics properties of to time series (mean deltas, delta min,
            delta max,
31      * standardDeviation).
32      *
33      * The mean, percent and standard deviation are calculated using the incremental
            update formula.
34      * Two reasons motivated that choice:</br>
35      * <ul>
36      * <li>Those formulas avoid values summation to grow to big and enhanced the
            accuracy of the
```

```
37      * computation</li>
38      * <li>Those formulas seems easier to adapt to a stream</li>
39      * </ul>
40      *
41      * @param t1
42      * @param t2
43      * @param computePercents
44      * @param searchMinMax
45      * @param computeStandardDeviation
46      * @param computeCorrelation
47      * @return
48      */
49     public static void statistics( TimeSeriesDoubleDouble modelSeries,
            TimeSeriesDoubleDouble rawSeries )
50             throws Exception
51     {
52         statsMap = new HashMap<String, Double>();
53
54         // initializing statsMap
55         statsMap.put( MEAN_DELTA, Double.POSITIVE_INFINITY );
56         statsMap.put( STANDARD_DEVIATION, Double.POSITIVE_INFINITY );
57         statsMap.put( MEAN_DELTA_MIN, Double.POSITIVE_INFINITY );
58         statsMap.put( MEAN_DELTA_MAX, Double.POSITIVE_INFINITY );
59
60         // the models series size must be at least equal to the raw series size
61         if ( modelSeries.size() >= rawSeries.size() )
62         {
63             double modelFirstKey = modelSeries.firstKey();
64             double rawFirstKey = rawSeries.firstKey();
65             double modelSecondKey = modelSeries.higherKey( modelFirstKey );
66             double rawSecondKey = rawSeries.higherKey( rawFirstKey );
67             double modelH = modelSecondKey - modelFirstKey;
68             double rawH = rawSecondKey - rawFirstKey;
69
70             if ( Math.abs( modelH - rawH ) <= 10e-7 )
71             {
72                 // the current delta value, needed in the for loop
73                 double currentDelta = Math.abs(
74                         modelSeries.get( modelSeries.firstKey() ).doubleValue()
75                                 - rawSeries.get( modelSeries.firstKey() ) );
76                 // when calculation the n-th value of the mean, it contains the
77                 // (n-1)-th
78                 // value
79                 double previousMean = currentDelta;
80
81                 // mean variables
82                 double meanDelta = currentDelta;
83                 double deltaMin = Double.POSITIVE_INFINITY;
84                 double deltaMax = Double.NEGATIVE_INFINITY;
85
```

```
86                  // standard deviation variables
87                  double standardDeviation = 0.0;
88
89                  // double correlation;
90                  TimeSeriesDoubleDouble clone = rawSeries.clone();
91                  Set<Double> keys = clone.keySet();
92                  keys.remove( clone.firstKey() );
93
94                  int n = 2;
95                  // going through the time series
96                  for ( Double t : keys )
97                  {
98                      // computes current delta
99                      if ( rawSeries.get( t ) == null )
100                     {
101                         meanDelta = Double.POSITIVE_INFINITY;
102                         standardDeviation = Double.POSITIVE_INFINITY;
103                         deltaMin = Double.POSITIVE_INFINITY;
104                         deltaMax = Double.NEGATIVE_INFINITY;
105                     } else
106                     {
107                         currentDelta = Math.abs( modelSeries.get( t ) - rawSeries.get(
                                t ) );
108                         // update mean, percent and standard deviation
109                         meanDelta = meanDelta + ( currentDelta - meanDelta ) / n;
110                         standardDeviation = standardDeviation
111                                 + ( currentDelta - previousMean ) * ( currentDelta -
                                    meanDelta );
112                         // prepares previous mean value for the next loop
113                         previousMean = currentDelta;
114
115                         // min values seeking
116                         if ( deltaMin > meanDelta )
117                             deltaMin = meanDelta;
118
119                         // max values seeking
120                         if ( deltaMax < meanDelta )
121                             deltaMax = meanDelta;
122
123                         n++;
124                     }
125                 }
126
127                 // standard deviation is not calculated directly in the loop, in
128                 // fact it is the variance that is calculated, the standard
129                 // deviation is derived from it.
130                 standardDeviation = Math.sqrt( standardDeviation / rawSeries.size() )
                        ;
131
132                 // storing the results
```

```
133                 statsMap.put( MEAN_DELTA, meanDelta );
134                 statsMap.put( STANDARD_DEVIATION, standardDeviation );
135                 statsMap.put( MEAN_DELTA_MIN, deltaMin );
136                 statsMap.put( MEAN_DELTA_MAX, deltaMax );
137             }
138         }
139     }
140 
141     public static void printResults()
142     {
143         String results = "";
144         results = "Mean deltas: " + statsMap.get( MEAN_DELTA ) + "; ";
145         results = results + "Mean deltas min: " + statsMap.get( MEAN_DELTA_MIN ) + ";
                ";
146         results = results + "Mean deltas max: " + statsMap.get( MEAN_DELTA_MAX ) + ";
                ";
147         results = results + "Standard deviation: " + statsMap.get( STANDARD_DEVIATION
                ) + ";";
148         Logger.getLogger( "STATIC_LOGGER:CompareAlgorithm" ).debug( results );
149     }
150 
151     public static Map<String, Double> getStatsMap()
152     {
153         return statsMap;
154     }
155 
156     public static ENotification notify( double meanErrorMax, double
            standardDeviationMax, double meanErrorMin,
157             double standardDeviationMin )
158     {
159         if ( statsMap.isEmpty() )
160             return ENotification.MODEL_REJECTED;
161         else if ( statsMap.get( MEAN_DELTA ) > meanErrorMax
162                 || statsMap.get( STANDARD_DEVIATION ) > standardDeviationMax )
163             return ENotification.MODEL_REJECTED;
164         else if ( statsMap.get( MEAN_DELTA ) < meanErrorMin
165                 && statsMap.get( STANDARD_DEVIATION ) < standardDeviationMin )
166             return ENotification.MODEL_ACCEPTED;
167         else
168             return ENotification.UNDETERMINED;
169     }
170 }
```

C.3 Implémentation Java des algorithmes de conversions Arbre-XML

C.3.1 Lecture de l'ensemble des paramètres

```
1  /**
2   * reads the parameters set of an equation in an XML document
3   * @param document
4   * @param differentialEquation
5  */
6  private static void createParametersSet20( Document document,
7          DifferentialEquation2 differentialEquation ) {
8      // initializes the initial values, input functions and variables values
9      // maps
10     Map<FunctionKey, Double> initialValues = new HashMap<FunctionKey, Double>();
11     Map<FunctionKey, String> inputFunctions = new HashMap<FunctionKey, String>();
12     Map<String, Double> variablesValues = new HashMap<String, Double>();
13     // get all the initial values tags
14     NodeList nodeList = document.getElementsByTagName( INITIAL_VALUE_TAG_NAME );
15     for ( int i = 0; i < nodeList.getLength(); i++ ) {
16         // goes through the initial values tags list and add new initial
17         // value in the initial values map, for each tag of the list
18         Element element = (Element) nodeList.item( i );
19         differentialEquation.getParametersSet().getInitialValues()
20                 .put( new FunctionKey( element.getAttribute( NAME_ATTRIBUTE ),
21                         Integer.parseInt( element.getAttribute( DERIV_ATTRIBUTE ) ) ),
22                         Double.parseDouble( element.getAttribute( VALUE_ATTRIBUTE ) )
                               );
23     }
24
25     // idem with input function tags
26     nodeList = document.getElementsByTagName( INPUT_FUNCTION_TAG_NAME );
27     for ( int i = 0; i < nodeList.getLength(); i++ ) {
28         Element element = (Element) nodeList.item( i );
29         FunctionKey key = new FunctionKey( element.getAttribute( NAME_ATTRIBUTE ),
30                 Integer.parseInt( element.getAttribute( DERIV_ATTRIBUTE ) ) );
31         differentialEquation.getParametersSet().getInputFunctions().put( key, "" );
32         NodeList timeseriesElements = element.getElementsByTagName(
               TIME_SERIES_TAG_NAME );
33         Element timeseriesElement = (Element) timeseriesElements.item( 0 );
34         NodeList timeValuesNodes = element.getElementsByTagName( TIME_VALUE_TAG_NAME
               );
35         // reading current input function values
36         TimeSeriesDoubleDouble series = new TimeSeriesDoubleDouble();
37         for ( int j = 0; j < timeValuesNodes.getLength(); j++ ) {
38             Element timeValueElement = (Element) timeValuesNodes.item( j );
39             series.put( Double.parseDouble( timeValueElement.getAttribute(
                   TIME_ATTRIBUTE ) ),
```

```
40                      Double.parseDouble( timeValueElement.getAttribute( VALUE_ATTRIBUTE
                            ) ) );
41              differentialEquation.getInputFunctions().put( key, series );
42          }
43          differentialEquation.getInputFunctions().get( key ).setInterpolationFunction(
                EInterpolationFunction
44                  .getInterpolationFunction( timeseriesElement.getAttribute(
                        INTERPOLATION_METHOD_ATTRIBUTE ) ) );
45      }
46      // and idem with variable value tags
47      nodeList = document.getElementsByTagName( VARIABLE_VALUE_TAG_NAME );
48      for ( int i = 0; i < nodeList.getLength(); i++ ) {
49          Element element = (Element) nodeList.item( i );
50          differentialEquation.getParametersSet().getMapping().put( element.
                getAttribute( NAME_ATTRIBUTE ),
51                  Double.parseDouble( element.getAttribute( VALUE_ATTRIBUTE ) ) );
52      }
53  }
```

C.3.2 Lecture de la formule

```
1   /**
2    * Builds an equation tree, from its XML description.
3    *
4    * @param element
5    * @return
6    */
7   private static NodeObject createNodeObject( Element element ) {
8       NodeObject newNode = null;
9       if ( element.getTagName().equals( BINARY_OPERATOR_TAG_NAME ) ) {
10          // the current element is a binary-operator element
11          newNode = new BinaryOperator();
12          // sets operator value
13          newNode.setValue( element.getAttribute( VALUE_ATTRIBUTE ) );
14          List<Element> children = getChildNodes( element );
15          // sets left and right children recursively
16          newNode.setLeftChild( createNodeObject( children.get( 0 ) ) );
17          newNode.setRightChild( createNodeObject( children.get( 1 ) ) );
18          return newNode;
19      } else if ( element.getTagName().equals( UNARY_OPERATOR_TAG_NAME ) ) {
20          // the current element is an unary operator
21          newNode = new UnaryOperator();
22          // sets operator value
23          newNode.setValue( element.getAttribute( VALUE_ATTRIBUTE ) );
24          List<Element> children = getChildNodes( element );
25          // sets left child recursively
26          newNode.setRightChild( createNodeObject( children.get( 0 ) ) );
27          return newNode;
```

```
28      } else if ( element.getTagName().equals( FUNCTION_TAG_NAME ) ) {
29          // the current node is a function
30          newNode = new Function();
31          // sets deriv and value (name) of the function
32          newNode.setDeriv( Integer.parseInt( element.getAttribute( DERIV_ATTRIBUTE ) )
                );
33          newNode.setValue( element.getAttribute( VALUE_ATTRIBUTE ) );
34          List<Element> child = getChildNodes( element );
35          // if the function has a child, it is set recursively as well
36          if ( !child.isEmpty() ) {
37              newNode.setLeftChild( createNodeObject( child.get( 0 ) ) );
38          }
39          return newNode;
40      } else if ( element.getTagName().equals( VARIABLE_TAG_NAME ) ) {
41          // the current node is a variable
42          newNode = new Variable();
43          // sets the value (name) of the variable and stops the recursion
44          // (variables have no children)
45          newNode.setValue( element.getAttribute( VALUE_ATTRIBUTE ) );
46          return newNode;
47      } else if ( element.getTagName().equals( NUMBER_TAG_NAME ) ) {
48          // the current node is a number
49          newNode = new Number();
50          // sets the number value and stops the recursion (numbers have no
51          // children)
52          newNode.setValue( element.getAttribute( VALUE_ATTRIBUTE ) );
53          return newNode;
54      }
55      return newNode;
56  }
```

C.3.3 Écriture de l'ensemble des paramètres

```
1   /**
2    * Creates the element structure for a parameters set (system).
3    *
4    * @param parametersSet
5    * @param document
6    * @return
7    */
8   private static Element createElement( DifferentialEquation2 equation,
9                                         Document document ) {
10      ParametersSet parametersSet = equation.getParametersSet();
11      // create new parameters set element
12      Element parametersSetElement = document.createElement( PARAMETERS_SET_TAG_NAME )
            ;
13      // create initial values element
14      Element initialValuesElement = document.createElement( INITIAL_VALUES_TAG_NAME )
            ;
```

```
15      initialValuesElement.setAttribute( NUMBER_OF_INITIAL_VALUES_ATTRIBUTE,
16              String.valueOf( parametersSet.getInitialValues().size() ) );
17      // going through the initial values list
18      for ( FunctionKey key : parametersSet.getInitialValues().keySet() ) {
19          // create an initial value element
20          Element initialValueElement = document.createElement( INITIAL_VALUE_TAG_NAME
                );
21          initialValueElement.setAttribute( NAME_ATTRIBUTE, key.getName() );
22          initialValueElement.setAttribute( DERIV_ATTRIBUTE,
23                                            String.valueOf( key.getDeriv() ) );
24          initialValueElement.setAttribute( VALUE_ATTRIBUTE,
25                  String.valueOf( parametersSet.getInitialValues().get( key ) ) );
26          initialValuesElement.appendChild( initialValueElement );
27      }
28      // appending initial values element
29      parametersSetElement.appendChild( initialValuesElement );
30      // create variables value element
31      Element variablesValuesElement = document
32                                        .createElement( VARIABLES_VALUES_TAG_NAME );
33      variablesValuesElement.setAttribute( NUMBER_OF_VARIABLES_ATTRIBUTE,
34              String.valueOf( parametersSet.getMapping().size() ) );
35      // going through the variable list of the parameters set
36      for ( String variable : parametersSet.getMapping().keySet() ) {
37          // create a variable value element
38          Element variableValueElement = document
39                                        .createElement( VARIABLE_VALUE_TAG_NAME );
40          variableValueElement.setAttribute( NAME_ATTRIBUTE, variable );
41          variableValueElement.setAttribute( VALUE_ATTRIBUTE,
42                  String.valueOf( parametersSet.getMapping().get( variable ) ) );
43          variablesValuesElement.appendChild( variableValueElement );
44      }
45      // appending variables values element to parameters set element
46      parametersSetElement.appendChild( variablesValuesElement );
47      // create input functions element
48      Element inputFunctionsElement = document.createElement( INPUT_FUNCTIONS_TAG_NAME
            );
49      inputFunctionsElement.setAttribute( NUMBER_OF_INPUT_FUNCTIONS_ATTRIBUTE,
50              String.valueOf( equation.getInputFunctions().size() ) );
51      // going through the input functions list of the parameters set
52      for ( FunctionKey key : equation.getInputFunctions().keySet() ) {
53          // create input function element
54          Element inputFunctionElement = document
55                                         .createElement( INPUT_FUNCTION_TAG_NAME );
56          inputFunctionElement.setAttribute( NAME_ATTRIBUTE, key.getName() );
57          inputFunctionElement.setAttribute( DERIV_ATTRIBUTE,
58                                             String.valueOf( key.getDeriv() ) );
59          // creates a time series tag
60          Element timeSeriesElement = document.createElement( TIME_SERIES_TAG_NAME );
61          timeSeriesElement.setAttribute( INTERPOLATION_METHOD_ATTRIBUTE,
62                  equation.getInputFunctions().get( key )
```

```
                .getInterpolationFunction().getInterpolationFunctionName().value() );
        // reads the input series of the equation and copy the value into
        // time value tags, which are then appended to the time series tag
        for ( Double timeKey : equation.getInputFunctions().get( key ).keySet() ) {
            Element timeValueElement = document.createElement( TIME_VALUE_TAG_NAME );
            timeValueElement.setAttribute( TIME_ATTRIBUTE, String.valueOf( timeKey )
                );
            timeValueElement.setAttribute( VALUE_ATTRIBUTE,
                    String.valueOf( equation.getInputFunctions()
                                            .get( key ).get( timeKey ) ) );
            timeSeriesElement.appendChild( timeValueElement );
        }
        inputFunctionElement.appendChild( timeSeriesElement );
        inputFunctionsElement.appendChild( inputFunctionElement );
    }
    // appending input functions element to parameters set element
    parametersSetElement.appendChild( inputFunctionsElement );
    return parametersSetElement;
}
```

C.3.4 Écriture d'une formule

```
/**
 * Recursive building of the element structure corresponding to the equation.
 *
 * @param node
 * @param document
 * @return
 */
private static Element createElement( NodeObject node, Document document ) {
    Element element = null;
    if ( node instanceof BinaryOperator ) {
        // creating binary operator element
        element = document.createElement( BINARY_OPERATOR_TAG_NAME );
        // setting value attribute
        Attr value = document.createAttribute( VALUE_ATTRIBUTE );
        value.setValue( node.getValue() );
        element.setAttributeNode( value );
        // creating left child element
        element.appendChild( createElement( node.getLeftPart(), document ) );
        // creating right child element
        element.appendChild( createElement( node.getRightPart(), document ) );
    } else if ( node instanceof UnaryOperator ) {
        // creating unary operator element
        element = document.createElement( UNARY_OPERATOR_TAG_NAME );
        // setting value attribute
        element.setAttribute( VALUE_ATTRIBUTE, node.getValue() );
        // creating child
```

```
27          element.appendChild( createElement( node.getLeftPart(), document ) );
28      } else if ( node instanceof Function ) {
29          // creating function element
30          element = document.createElement( FUNCTION_TAG_NAME );
31          // setting attributes
32          element.setAttribute( VALUE_ATTRIBUTE, node.getValue() );
33          element.setAttribute( DERIV_ATTRIBUTE, String.valueOf( node.getDeriv() ) );
34          // check if the function has a child
35          if ( node.hasChildren() )
36              element.appendChild( createElement( node.getLeftPart(), document ) );
37      } else if ( node instanceof Number ) {
38          // creating number element
39          element = document.createElement( NUMBER_TAG_NAME );
40          // setting attribute
41          element.setAttribute( VALUE_ATTRIBUTE, node.getValue() );
42      } else if ( node instanceof Variable ) {
43          // creating variable element
44          element = document.createElement( VARIABLE_TAG_NAME );
45          // setting attribute
46          element.setAttribute( VALUE_ATTRIBUTE, node.getValue() );
47      }
48      return element;
49  }
```

C.4 Implémentation Java des algorithmes de conversions XML-Flat

C.4.1 Lecture de l'ensemble des paramètres XML à Flat

```
1       /**
2        * Reads the elements of the parameters set of the XML file and maps the
             information into the
3        * appropriate objects of the FlatDifferentialEquation object.
4        *
5        * @param flateqdiff
6        * @param document
7        * @see fr.ensma.lias.mmwdbapi.pojos.FlatDifferentialEquation
8        */
9       private void readParametersSet( FlatDifferentialEquation flateqdiff, Document
            document,
10              ArrayList<Long> initialValuesUnavailableIds, ArrayList<Long>
                    inputFunctionsUnavailbleIds,
11              ArrayList<Long> variableValuesUnavailableIds ) {
12          // reading initial values
13          // getting initial-value tags list
14          NodeList nodesList = document.getElementsByTagName( INITIAL_VALUE_TAG_NAME );
15          // preparing generator
```

```
16          IdGenerator idGenerator = new IdGenerator( initialValuesUnavailableIds );
17          Element currentNode = null;
18          Long currentId;
19
20          // creating a new InitialValue Object for each tag in nodesList and
21          // adding it to initialValues map in FlatDifferentialEquation object
22          for ( int i = 0; i < nodesList.getLength(); i++ ) {
23              currentNode = (Element) nodesList.item( i );
24              currentId = idGenerator.getValidId();
25              flateqdiff.getInitialValues().put( currentId,
26                     new InitialValue( currentId, currentNode.getAttribute(
                           NAME_ATTRIBUTE ),
27                         Integer.parseInt( currentNode.getAttribute( DERIV_ATTRIBUTE
                               ) ),
28                         Double.parseDouble(
29                                currentNode.getAttribute( VALUE_ATTRIBUTE ) ) ) );
30          }
31
32          // reading variable values
33          nodesList = document.getElementsByTagName( VARIABLE_VALUE_TAG_NAME );
34          // reset the generator
35          idGenerator = new IdGenerator( variableValuesUnavailableIds );
36
37          for ( int i = 0; i < nodesList.getLength(); i++ ) {
38              currentNode = (Element) nodesList.item( i );
39              currentId = idGenerator.getValidId();
40              flateqdiff.getVariables().put( currentId,
41                     new Variable( currentId, currentNode.getAttribute( NAME_ATTRIBUTE
                           ),
42                         Double.parseDouble( currentNode.getAttribute(
                               VALUE_ATTRIBUTE ) ) ) );
43          }
44
45          // reading input functions
46          nodesList = document.getElementsByTagName( INPUT_FUNCTION_TAG_NAME );
47          idGenerator = new IdGenerator( inputFunctionsUnavailbleIds );
48
49          for ( int i = 0; i < nodesList.getLength(); i++ ) {
50              currentNode = (Element) nodesList.item( i );
51              currentId = idGenerator.getValidId();
52              NodeList timeseriesElements = currentNode.getElementsByTagName(
                    TIME_SERIES_TAG_NAME );
53              Element timeseriesElement = (Element) timeseriesElements.item( 0 );
54              flateqdiff.getInputs().put( currentId, new Input( currentId, currentNode.
                    getAttribute( NAME_ATTRIBUTE ),
55                      Integer.parseInt( currentNode.getAttribute( DERIV_ATTRIBUTE ) ),
56                      timeseriesElement.getAttribute( INTERPOLATION_METHOD_ATTRIBUTE ) )
                            );
57              NodeList timeValueElements = currentNode.getElementsByTagName(
                    TIME_VALUE_TAG_NAME );
```

```
58            for ( int j = 0; j < timeValueElements.getLength(); j++ ) {
59                Element timeValueElement = (Element) timeValueElements.item( j );
60                flateqdiff.getInputs().get( currentId ).getSeries().put(
61                        Double.parseDouble( timeValueElement.getAttribute(
                              TIME_ATTRIBUTE ) ),
62                        Double.parseDouble( timeValueElement.getAttribute(
                              VALUE_ATTRIBUTE ) ) );
63            }
64        }
65    }
```

C.4.2 Lecture de la formule XML à Flat

```
1     /**
2      * Recursively reads the formula tag of the XML file to map the node in the
           nodes list of the
3      * Formula object of a FlatDifferentialEquation object.
4      *
5      * @param flateqdiff
6      * @param document
7      * @param currentElement,
8      * the current node Element to put in the node list
9      * @param currentNodeId,
10     * the id of the current node to put in the nodes map
11     * @param parentId,
12     * the id of the parent of the node
13     * @param depth,
14     * the depth of the node in the tree formula
15     * @param generator,
16     * the id generator
17     * @see fr.ensma.lias.mmwdbapi.pojos.FlatDifferentialEquation
18     * @see fr.ensma.lias.mmwdbapi.pojos.Formula
19     */
20    private void readFormula( FlatDifferentialEquation flateqdiff, Document document
          , Element currentElement,
21            Long currentNodeId, Long parentId, int depth, IdGenerator generator ) {
22        Long leftId;
23        Long rightId;
24        int deriv;
25
26        if ( currentElement.getAttribute( DERIV_ATTRIBUTE ).equals( "" ) ) {
27            deriv = -1;
28        } else {
29            deriv = Integer.parseInt( currentElement.getAttribute( DERIV_ATTRIBUTE )
                  );
30        }
31
32        if ( currentElement.getTagName().equals( EMathObject.BINARY_OPERATOR.value()
              ) ) {
```

```
33              // the current node is a binary operator, thus has two children
34              leftId = generator.getValidId(); // get an id for each of its kids
35              rightId = generator.getValidId();
36              // fill the formula node with the current node data
37              flateqdiff.getFormula().put( currentNodeId,
38                      new Node( currentNodeId, EMathObject.fromValue( currentElement.
                            getTagName() ),
39                          currentElement.getAttribute( VALUE_ATTRIBUTE ), deriv,
                                depth, parentId, leftId, rightId,
40                          flateqdiff.getId() ) );
41              // recursively process the left node of the current node
42              // which depth is increased by one, parentId is the id of the
43              // current node and node id is the leftId calculated above
44              try {
45                  readFormula( flateqdiff, document, (Element) currentElement.
                        getChildNodes().item( 0 ), leftId,
46                          currentNodeId, depth + 1, generator );
47                  // idem for right node
48                  readFormula( flateqdiff, document, (Element) currentElement.
                        getChildNodes().item( 1 ), rightId,
49                          currentNodeId, depth + 1, generator );
50              } catch ( ClassCastException e ) {
51                  e.printStackTrace();
52              }
53          } else if ( currentElement.getTagName().equals( EMathObject.UNARY_OPERATOR.
                value() )
54                  || ( currentElement.getTagName().toUpperCase().equals( EMathObject.
                        FUNCTION.value() )
55                          && currentElement.getFirstChild() != null ) ) {
56              // the current node is an unary node (has only one child)
57              // generates an id for its child
58              leftId = generator.getValidId();
59              // fills the formula with the node data
60              flateqdiff.getFormula().put( currentNodeId,
61                      new Node( currentNodeId, EMathObject.fromValue( currentElement.
                            getTagName() ),
62                          currentElement.getAttribute( VALUE_ATTRIBUTE ), deriv,
                                depth, parentId, leftId, 0,
63                          flateqdiff.getId() ) );
64              // processes the child of the node as a left child
65              readFormula( flateqdiff, document, (Element) currentElement.getChildNodes
                    ().item( 0 ), leftId,
66                      currentNodeId, depth + 1, generator );
67          } else if ( currentElement.getTagName().equals( EMathObject.VARIABLE.value()
                )
68                  || currentElement.getTagName().equals( EMathObject.NUMBER.value() )
69                  || ( currentElement.getTagName().equals( EMathObject.FUNCTION.value()
                         )
70                          && currentElement.getFirstChild() == null ) ) {
71              // the current node has no child
```

```
72              // processes the current node and stops the recursion
73              flateqdiff.getFormula().put( currentNodeId,
74                      new Node( currentNodeId, EMathObject.fromValue( currentElement.
                            getTagName() ),
75                          currentElement.getAttribute( VALUE_ATTRIBUTE ), deriv,
                                depth, parentId, 0, 0,
76                          flateqdiff.getId() ) );
77          }
78      }
79
80  }
```

C.4.3 Écriture de l'ensemble des paramètres Flat à XML

```
1       /**
2        * Creates the element structure for a parameters set (system).
3        *
4        * @param parametersSet
5        * @param document
6        * @return
7        */
8       private Element createElement( FlatDifferentialEquation eqdiff, Document
            document ) {
9           // create new parameters set element
10          Element parametersSetElement = document.createElement(
                PARAMETERS_SET_TAG_NAME );
11          // create initial element
12          Element initialValuesElement = document.createElement(
                INITIAL_VALUES_TAG_NAME );
13          initialValuesElement.setAttribute( NUMBER_OF_INITIAL_VALUES_ATTRIBUTE,
14                  String.valueOf( eqdiff.getInitialValues().size() ) );
15          // going through the initial value list of the parameters set
16          for ( Long id : eqdiff.getInitialValues().keySet() ) {
17              // create a variable value element
18              Element initialValueElement = document.createElement(
                    INITIAL_VALUE_TAG_NAME );
19              initialValueElement.setAttribute( NAME_ATTRIBUTE, eqdiff.getInitialValues
                    ().get( id ).getName() );
20              initialValueElement.setAttribute( VALUE_ATTRIBUTE,
21                      Double.toString( eqdiff.getInitialValues().get( id ).
                            getInitialValue() ) );
22              initialValueElement.setAttribute( DERIV_ATTRIBUTE,
23                      Integer.toString( eqdiff.getInitialValues().get( id ).getDeriv() )
                             );
24              initialValuesElement.appendChild( initialValueElement );
25          }
26          parametersSetElement.appendChild( initialValuesElement );
27          // create variables value element
```

```
28          Element variablesValuesElement = document.createElement(
                VARIABLES_VALUES_TAG_NAME );
29          variablesValuesElement.setAttribute( NUMBER_OF_VARIABLES_ATTRIBUTE,
30                  Integer.toString( eqdiff.getVariables().size() ) );
31          // going through the variable list of the parameters set
32          for ( Long id : eqdiff.getVariables().keySet() ) {
33              // create a variable value element
34              Element variableValueElement = document.createElement(
                    VARIABLE_VALUE_TAG_NAME );
35              variableValueElement.setAttribute( NAME_ATTRIBUTE, eqdiff.getVariables().
                    get( id ).getName() );
36              variableValueElement.setAttribute( VALUE_ATTRIBUTE,
37                      Double.toString( eqdiff.getVariables().get( id ).getValue() ) );
38              variablesValuesElement.appendChild( variableValueElement );
39          }
40          // appending variables values element to parameters set element
41          parametersSetElement.appendChild( variablesValuesElement );
42          // create input functions element
43          Element inputFunctionsElement = document.createElement(
                INPUT_FUNCTIONS_TAG_NAME );
44          inputFunctionsElement.setAttribute( NUMBER_OF_INPUT_FUNCTIONS_ATTRIBUTE,
45                  Integer.toString( eqdiff.getInputs().size() ) );
46          // going through the input functions list of the parameters set
47          for ( Long id : eqdiff.getInputs().keySet() ) {
48              // create input function element
49              Element inputFunctionElement = document.createElement(
                    INPUT_FUNCTION_TAG_NAME );
50              inputFunctionElement.setAttribute( NAME_ATTRIBUTE, eqdiff.getInputs().get
                    ( id ).getName() );
51              inputFunctionElement.setAttribute( DERIV_ATTRIBUTE,
52                      Integer.toString( eqdiff.getInputs().get( id ).getDeriv() ) );
53              // create time series element
54              Element timeSeriesElement = document.createElement( TIME_SERIES_TAG_NAME
                    );
55              timeSeriesElement.setAttribute( INTERPOLATION_METHOD_ATTRIBUTE,
56                      eqdiff.getInputs().get( id ).getInterpolationFunction().value() );
57              for ( Double timeKey : eqdiff.getInputs().get( id ).getSeries().keySet()
                    ) {
58                  Element timeValueElement = document.createElement(
                        TIME_VALUE_TAG_NAME );
59                  timeValueElement.setAttribute( TIME_ATTRIBUTE, String.valueOf(
                        timeKey ) );
60                  timeValueElement.setAttribute( VALUE_ATTRIBUTE,
61                          String.valueOf( eqdiff.getInputs().get( id ).getSeries().get(
                                timeKey ) ) );
62                  timeSeriesElement.appendChild( timeValueElement );
63              }
64              inputFunctionElement.appendChild( timeSeriesElement );
65              inputFunctionsElement.appendChild( inputFunctionElement );
66          }
```

```
67          // appending input functions element to parameters set element
68          parametersSetElement.appendChild( inputFunctionsElement );
69          return parametersSetElement;
70      }
```

C.4.4 Écriture de l'ensemble de la formule Flat à XML

```
1       /**
2        * Recursive building of the element structure corresponding to the equation.
3        *
4        * @param node
5        * @param document
6        * @return
7        */
8       private Element createElement( Formula formula, Document document, Long id ) {
9           Element element = null;
10          Node currentNode = formula.get( id );
11          if ( currentNode.getMathObject().equals( EMathObject.BINARY_OPERATOR ) ) {
12              // creating binary operator element
13              element = document.createElement( BINARY_OPERATOR_TAG_NAME );
14              // setting value attribute
15              Attr value = document.createAttribute( VALUE_ATTRIBUTE );
16              value.setValue( currentNode.getName() );
17              element.setAttributeNode( value );
18              // creating left child element
19              element.appendChild( createElement( formula, document, currentNode.
                    getLeft() ) );
20              // creating right child element
21              element.appendChild( createElement( formula, document, currentNode.
                    getRight() ) );
22          } else if ( currentNode.getMathObject().equals( EMathObject.UNARY_OPERATOR )
                ) {
23              // creating unary operator element
24              element = document.createElement( UNARY_OPERATOR_TAG_NAME );
25              // setting value attribute
26              element.setAttribute( VALUE_ATTRIBUTE, currentNode.getName() );
27              // creating child
28              element.appendChild( createElement( formula, document, currentNode.
                    getLeft() ) );
29          } else if ( currentNode.getMathObject().equals( EMathObject.FUNCTION ) ) {
30              // creating function element
31              element = document.createElement( FUNCTION_TAG_NAME );
32              // setting attributes
33              element.setAttribute( VALUE_ATTRIBUTE, currentNode.getName() );
34              element.setAttribute( DERIV_ATTRIBUTE, String.valueOf( currentNode.
                    getDeriv() ) );
35              // check if the function has a child
36              if ( currentNode.getLeft() > 0 )
```

```
37              element.appendChild( createElement( formula, document, currentNode.
                    getLeft() ) );
38          } else if ( currentNode.getMathObject().equals( EMathObject.NUMBER ) ) {
39              // creating number element
40              element = document.createElement( NUMBER_TAG_NAME );
41              // setting attribute
42              element.setAttribute( VALUE_ATTRIBUTE, currentNode.getName() );
43          } else if ( currentNode.getMathObject().equals( EMathObject.VARIABLE ) ) {
44              // creating variable element
45              element = document.createElement( VARIABLE_TAG_NAME );
46              // setting attribute
47              element.setAttribute( VALUE_ATTRIBUTE, currentNode.getName() );
48          }
49          return element;
50      }
```

C.5 Implémentation des processus d'insertion et d'extractions (conversions Flat-Tuple)

C.5.1 Processus d'insertion

```
1   /**
2    * Insert into a new equation into the warehouse, read from an XML string
3    *
4    * @param xmlString
5    */
6   @Override
7   public void putEquation( String xmlString ) {
8       // get the list of unavailable ids from database
9       ArrayList<Long> differentialEquationsUnavailableIds = getUnavailableIds(
10              "SELECT " + ETableName.DIFFERENTIAL_EQUATION_TABLE.value() + "." +
                    EAttributeName.ID_ATTRIBUTE.value()
11                  + " FROM " + ETableName.DIFFERENTIAL_EQUATION_TABLE.value() + "
                        ORDER BY "
12                  + EAttributeName.ID_ATTRIBUTE.value() + ";" );
13      ArrayList<Long> nodesUnavailableIds = getUnavailableIds(
14              "SELECT " + ETableName.NODE_TABLE.value() + "." + EAttributeName.
                    ID_ATTRIBUTE.value() + " FROM "
15                  + ETableName.NODE_TABLE.value() + " ORDER BY " + EAttributeName.
                        ID_ATTRIBUTE.value() + ";" );
16      ArrayList<Long> inputFunctionsUnavailbleIds = getUnavailableIds( "SELECT "
17              + ETableName.INPUT_FUNCTION_TABLE.value() + "." + EAttributeName.
                    ID_ATTRIBUTE.value() + " FROM "
18              + ETableName.INPUT_FUNCTION_TABLE.value() + " ORDER BY " + EAttributeName
                    .ID_ATTRIBUTE.value() + ";" );
19      ArrayList<Long> initialValuesUnavailableIds = getUnavailableIds( "SELECT "
```

```
20          + ETableName.INITIAL_VALUE_TABLE.value() + "." + EAttributeName.
                ID_ATTRIBUTE.value() + " FROM "
21          + ETableName.INITIAL_VALUE_TABLE.value() + " ORDER BY " + EAttributeName.
                ID_ATTRIBUTE.value() + ";" );
22      ArrayList<Long> variableValuesUnavailableIds = getUnavailableIds( "SELECT "
23          + ETableName.VARIABLE_VALUE_TABLE.value() + "." + EAttributeName.
                ID_ATTRIBUTE.value() + " FROM "
24          + ETableName.VARIABLE_VALUE_TABLE.value() + " ORDER BY " + EAttributeName
                .ID_ATTRIBUTE.value() + ";" );
25      ArrayList<Long> groupsUnavailableIds = getUnavailableIds(
26          "SELECT " + ETableName.GROUP_TABLE.value() + "." + EAttributeName.
                ID_ATTRIBUTE.value() + " FROM "
27              + ETableName.GROUP_TABLE.value() + " ORDER BY " + EAttributeName.
                    ID_ATTRIBUTE.value() + ";" );
28
29      // mapping XML attributes into FlatDifferentialEquation object
30      XMLFileFactory factory = XMLFileFactory.getInstance( XSD_FILE_PATH );
31      try {
32          FlatDifferentialEquation flateqdiff = factory.XMLread(
33                  xmlString.replaceAll( "[\\t\\n\\r]", "" ),
34                  differentialEquationsUnavailableIds, nodesUnavailableIds,
                        initialValuesUnavailableIds,
35                  inputFunctionsUnavailbleIds, variableValuesUnavailableIds,
                        groupsUnavailableIds );
36
37          // checking if newly read elements does not already exist in the
38          // database
39          PreparedStatement statement;
40
41          // variables checking
42          HashMap<Long, Variable> replaceVariables = new HashMap<Long, Variable>();
43          for ( Long id : flateqdiff.getVariables().keySet() ) {
44              statement = connection.prepareStatement(
45                      "SELECT * FROM " + ETableName.VARIABLE_VALUE_TABLE.value() + "
                            WHERE "
46                          + EAttributeName.NAME_ATTRIBUTE.value()
47                          + " = ? AND " + EAttributeName.VARIABLE_VALUE_ATTRIBUTE.
                                value() + " = ?;" );
48              statement.setString( 1, flateqdiff.getVariables().get( id ).getName() );
49              statement.setDouble( 2, flateqdiff.getVariables().get( id ).getValue() );
50              logger.debug( statement.toString() );
51              ResultSet result = statement.executeQuery();
52
53              if ( result.next() ) {
54                  logger.debug( result.getLong( 1 ) + " " + result.getString( 2 ) + " "
                            + result.getDouble( 3 ) );
55                  replaceVariables.put( id, new Variable( result ) );
56              }
57          }
58
```

```
59          for ( Long id : replaceVariables.keySet() ) {
60              flateqdiff.getVariables().put( replaceVariables.get( id ).getId(),
61                      replaceVariables.get( id ) );
62              flateqdiff.getVariables().remove( id );
63              flateqdiff.getVariables().get( replaceVariables.get( id ).getId() ).
                    setExist( true );
64          }
65
66          logger.debug( flateqdiff.getVariables().toString() );
67
68          // initial values checking
69          Iterator<Entry<Long, InitialValue>> iterator = flateqdiff.getInitialValues().
                entrySet().iterator();
70          HashMap<Long, InitialValue> replaceInitialValues = new HashMap<Long,
                InitialValue>();
71          while ( iterator.hasNext() ) {
72              Entry<Long, InitialValue> entry = iterator.next();
73              Long id = entry.getKey();
74              statement = connection.prepareStatement(
75                      "SELECT * FROM " + ETableName.INITIAL_VALUE_TABLE.value() + "
                            WHERE "
76                              + EAttributeName.NAME_ATTRIBUTE.value()
77                              + " = ? AND " + EAttributeName.DERIV_ATTRIBUTE.value() + "
                                    = ? AND "
78                              + EAttributeName.INITIAL_VALUE_ATTRIBUTE.value() + " = ?;"
                                    );
79              statement.setString( 1, flateqdiff.getInitialValues().get( id ).getName()
                     );
80              statement.setInt( 2, flateqdiff.getInitialValues().get( id ).getDeriv() )
                    ;
81              statement.setDouble( 3, flateqdiff.getInitialValues().get( id ).
                    getInitialValue() );
82              logger.debug( statement.toString() );
83              ResultSet result = statement.executeQuery();
84
85              if ( result.next() ) {
86                  logger.debug( result.getLong( 1 ) + " " + result.getString( 2 ) + " "
                         + result.getInt( 3 ) + " "
87                            + result.getDouble( 4 ) );
88                  replaceInitialValues.put( id, new InitialValue( result ) );
89              }
90          }
91
92          for ( Long id : replaceInitialValues.keySet() ) {
93              flateqdiff.getInitialValues().put( replaceInitialValues.get( id ).getId()
                    ,
94                      replaceInitialValues.get( id ) );
95              flateqdiff.getInitialValues().remove( id );
96              flateqdiff.getInitialValues().get( replaceInitialValues.get( id ).getId()
                     ).setExist( true );
```

```
97          }
98
99          logger.debug( flateqdiff.getInitialValues().toString() );
100
101         IdGenerator nodeContentIdGenerator = new IdGenerator( getUnavailableIds(
102                 "SELECT " + ETableName.NODE_CONTENT_TABLE.value() + "." +
                        EAttributeName.ID_ATTRIBUTE.value()
103                     + " FROM " + ETableName.NODE_CONTENT_TABLE.value() + ";" ) );
104
105         // nodes content
106         for ( Long id : flateqdiff.getFormula().keySet() ) {
107             statement = connection.prepareStatement( "SELECT * FROM " + ETableName.
                    NODE_CONTENT_TABLE.value()
108                     + " WHERE " + EAttributeName.MATH_OBJECT_ATTRIBUTE.value() + " = ?
                            " + MATHEMATICAL_OBJECT_CAST
109                     + " AND " + EAttributeName.DERIV_ATTRIBUTE.value() + " = ? AND "
110                     + EAttributeName.NAME_ATTRIBUTE.value() + " = ?;" );
111             statement.setString( 1, flateqdiff.getFormula().get( id ).getMathObject()
                    .value() );
112             statement.setInt( 2, flateqdiff.getFormula().get( id ).getDeriv() );
113             statement.setString( 3, flateqdiff.getFormula().get( id ).getName() );
114             logger.debug( statement.toString() );
115             ResultSet result = statement.executeQuery();
116
117             if ( result.next() ) {
118                 logger.debug( result.getLong( 1 ) + " " + result.getString( 2 ) + " "
                            + result.getString( 3 )
119                         + " " + result.getInt( 4 ) );
120                 flateqdiff.getFormula().get( id ).setIdContent( result.getLong( 1 ) )
                        ;
121                 flateqdiff.getFormula().get( id ).setExist( true );
122             } else {
123                 flateqdiff.getFormula().get( id ).setIdContent(
                        nodeContentIdGenerator.getValidId() );
124             }
125         }
126
127         // group checking
128         statement = connection.prepareStatement( "SELECT * FROM " + ETableName.
                GROUP_TABLE.value() + " WHERE "
129                 + EAttributeName.GROUP_ATTRIBUTE.value() + " = ?;" );
130         statement.setString( 1, flateqdiff.getGroup() );
131         logger.debug( statement.toString() );
132         ResultSet tuples = statement.executeQuery();
133
134         if ( tuples.next() ) {
135             logger.debug( "( " + tuples.getLong( 1 ) + ", " + tuples.getString( 2 ) +
                        ")" );
136             flateqdiff.setIdGroup( tuples.getLong( 1 ) );
137             flateqdiff.setGroupExist( true );
```

```
138         }
139
140         logger.debug( flateqdiff.getFormula().toString() );
141
142         flateqdiff.updateAllExist();
143         // checkIdsValidity( flateqdiff );
144
145         if ( !flateqdiff.allExist() ) {
146             // insertions
147
148             // potential new group insertion
149             if ( !flateqdiff.groupExist() ) {
150                 statement = connection
151                         .prepareStatement( "INSERT INTO " + ETableName.GROUP_TABLE.
                            value() + " VALUES (?,?);" );
152                 statement.setLong( 1, flateqdiff.getIdGroup() );
153                 statement.setString( 2, flateqdiff.getGroup() );
154                 logger.debug( statement.toString() );
155                 // sendLogMessage( statement.toString() );
156                 statement.executeUpdate();
157             }
158
159             // equation ID
160             statement = connection.prepareStatement(
161                     "INSERT INTO " + ETableName.DIFFERENTIAL_EQUATION_TABLE.value() +
                        " VALUES (?,?,?,?);" );
162             statement.setLong( 1, flateqdiff.getId() );
163             statement.setInt( 2, flateqdiff.getOrder() );
164             statement.setString( 3, flateqdiff.getName() );
165             statement.setLong( 4, flateqdiff.getIdGroup() );
166             logger.debug( statement.toString() );
167             // sendLogMessage( statement.toString() );
168             statement.executeUpdate();
169
170             // insertions of the new node contents and nodes
171             for ( Long id : flateqdiff.getFormula().keySet() ) {
172                 if ( !flateqdiff.getFormula().get( id ).exist() ) {
173                     statement = connection.prepareStatement( "INSERT INTO " +
                            ETableName.NODE_CONTENT_TABLE.value()
174                             + " VALUES (?,?" + MATHEMATICAL_OBJECT_CAST + ",?,?);" );
175                     statement.setLong( 1, flateqdiff.getFormula().get( id ).
                            getIdContent() );
176                     statement.setString( 2, flateqdiff.getFormula().get( id ).
                            getMathObject().value() );
177                     statement.setString( 3, flateqdiff.getFormula().get( id ).getName
                            () );
178                     statement.setInt( 4, flateqdiff.getFormula().get( id ).getDeriv()
                            );
179                     logger.debug( statement.toString() );
180                     // sendLogMessage( statement.toString() );
```

```
181                 statement.executeUpdate();
182             }
183 
184             statement = connection.prepareStatement(
185                     "INSERT INTO " + ETableName.NODE_TABLE.value() + " VALUES
                        (?,?);" );
186             statement.setLong( 1, id );
187             statement.setLong( 2, flateqdiff.getFormula().get( id ).getIdContent
                    () );
188             logger.debug( statement.toString() );
189             // sendLogMessage( statement.toString() );
190             statement.executeUpdate();
191         }
192 
193         IdGenerator nodeConnectionIdGenerator = new IdGenerator(
194                 getUnavailableIds( "SELECT " + ETableName.NODE_CONNECTION_TABLE.
                        value() + "."
195                         + EAttributeName.ID_ATTRIBUTE.value() + " FROM "
196                         + ETableName.NODE_CONNECTION_TABLE.value() ) );
197 
198         // insertions of the new node connections
199         for ( Long id : flateqdiff.getFormula().keySet() ) {
200             statement = connection.prepareStatement(
201                     "INSERT INTO " + ETableName.NODE_CONNECTION_TABLE.value() + "
                        VALUES (?,?,?,?,?,?,?);" );
202             statement.setLong( 1, nodeConnectionIdGenerator.getValidId() );
203             statement.setLong( 2, flateqdiff.getId() );
204             statement.setLong( 3, id );
205             if ( flateqdiff.getFormula().get( id ).getParent() == 0 )
206                 statement.setNull( 4, Types.INTEGER );
207             else
208                 statement.setLong( 4, flateqdiff.getFormula().get( id ).getParent
                        () );
209             if ( flateqdiff.getFormula().get( id ).getLeft() == 0 )
210                 statement.setNull( 5, Types.INTEGER );
211             else
212                 statement.setLong( 5, flateqdiff.getFormula().get( id ).getLeft()
                        );
213             if ( flateqdiff.getFormula().get( id ).getRight() == 0 )
214                 statement.setNull( 6, Types.INTEGER );
215             else
216                 statement.setLong( 6, flateqdiff.getFormula().get( id ).getRight()
                         );
217             statement.setInt( 7, flateqdiff.getFormula().get( id ).getDepth() );
218             logger.debug( statement.toString() );
219             sendLogMessage( statement.toString() );
220             statement.executeUpdate();
221         }
222 
223         IdGenerator equationVariablesIdGenerator = new IdGenerator(
```

```
224                 getUnavailableIds( "SELECT " + ETableName.
                        EQUATION_VARIABLE_VALUE_TABLE.value() + "."
225                         + EAttributeName.ID_ATTRIBUTE.value() + " FROM "
226                         + ETableName.EQUATION_VARIABLE_VALUE_TABLE.value() ) );
227
228             // insertions of the new variables and links to the equation
229             for ( Long id : flateqdiff.getVariables().keySet() ) {
230                 if ( !flateqdiff.getVariables().get( id ).exist() ) {
231                     statement = connection.prepareStatement(
232                             "INSERT INTO " + ETableName.VARIABLE_VALUE_TABLE.value() +
                                " VALUES (?,?,?);" );
233                     statement.setLong( 1, id );
234                     statement.setString( 2, flateqdiff.getVariables().get( id ).
                            getName() );
235                     statement.setDouble( 3, flateqdiff.getVariables().get( id ).
                            getValue() );
236                     logger.debug( statement.toString() );
237                     // sendLogMessage( statement.toString() );
238                     statement.executeUpdate();
239                 }
240
241                 statement = connection.prepareStatement(
242                         "INSERT INTO " + ETableName.EQUATION_VARIABLE_VALUE_TABLE.
                            value() + " VALUES (?,?,?);" );
243                 statement.setLong( 1, equationVariablesIdGenerator.getValidId() );
244                 statement.setLong( 2, flateqdiff.getId() );
245                 statement.setLong( 3, id );
246                 logger.debug( statement.toString() );
247                 // sendLogMessage( statement.toString() );
248                 statement.executeUpdate();
249             }
250
251             IdGenerator equationInputsIdGenerator = new IdGenerator(
                    getUnavailableIds(
252                         "SELECT " + ETableName.EQUATION_INPUT_TABLE.value() + "." +
                            EAttributeName.ID_ATTRIBUTE.value()
253                             + " FROM " + ETableName.EQUATION_INPUT_TABLE.value() ) );
254
255             CSVFileFactory csvFactory = CSVFileFactory.getInstance();
256
257             // insertion of the input functions and links to the equation
258             for ( Long id : flateqdiff.getInputs().keySet() ) {
259                 statement = connection
260                         .prepareStatement( "INSERT INTO " + ETableName.
                            INPUT_FUNCTION_TABLE.value()
261                                 + " VALUES (?,?,?,?,?" + INTERPOLATION_FUNCTION_CAST +
                                    ");" );
262                 statement.setLong( 1, id );
263                 statement.setString( 2, flateqdiff.getInputs().get( id ).getName() );
264                 statement.setInt( 3, flateqdiff.getInputs().get( id ).getDeriv() );
```

```
265                 statement.setString( 4, flateqdiff.getInputs().get( id ).getSerialKey
                        () );
266                 statement.setString( 5, flateqdiff.getInputs().get( id ).
                        getInterpolationFunction().value() );
267                 logger.debug( statement.toString() );
268                 // sendLogMessage( statement.toString() );
269                 statement.executeUpdate();
270
271                 logger.debug( "writing in file..." );
272                 csvFactory.writeSeriesInFile( flateqdiff.getInputs().get( id ).
                        getSeries(),
273                         RESOURCES + flateqdiff.getInputs().get( id ).getSerialKey() +
                            CSV_FILE_FORMAT );
274                 logger.debug( "writing in file... done." );
275
276                 statement = connection.prepareStatement(
277                         "INSERT INTO " + ETableName.EQUATION_INPUT_TABLE.value() + "
                            VALUES (?,?,?);" );
278                 statement.setLong( 1, equationInputsIdGenerator.getValidId() );
279                 statement.setLong( 2, flateqdiff.getId() );
280                 statement.setLong( 3, id );
281                 logger.debug( statement.toString() );
282                 // sendLogMessage( statement.toString() );
283                 statement.executeUpdate();
284             }
285
286             IdGenerator equationInitialValuesIdGenerator = new IdGenerator(
                    getUnavailableIds(
287                     "SELECT " + ETableName.EQUATION_INITIAL_VALUE_TABLE.value() + "."
288                             + EAttributeName.ID_ATTRIBUTE.value() + " FROM "
289                             + ETableName.EQUATION_INITIAL_VALUE_TABLE.value() ) );
290
291             // insertion of the initial values and links to the equation
292             for ( Long id : flateqdiff.getInitialValues().keySet() ) {
293                 if ( !flateqdiff.getInitialValues().get( id ).exist() ) {
294                     statement = connection.prepareStatement(
295                             "INSERT INTO " + ETableName.INITIAL_VALUE_TABLE.value() + "
                                VALUES (?,?,?,?);" );
296                     statement.setLong( 1, id );
297                     statement.setString( 2, flateqdiff.getInitialValues().get( id ).
                            getName() );
298                     statement.setInt( 3, flateqdiff.getInitialValues().get( id ).
                            getDeriv() );
299                     statement.setDouble( 4, flateqdiff.getInitialValues().get( id ).
                            getInitialValue() );
300                     logger.debug( statement.toString() );
301                     // sendLogMessage( statement.toString() );
302                     statement.executeUpdate();
303                 }
304
```

```
305                 statement = connection.prepareStatement(
306                         "INSERT INTO " + ETableName.EQUATION_INITIAL_VALUE_TABLE.value
                            () + " VALUES (?,?,?);" );
307                 statement.setLong( 1, equationInitialValuesIdGenerator.getValidId() )
                        ;
308                 statement.setLong( 2, flateqdiff.getId() );
309                 statement.setLong( 3, id );
310                 logger.debug( statement.toString() );
311                 // sendLogMessage( statement.toString() );
312                 statement.executeUpdate();
313             }
314
315             // send new equation ID and Name data to notify clients about
316             // the new entry
317             logger.debug( "new insertion notification..." );
318             sender.publish( EQueueName.GET_EQUATIONS_ID_AND_NAMES_RESPONSE_QUEUE_NAME
                    .value(),
319                     Serializer.serialize( flateqdiff.getId(), flateqdiff.getName(),
                            flateqdiff.getGroup() ) );
320
321             Thread.sleep( 3000 );
322         }
323     } catch ( ParserConfigurationException | SAXException | IOException |
            SQLException | InterruptedException e ) {
324         e.printStackTrace();
325     }
326 }
327
328 private void checkIdsValidity( FlatDifferentialEquation flatEqDiff ) {
329     // get the list of unavailable ids from database
330     ArrayList<Long> differentialEquationsUnavailableIds = getUnavailableIds(
331             "SELECT " + ETableName.DIFFERENTIAL_EQUATION_TABLE.value() + "." +
                    EAttributeName.ID_ATTRIBUTE.value()
332                     + " FROM " + ETableName.DIFFERENTIAL_EQUATION_TABLE.value() + "
                            ORDER BY "
333                     + EAttributeName.ID_ATTRIBUTE.value() + ";" );
334     ArrayList<Long> nodesUnavailableIds = getUnavailableIds(
335             "SELECT " + ETableName.NODE_TABLE.value() + "." + EAttributeName.
                    ID_ATTRIBUTE.value() + " FROM "
336                     + ETableName.NODE_TABLE.value() + " ORDER BY " + EAttributeName.
                            ID_ATTRIBUTE.value() + ";" );
337     ArrayList<Long> inputFunctionsUnavailbleIds = getUnavailableIds( "SELECT "
338             + ETableName.INPUT_FUNCTION_TABLE.value() + "." + EAttributeName.
                    ID_ATTRIBUTE.value() + " FROM "
339             + ETableName.INPUT_FUNCTION_TABLE.value() + " ORDER BY " + EAttributeName
                    .ID_ATTRIBUTE.value() + ";" );
340     ArrayList<Long> initialValuesUnavailableIds = getUnavailableIds( "SELECT "
341             + ETableName.INITIAL_VALUE_TABLE.value() + "." + EAttributeName.
                    ID_ATTRIBUTE.value() + " FROM "
```

```
342             + ETableName.INITIAL_VALUE_TABLE.value() + " ORDER BY " + EAttributeName.
                    ID_ATTRIBUTE.value() + ";" );
343     ArrayList<Long> variableValuesUnavailableIds = getUnavailableIds( "SELECT "
344             + ETableName.VARIABLE_VALUE_TABLE.value() + "." + EAttributeName.
                    ID_ATTRIBUTE.value() + " FROM "
345             + ETableName.VARIABLE_VALUE_TABLE.value() + " ORDER BY " + EAttributeName
                    .ID_ATTRIBUTE.value() + ";" );
346     ArrayList<Long> groupsUnavailableIds = getUnavailableIds(
347             "SELECT " + ETableName.GROUP_TABLE.value() + "." + EAttributeName.
                    ID_ATTRIBUTE.value() + " FROM "
348                     + ETableName.GROUP_TABLE.value() + " ORDER BY " + EAttributeName.
                        ID_ATTRIBUTE.value() + ";" );
349
350     // check equation id
351     IdGenerator differentialEquationsIdsGenerator = new IdGenerator(
            differentialEquationsUnavailableIds );
352     if ( !differentialEquationsIdsGenerator.isValid( flatEqDiff.getId() ) ) {
353         long newId = differentialEquationsIdsGenerator.getValidId();
354         flatEqDiff.updateId( newId );
355     }
356
357     // check nodes ids
358     IdGenerator nodesIdsGenerator = new IdGenerator( nodesUnavailableIds );
359     for ( Long key : flatEqDiff.getFormula().keySet() ) {
360         if ( !nodesIdsGenerator.isValid( flatEqDiff.getFormula().get( key ).getId() )
                ) {
361             long newId = nodesIdsGenerator.getValidId();
362             Node node = flatEqDiff.getFormula().get( key );
363             flatEqDiff.getFormula().remove( key );
364             node.setId( newId );
365             flatEqDiff.getFormula().put( newId, node );
366         }
367     }
368
369     // check inputs ids
370     IdGenerator inputFunctionsIdGenerator = new IdGenerator(
            inputFunctionsUnavailbleIds );
371     for ( Long key : flatEqDiff.getInputs().keySet() ) {
372         if ( inputFunctionsIdGenerator.isValid( key ) ) {
373             long newId = inputFunctionsIdGenerator.getValidId();
374             Input input = flatEqDiff.getInputs().get( key );
375             flatEqDiff.getInputs().remove( key );
376             input.setId( newId );
377             flatEqDiff.getInputs().put( newId, input );
378         }
379     }
380
381     // check initial values
382     IdGenerator initialValuesIdGenerator = new IdGenerator(
            initialValuesUnavailableIds );
```

```
383     for ( Long key : flatEqDiff.getInitialValues().keySet() ) {
384         if ( !initialValuesIdGenerator.isValid( key ) ) {
385             long newId = initialValuesIdGenerator.getValidId();
386             InitialValue initial = flatEqDiff.getInitialValues().get( key );
387             flatEqDiff.getInitialValues().remove( key );
388             initial.setId( newId );
389             flatEqDiff.getInitialValues().put( newId, initial );
390         }
391     }
392 
393     // check variables ids
394     IdGenerator variablesIdGenerator = new IdGenerator( variableValuesUnavailableIds
          );
395     for ( Long key : flatEqDiff.getVariables().keySet() ) {
396         if ( !variablesIdGenerator.isValid( key ) ) {
397             long newId = variablesIdGenerator.getValidId();
398             Variable variable = flatEqDiff.getVariables().get( key );
399             flatEqDiff.getVariables().remove( key );
400             variable.setId( newId );
401             flatEqDiff.getVariables().put( newId, variable );
402         }
403     }
404 }
405 
406 /**
407  * Get a list of already used id's in the database.
408  *
409  * @param query
410  * @return
411  */
412 private ArrayList<Long> getUnavailableIds( String query ) {
413     ArrayList<Long> ids = new ArrayList<Long>();
414     PreparedStatement statement;
415     try {
416         statement = connection.prepareStatement( query );
417         logger.debug( statement.toString() );
418         ResultSet result = statement.executeQuery();
419 
420         while ( result.next() ) {
421             ids.add( result.getLong( EAttributeName.ID_ATTRIBUTE.value() ) );
422         }
423 
424         Collections.sort( ids );
425 
426         logger.debug( "ID's found: " + ids );
427     } catch ( SQLException e ) {
428         e.printStackTrace();
429     }
430     return ids;
431 }
```

C.5.2 Processus d'extraction

```
1  /**
2   * Retrieves the equation of the database for a given ID.
3   *
4   * @param id
5   * @return
6   * @throws IOException
7   * @throws SQLException
8   */
9  public FlatDifferentialEquation getEquation( Long id ) throws IOException,
       SQLException {
10     // the following string must be added to the queries
11     // ("schema_mmw.differential_equation.id = id")
12     String clause = ETableName.DIFFERENTIAL_EQUATION_TABLE.value() + "." +
           EAttributeName.ID_ATTRIBUTE.value()
13             + " = " + id;
14
15     // reads the request necessary for equation retrieval in the
16     // src/main/resources/queries/retrieval.sql file
17     File file = new File( RETRIEVAL_QUERIES_FILE_PATH );
18     FileReader fileReader = new FileReader( file );
19     BufferedReader bufferedReader = new BufferedReader( fileReader );
20     ArrayList<String> queries = new ArrayList<String>();
21     String line;
22     while ( ( line = bufferedReader.readLine() ) != null ) {
23         queries.add( line );
24     }
25     logger.debug( queries.toString() );
26
27     // requests
28     // equations measurements
29     PreparedStatement statement = connection
30             .prepareStatement( insertWhereClause( queries.get( EQUATIONS_MEASURES ),
                   clause ) );
31     logger.debug( statement.toString() );
32     sendLogMessage( statement.toString() );
33     ResultSet measurements = statement.executeQuery();
34
35     // nodes table request
36     statement = connection.prepareStatement( insertWhereClause( queries.get( NODES )
           , clause ) );
37     logger.debug( statement.toString() );
38     sendLogMessage( statement.toString() );
39     ResultSet nodesTable = statement.executeQuery();
40
41     // variables request
42     statement = connection.prepareStatement( insertWhereClause( queries.get(
           VARIABLES ), clause ) );
43     logger.debug( statement.toString() );
```

```
44      sendLogMessage( statement.toString() );
45      ResultSet variablesTable = statement.executeQuery();
46
47      // inputs request
48      statement = connection.prepareStatement( insertWhereClause( queries.get( INPUTS
            ), clause ) );
49      logger.debug( statement.toString() );
50      sendLogMessage( statement.toString() );
51      ResultSet inputsTable = statement.executeQuery();
52
53      // initial values request
54      statement = connection.prepareStatement( insertWhereClause( queries.get(
            INITIALS ), clause ) );
55      logger.debug( statement.toString() );
56      sendLogMessage( statement.toString() );
57      ResultSet initialValuesTable = statement.executeQuery();
58
59      // equations groups
60      statement = connection.prepareStatement( insertWhereClause( queries.get( GROUPS
            ), clause ) );
61      logger.debug( statement.toString() );
62      sendLogMessage( statement.toString() );
63      ResultSet groupsTable = statement.executeQuery();
64      // logger.debug( Serializer.serialize(
65      // groupsTable ) );
66
67      return processResults( measurements, nodesTable, variablesTable, inputsTable,
            initialValuesTable, groupsTable )
68              .get( id );
69  }
70
71  /**
72   * Maps the data of result sets into a list of FlatDifferentialEquation object.
73   *
74   * @see fr.ensma.lias.dbcoreapi.models.FlatDifferentialEquation
75   * @param measurements
76   * @param nodesTable
77   * @param variablesTable
78   * @param inputsTable
79   * @param initialValuesTable
80   * @param groupsTable
81   * @return
82   * @throws IOException
83   * @throws SQLException
84   */
85  private Map<Long, FlatDifferentialEquation> processResults( ResultSet measurements,
        ResultSet nodesTable,
86          ResultSet variablesTable, ResultSet inputsTable, ResultSet initialValuesTable
                , ResultSet groupsTable )
87          throws IOException, SQLException {
```

```
88      Map<Long, FlatDifferentialEquation> differentialEquations = new HashMap<Long,
            FlatDifferentialEquation>();
89      Long currentId;
90
91      // sorts out the nodes then variables then inputs functions and then
92      // initial values into a list of FlatDifferentialEquation object
93      // (one per equation)
94      // reading measurements
95      while ( measurements.next() ) {
96          currentId = measurements.getLong( EAttributeName.ID_ATTRIBUTE.value() );
97          differentialEquations.put( currentId, new FlatDifferentialEquation() );
98          differentialEquations.get( currentId ).setName(
99                  measurements.getString( EAttributeName.NAME_ATTRIBUTE.value() ).
                        replaceAll( "\\s+", "" ) );
100     }
101
102     // reading the nodes table
103     while ( nodesTable.next() ) {
104         // reads the id of the equation of the current node
105         currentId = nodesTable.getLong( EAttributeName.ID_EQUATION_ATTRIBUTE.value()
                ) ;
106         // the nodes informations are added to the corresponding
107         // equation object.
108         if ( differentialEquations.containsKey( currentId ) )
109             differentialEquations.get( currentId ).getFormula().put(
110                     nodesTable.getLong( EAttributeName.ID_NODE_ATTRIBUTE.value() ),
111                     new Node( nodesTable ) );
112     }
113
114     for ( Long id : differentialEquations.keySet() ) {
115         logger.debug( differentialEquations.get( id ).getFormula().toString() );
116     }
117
118     // reading the variables table
119     while ( variablesTable.next() ) {
120         currentId = variablesTable.getLong( EAttributeName.ID_EQUATION_ATTRIBUTE.
                value() );
121         // if the variable is related to an equation that is in the
122         // equations list, the said equation is fed with the variable
123         // data.
124         if ( differentialEquations.containsKey( currentId ) )
125             differentialEquations.get( currentId ).getVariables()
126                     .put( variablesTable.getLong( EAttributeName.ID_VARIABLE_ATTRIBUTE
                            .value() ),
127                             new Variable( variablesTable ) );
128     }
129
130     for ( Long id : differentialEquations.keySet() ) {
131         logger.debug( differentialEquations.get( id ).getVariables().toString() );
132     }
```

```
133
134     // the inputs are stored into CSV files, the path to the file is
135     // stored into the database. In order to retrieve the inputs value,
136     // the file path is read in the query result table and used to
137     // access the correct input file.
138     CSVFileFactory csvfactory = CSVFileFactory.getInstance();
139     while ( inputsTable.next() ) {
140         currentId = inputsTable.getLong( EAttributeName.ID_EQUATION_ATTRIBUTE.value()
                );
141         // if the input is related to an equation in the list, the
142         // equation is fed with the input data
143         if ( differentialEquations.containsKey( currentId ) ) {
144             // creates a new input
145             Input input = new Input( inputsTable );
146             // fill it with the values read in the csv file
147             input.setSeries( csvfactory.readSeriesFromFile(
148                     RESOURCES + inputsTable.getString( EAttributeName.
                            SERIAL_KEY_ATTRIBUTE.value() )
149                             + CSV_FILE_FORMAT ) );
150             // add it to the corresponding equation
151             differentialEquations.get( currentId ).getInputs()
152                     .put( inputsTable.getLong( EAttributeName.
                            ID_INPUT_FUNCTION_ATTRIBUTE.value() ), input );
153         }
154     }
155
156     for ( Long id : differentialEquations.keySet() ) {
157         logger.debug( differentialEquations.get( id ).getInputs().toString() );
158     }
159
160     // reading the initial values table
161     while ( initialValuesTable.next() ) {
162         currentId = initialValuesTable.getLong( EAttributeName.ID_EQUATION_ATTRIBUTE.
                value() );
163         if ( differentialEquations.containsKey( currentId ) )
164             differentialEquations.get( currentId ).getInitialValues().put(
165                     initialValuesTable.getLong( EAttributeName.ID_INITIAL_ATTRIBUTE.
                            value() ),
166                     new InitialValue( initialValuesTable ) );
167     }
168
169     for ( Long id : differentialEquations.keySet() ) {
170         logger.debug( differentialEquations.get( id ).getInitialValues().toString() )
                ;
171     }
172
173     // reading the groups table values
174     while ( groupsTable.next() ) {
175         currentId = groupsTable.getLong( EAttributeName.ID_EQUATION_ATTRIBUTE.value()
                );
```

```
176         if ( differentialEquations.containsKey( currentId ) ) {
177             differentialEquations.get( currentId )
178                     .setGroup( groupsTable.getString( EAttributeName.GROUP_ATTRIBUTE.
                        value() ).trim() );
179         }
180     }
181 
182     for ( Long id : differentialEquations.keySet() ) {
183         logger.debug( differentialEquations.get( id ).getGroup() );
184     }
185 
186     logger.debug( differentialEquations.toString() );
187 
188     return differentialEquations;
189 }
```

Bibliographie

[AAB+08] R. Agrawal, A. Ailamaki, P. Bernstein, E. Brewer, M. Carey, S. Chaudhuri, A. Doan, D. Florescu, M. Franklin, H. Garcia-Molina, J. Gehrke, L. Gruenwald, L. Haas, A. Halevy, J. Hellerstein, Y. Ioannidis, H. Korth, D. Kossmann, S. Madden, R. Magoulas, B. Ooi, T. O'Reilly, R. Ramakrishnan, S. Sarawagi, M. Stonebraker, A. Szalay, and G. Weikum. The claremont report on database research. *SIGMOD Rec.*, 37(3) :9–19, 2008.

[ABC+14] R. Ausbrooks, S. Buswell, D. Carlisle, G. Chavchanidze, S. Dalmas, S. Devitt, A. Diaz, S. Dooley, Roger Hunter, P. Ion, M. Kohlhase, A. Lazrek, P. Libbrecht, B. Miller, R. Miner, C. Rowley, M. Sargent, B. Smith, N. Soiffer, R. Sutor, and S. Watt. *Mathematical Markup Language (MathML) Version 3.0.* 2nd edition, 2014.

[Aca07] Vincent Acary. Généralités sur les équations différentielles. `https://bipop.inrialpes.fr/people/acary/Teaching/Chapitre1.pdf`, 2007. [accessed : 04/11/2015].

[ACFR02] Serge Abiteboul, Sophie Cluet, Guy Ferran, and Marie-Christine Rousset. The xyleme project. *Computer Networks*, 39(3) :225–238, 2002.

[ACRWW14] Bikash Agrawal, Antorweep Chakravorty, Chunming Rong, and Tomasz Wiktor Wlodarczyk. R2time : A framework to analyse open tsdb time-series data in hbase. In *Proceedings of 2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 970 – 975, 2014.

[Ada06] C. Adamson. *Mastering Data Warehouse Aggregates : Solutions for Star Schema Performance.* Wiley Publishing, Inc., third edition, 2006.

[AGM+14] Julien Aligon, Matteo Golfarelli, Patrick Marcel, Stefano Rizzi, and Elisa Turricchia. Similarity measures for OLAP sessions. *Knowl. Inf. Syst.*, 39(2) :463–489, 2014.

[AGR15] Jason Arnold, Boris Glavic, and Ioan Raicu. Hrdbms : A newsql database for analytics. In *Proceedings of the 2015 IEEE International Conference on Cluster Computing*, CLUSTER '15, pages 519–520, Washington, DC, USA, 2015. IEEE Computer Society.

[APWZ95] R. Agrawal, G. Psaila, E. Wimmers, and M. Zaït. Querying shapes of histories. In *Proceedings of the 21st VLBD Conference*, pages 502–514, 1995.

[ATB+] Y. Abduallah, T. Turki, K. Byron, Z. Du, M. Cervantes-Cervantes, and J. Wang. Mapreduce algorithms for inferring gene regulatory networks from time-series microarray data using an information-theoretic approach. *BioMed Research International*, 2017.

[Aut17] The OpenTSDB Authors. How does opentsdb work ? `http://opentsdb.net/overview.html`, 2017. [accessed : 22/01/2018].

[AVM15] S. Agrawal, J. Verma, and B. Mahidhariya. Survey on mongodb : an open-source document database. *International Journal of Advanced Research in Engineering and Technology*, 6 :1 – 11, 2015.

[Bar14] D. Bartholomew. *MariaDB Cookbook*. PACKT, 2014.

[BFG+06] C. Ballard, D. M. Farrell, A. Gupta, C Mazuela, and S. Vohnik. *Dimensional Modeling : In a Business Intelligence Environment*. IBM Corp., first edition, 2006.

[BHS+98] C. Ballard, D. Herreman, D. Schau, R. Bell, E. Kim, and A. Valencic. *Data Modeling Techniques for Data Warehousing*. IBM, 1998.

[BMCDV+06] Claudia Bauzer-Medeiros, Olivier Carles, Florian De Vuyst, Bernard Hugueney, Marc Joliveau, Geneviève Jomier, Maude Manouvrier, Yosr Naija, Gérard Scémama, and Laurent Steffan. Vers un entrepôt de données pour le trafic routier. In *Entrepôt de données et Analyse en Ligne EDA'06*, Versailles, France, 2006.

[BO17] Z. Bicevska and I. Oditis. Towards nosql-based data warehouse solutions. *Procedia Computer Science*, 104(Supplement C) :104 – 111, 2017. ICTE 2016, Riga Technical University, Latvia.

[Bou16] S. Bouarar. Vers une conception logique et physique des bases de données avancées dirigée par la variabilité, 2016.

[Bra14] David Brailsford. Reverse polish grows on trees. `https://www.youtube.com/watch?v=TrfcJCulsF4`, 2014. [video posted on : 28/05/2014].

[BRK+06] A. Bagnall, C. Ratanamahatana, E. Keogh, S. Lonardi, and G. Janacek. A bit level representation for time series data mining with shape based similarity. *Data Mining Knowledge Discovery (DMKD)*, 13(1) :11–40, 2006.

[BRLTZ70] Yijian Bai, Chang R. Luo, Hetal Thakkar, and Carlo Zaniolo. Efficient support for time series queries in data stream management systems. 30 :113–132, 1970.

[Bru94] R. Bruchez. *Les bases de données NoSQL et le Big Data*. 2e edition, 1994.

[BT11] INC BASHO TECHNOLOGIES. Nosql optimized for time series and iot data. `http://basho.com/wp-content/uploads/2015/09/Basho-Riak-TS-Datasheet.pdf`, 2011. [accessed : 31/01/2018].

[Bun97] Peter Buneman. Semistructured data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 117–121, 1997.

[Bur20] D. Burleson. The hierarchical database model : Oracle database tips. `http://www.dba-oracle.com/t_object_hierarchical_database.htm`, 20. [accessed : 09/01/2018].

[Car15] O. Carton. L'essentiel de xml : Cours xml. `https://www.irif.fr/~carton/Enseignement/XML/Cours/support.pdf`, 2015. [accessed : 04/10/2016].

[Cas06] A. Castillejos. *Management of Time Series Data*. PhD thesis, School of Information Sciences and Engineering University of Canberra, 2006.

[Cel04] J. Celko. *Joe Celko's Trees and Hierachies in SQL for Smarties.* Morgan Kaufman, 2004.

[Cha] S.K. Chang. Hierarchical model, ims and mumps concepts. [Online ; accessed 09/01/2018].

[Cha14] K. Challapalli. The internet of things : A time series data challenge. `http://www.ibmbigdatahub.com/blog/internet-things-time-series-data-challenge`, 2014. [accessed : 02/02/2018].

[Che76] Peter Pin-Shan Chen. The entity-relationship model&mdash ;toward a unified view of data. *ACM Trans. Database Syst.*, 1(1) :9–36, March 1976.

[Che02] Peter Chen. Software pioneers. chapter Entity-relationship Modeling : Historical Events, Future Trends, and Lessons Learned, pages 296–310. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[Chr14] Zingg Christian. Reverse polish notation. `http://lec.inf.ethz.ch/ifmp/2014/ex_class_materials/week12/Reverse_Polish_Notation%20(EN).pdf`, 2014. [accessed : 06/03/2018].

[CKF16] M. Christ, A. W. Kempa-Liehr, and M. Feindt. Distributed and parallel time series feature extraction for industrial big data applications. *ArXiv e-prints*, 2016.

[CN16] Cláudio E. C. Campelo and Laércio Massaru Namikawa, editors. *Big data streaming for remote sensing time series analytics using MapReduce*, Campos de Jordao, Brazil, 2016. MCTIC/INPE.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6) :377–387, June 1970.

[Cod72] E. Codd. *Relational completeness of data base sublanguages*, volume 6, pages 65–98. N.J. : Prentice Hall, 1972.

[Cod82] E. F. Codd. Relational database : A practical foundation for productivity. *Commun. ACM*, 25(2) :109–117, 1982.

[Coë17a] Thomas Coëffé. Chiffres facebook - 2017. `https://www.blogdumoderateur.com/chiffres-facebook/`, 2017. [accessed : 16/02/2018].

[Coë17b] Thomas Coëffé. Chiffres twitter – 2017. `https://www.blogdumoderateur.com/chiffres-twitter/`, 2017. [accessed : 16/02/2018].

[CP14] G. Chakraborty and M. Pagolu. Analysis of unstructured data : Applications of text analytics and sentiment mining. In *Proceedings of SAS Global Forum 2014*. SAS Institute Inc., 2014.

[CPMC16] Sorin Ciolofan, Florin Pop, Mariana Mocanu, and Valentin Cristea. Rapid parallel detection of distance-based outliers in time series using mapreduce. *Control Engineering and Applied Informatics*, 18 :63–71, 10 2016.

[Cra17] Crate.io. Cratedb for time series. `http://go.cratedb.com/rs/832-QEZ-801/images/CrateDB-vs-Specialized-Time-Series-Databases.pdf`, 2017. [accessed : 15/02/2018].

[CZM+04] Josephine Chen, Po Zhao, Donald Massaro, Linda B. Clerch, Richard R. Almon, Debra C. DuBois, William J. Jusko, and Eric P. Hoffman. The pepr genechip data warehouse, and implementation of a dynamic time series query tool (sgqt) with graphical interface. *Nucleic Acids Research*, 32(suppl_1) :D578–D581, 2004.

[Dah17] Ehsan Akbari Dahouei. A cloud-based dashboard for time series analysis on hot topics from social media. In *Proceedings of the IEEE International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, pages 1428–1435. IEEE Computer Society, 2017.

[DBB16] Zouhir Djilani, Nabila Berkani, and Ladjel Bellatreche. Towards functional requirements analytics. In *7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 358–373, 2016.

[DDS94a] Werner Dreyer, Angelika Kotz Dittrich, and Duri Schmidt. An object-oriented data model for a time series management system. In *Proceedings of the 7th International Conference on Scientific and Statistical Database Management*, SSDBM'1994, pages 186–195, Washington, DC, USA, 1994. IEEE Computer Society.

[DDS94b] Werner Dreyer, Angelika Kotz Dittrich, and Duri Schmidt. Research perspectives for time series management systems. *SIGMOD Rec.*, 23(1) :10–15, 1994.

[Dew00] Mike Dewar. Openmath : An overview. *SIGSAM Bull.*, 34(2) :2–5, 2000.

[DF15] T. Dunning and E. Friedman. *Time Series Databases : New Ways to Store and Access Data.* O'Reilly Media, Inc., 2015.

[DM17] Jérôme Darmont and Patrick Marcel. Entrepôts de données et olap, analyse et décision dans l'entreprise. In *CNRS Editions. Les Big Data à découvert*, pages 132–133, 2017.

[DNS04] D. Draheim, W. Neun, and D. Suliman. Classifying differential equations on the web. *Proc. of MKM 2004*, pages 104–115, 2004.

[dru] About druid. `http://druid.io/druid.html`. [accessed : 22/01/2018].

[dru17] druid.io. Druid. `https://github.com/druid-io/druid/`, 2017. [accessed : 12/02/2018].

[DSK10] J. Durech, V. Sidorin, and M. Kaasalainen. DAMIT : a database of asteroid models. *Astronomy and Astrophysics*, 513, 2010.

[Dur08] Franz Durst. *Similarity Theory*, pages 193–219. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[EA12] P. Esling and C. Agón. Time-series data mining. *ACM Comput. Surv.*, 45(1) :12 :1–12 :34, December 2012.

[Ell13] Robert Elliott, editor. *The Data Warehouse Toolkit*, pages 1–35. Wiley Publishing, Inc., third edition, 2013.

[EN10] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems.* Addison-Wesley Publishing Company, USA, 6th edition, 2010.

[Fin09] T. Finch. Incremental calculation of weighted mean and variance. http://people.ds.cam.ac.uk/fanf2/hermes/doc/antiforgery/stats.pdf, 2009. [accessed : 04/04/2016].

[Foc18] Micro Focus. Overview of vertica. https://www.vertica.com/overview/, 2018. [accessed : 22/01/2018].

[Fou14] The Apache Software Foundation. Welcome to apache hadoop. https://hadoop.apache.org/, 2014. [accessed : 26/01/2018].

[Fou16] The Apache Software Foundation. Apache cassandra. https://cassandra.apache.org/, 2016. [accessed : 26/01/2018].

[Fou17] The Apache Software Foundation. Apache cassandra. https://github.com/apache/cassandra, 2017. [accessed : 12/02/2018].

[Fou18] The Apache Software Foundation. Welcome to apache hbase, 2018. [Online ; accessed 26/01/2018].

[Fre17] M. Freedman. Time-series data : Why (and how) to use a relational database instead of nosql. https://blog.timescale.com/time-series-data-why-and-how-to-use-a-relational-database-instead-of-nosql-d0cd6975e87c, 2017. [accessed : 19/01/2018].

[Fu11] T.-C. Fu. A review on time series data mining. *Engineering Applications of Artificial Intelligence*, 24(1) :164–181, 2011.

[Gar94] G. Gardarin. *Maitriser les bases de données : modèles et languages*. Second edition, 1994.

[GBKF15] T. Gunther, C. Buttner, A. Kasbohrer, and M. Filter. Development of a demonstrator for web based community-resource on mathematical models in the field of plant protection. *Communications in agricultural and applied biological sciences*, 80, 2015.

[GHS16] W. Giernacki, D. Horla, and T. Sadalla. Mathematical models database (MMD ver. 1.0). non-commercial proposal for researchers. In *Proceedings of the 21st International Conference on Methods and Models in Automation and Robotics*, pages 555 – 558, 2016.

[GHTC13] K. Grolinger, W. Higashino, A. Tiwari, and M. Capretz. Data management in cloud environments : Nosql and newsql data stores. *Journal of Cloud Computing : Advances, Systems and Applications*, 2(1) :49 :1–49 :24, 2013.

[GMUW09] H. Garcia-Molina, J. Ullman, and J. Widom. *Database systems : the complete book*. Prentice Hall, second edition, 2009.

[GR09] Matteo Golfarelli and Stefano Rizzi. *Data warehouse design : Modern principles and methodologies*, volume 5. Mcgraw-hill New York, 2009.

[GZLW09] A. Guazzelli, M. Zeller, W. Lin, and G. Williams. Pmml : An open standard for sharing models. *The R Journal*, 1 :60–65, 2009.

[Ham15] Hendrik F. Hamann. From smart sensors to smarter solutions with physical analytics. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, pages 3–3, New York, NY, USA, 2015. ACM.

[Hel79] J. Heler. Data management system (dms) for time series. 22 :0122–0125, 1979.

[HGW+95] Joachim Hammer, Hector Garcia-Molina, Jennifer Widom, Wilburt Labio, and Yue Zhuge. The stanford data warehousing project. *IEEE Data Eng. Bull.*, 18(2) :41–48, 1995.

[HGZ+16] Y. Hu, V. Gunapati, P. Zhao, D. Gordon, N. Wheeler, M. Hossain, T. Peshek, L. Bruckman, G.-Q. Zhang, and R. French. A nonrelational data warehouse for the analysis of field and laboratory data from multiple heterogeneous photovoltaic test sites. *IEEE Journal of Photovoltaics*, 2016.

[HSK+98] I. Halilovic, J. Simmonds, K., J. Klebanoff, and J. Luo. Informix timeseries datablade module : User's guide. `http://www.oninit.com/manual/informix/english/docs/datablade/5059.pdf`, 1998. [accessed : 06/07/2018].

[IHPZ14] Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. Towards exploratory OLAP over linked open data - A case study. In *International Workshop on Enabling Real-Time Business Intelligence (BIRTE)*, pages 114–132, 2014.

[Inc17] Quandl Inc. Quandl homepage. `https://www.quandl.com/`, 2017. [accessed : 31/01/2018].

[Inc18] Kaggle Inc. The home of data science & machine learning. `https://www.kaggle.com/`, 2018. [accessed : 31/01/2018].

[Inm02a] W. H. Inmon. *Building the Data Warehouse*. John Wiley & Sons, Inc., third edition, 2002.

[Inm02b] William H. Inmon. *Building the data warehouse*. J. Wiley, 2002.

[Int] ECMA International. The json data interchange syntax.

[JPT17] Søren Jensen, Torben Pedersen, and Christian Thomsen. Time series management systems : A survey. *IEEE Transactions on Knowledge and Data Engineering*, PP :1–1, 2017.

[jso] Json : The fat-free alternative to xml. `http://www.json.org/xml.html`. [accessed : 25/01/2018].

[Kai15] KairosDB. Kairosdb. `http://kairosdb.github.io/`, 2015. [accessed : 22/01/2018].

[KAL+17] Fedor Kazantsev, Ilya Akberdin, Sergey Lashin, Natalia Ree, Vladimir Timonov, Alexander Ratushny, Tamara Khlebodarova, and Vitaly Likhoshvai. Mammoth : A new database for curated mathematical models of biomolecular systems. *Journal of Bioinformatics and Computational Biology*, 16 :1740010, 2017.

[Kar10] B. Karwin. *SQL Antipattern : Avoiding the Pitfalls of Database Programming*. Pragmatic Bookshelf, 2010.

[KC14] Rakesh Kumar and Shilpi Charu. Newsql databases : Scalable rdbms for oltp needs to handle big data. *International Journal of Modern Computer Science (IJMCS)*, 3 :13–17, 2014.

[KFK+14] Maha Ben Kraiem, Jamel Feki, Kaïs Khrouf, Franck Ravat, and Olivier Teste. OLAP of the tweets : From modeling toward exploitation. In *IEEE 8th International Conference on Research Challenges in Information Science, RCIS*, pages 1–10, 2014.

[KGCJ14] Rakesh Kumar, Neha Gupta, Shilpi Charu, and Sunil Jangir. Manage big data through newsql. In *Proceedings of the National Conference on Innovation in Wireless Communication and Networking Technology*, Jaipur, India, 2014.

[KGM+] R. Kumar, N. Gupta, H. Maharwal, S. Charu, and K. Yadav. Critical analysis of database management using newsql. *International Journal of Computer Sciences and Mobile Computing*, 3 :434 – 438.

[KK] J. Kumar and S. Kumar. Big data time series analysis : An approach with the size of the time series. *International Journal of Advance Research in Computer Science and Software Engineering*, 7 :767 – 774.

[KK13] Mirko Kaempf and Jan Kantelhardt. Hadoop. ts : Large-scale time-series processing. *International Journal of Computer Applications*, 74 :1–8, 07 2013.

[KKA+17] Nikolaos Konstantinou, Martin Koehler, Edward Abel, Cristina Civili, Bernd Neumayr, Emanuel Sallinger, Alvaro A. A. Fernandes, Georg Gottlob, John A. Keane, Leonid Libkin, and Norman W. Paton. The VADA architecture for cost-effective data wrangling. In *ACM International Conference on Management of Data, SIGMOD Conference*, pages 1599–1602, 2017.

[KPG+] R. Kumar, B.B. Parashar, S. Gupta, Y. Sharma, and N. Gupta and. Apache hadoop, nosql and newsql solutions of big data. *International Journal of Advance Foundation and Research in Science & Engineering*, 1 :28 – 36.

[KXD12] A. Kasper, Z. Xue, and R. Dillmann. The KIT object database : An object model database for object recognition, localization and manipulation in service robotics. *International Journal of Robotics Research*, 31 :927 – 934, 2012.

[LAP14] P. Lacomme, S. Aridhi, and R. Phan. *Bases de données NoSQL et Big Data.* 2014.

[LFV+12] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database : C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12) :1790–1801, 2012.

[LGCE14] D. Labate, P. Guibbini, G. Chicco, and M. Ettore. Shape : A new business analytics web platform for getting insights on electrical load patterns. In *Congrès International des Réseaux Electriques de Distribution (CIRED'14) Worshop*, pages 1428–1435. IEEE Computer Society, 2014.

[LGCP15] D. Labate, P. Guibbini, G. Chicco, and F. Piglione. Shape : The load prediction and non-technical losses modules. In *Proceedings of the Congrès International des Réseaux Electriques de Distribution (CIRED'15)*, pages 1428–1435. IEEE Computer Society, 2015.

[LKL03] S. Lee, D. Kwon, and S. Lee. Dimensionality reduction for indexing time series based on the minimum distance. *Journal of Information Science and Engineering*, 19 :697–711, 2003.

[LLC18] Percona LLC. Percona tokudb. `https://www.percona.com/software/mysql-database/percona-tokudb`, 2018. [accessed : 22/01/2018].

[LMP10] W. Lang, M. Morse, and J.M. Patel. Dictionary-based compression for long time-series similarity. *IEEE Transactions on Knowledge and Data Engineering*, 22(11) :1609–1622, 2010.

[Mac16] Andrew MacDonald. Phildb : the time series database with built-in change logging. *PeerJ Computer Science*, 2 :e52, 2016.

[Mac17] Andrew MacDonald. Phildb project. `https://github.com/amacd31/phildb`, 2017. [accessed : 12/02/2018].

[Mat17] Houcine Matallah. Experimental comparative study of nosql databases : Hbase versus mongodb by ycsb. *International Journal of Computer Systems Science & Engineering*, 32, 07 2017.

[McI18] Mark McIlroy. Reverse polish notation. `http://mathworld.wolfram.com/ReversePolishNotation.html`, 2018. [accessed : 06/03/2018].

[MCV+06] Claudia Bauzer Medeiros, Olivier Carles, Florian De Vuyst, Georges Hébrail, Bernard Hugueney, Marc Joliveau, Geneviève Jomier, Maude Manouvrier, Yosr Naïja, Gérard Scémama, and Laurent Steffan. Vers un entrepôt de données pour le trafic routier. In *Actes de la 2ème journée francophone sur les Entrepôts de Données et l'Analyse en ligne, EDA 2006*, pages 119–138, 2006.

[Mon18] Inc. MongoDB. mongodb, for giant ideas. `https://www.mongodb.com/`, 2018. [accessed : 26/01/2018].

[MS09] H. Méloni and T. Spriet. Systèmes de gestion de bases de données : Le modèle hiérarchique. `http://e-ressources.univ-avignon.fr/sgbd/co/hierarchique.html`, 2009. [accessed : 03/01/2018].

[MSLB11] D. Morent, K. Stathatos, W. Lin, and M. Berthold. Comprehensive pmml preprocessing in knime. In *Proceedings of the 2011 Workshop on Predictive Markup Language Modeling*, (PMML), pages 28–31. ACM, 2011.

[MZ97] Iakovos Motakis and Carlo Zaniolo. Temporal aggregation in active database rules. *SIGMOD Rec.*, 26(2) :440–451, 1997.

[Nam15] D. Namiot. Time series databases. In *Proceedings of the XVII International Conference "Data Analytics and Management in Data Intensive Domains" (DAMDID/RCDL)*, pages 132–137, 2015.

[Nav92] Shamkant B. Navathe. Evolution of data modeling for databases. *Commun. ACM*, 35(9) :112–123, 1992.

[Noy18] Dan Noyes. The top 20 valuable facebook statistics. `https://zephoria.com/top-15-valuable-facebook-statistics/`, Last update 2018. [accessed : 15/02/2018].

[OO01] P. O'Neil and E. O'Neil. *Database–principles, Programming, and Performance.* The Morgan Kaufmann series in data management systems. Morgan Kaufmann Publishers, 2001.

[PA16] Andrew Pavlo and Matthew Aslett. What's really new with newsql ? *Special Interest Group on Management of Data Record*, 45(2) :45–55, 2016.

[PBB16] Cyrille Ponchateau, Ladjel Bellatreche, and Mickael Baron. Entrepôt de données dans l'ère data science : De la donnée au modèle. *Revue des Nouvelles Technologies de l'Information*, XIIe journées francophones sur les Entrepôts de Données et l Analyse en Ligne, RNTI-B-12 :65–80, 2016.

[PBG16] Jim Pivarski, Collin Bennett, and Robert L. Grossman. Deploying analytics with the portable format for analytics (PFA). In *Proceedings of the 22Nd International Conference on Knowledge Discovery and Data Mining*, (KDD), pages 579–588. ACM, 2016.

[PBOB16] C. Ponchateau, L. Bellatreche, C. Ordonnez, and M. Baron. A database model for time series : From a traditional data warehouse to a mathematical models warehouse. In *Actes de la conférence BDA 2016 Gestion de Données – Principes, Technologies et Applications*, Poitiers, France, 2016.

[PBOB17] C. Ponchateau, L. Bellatreche, C. Ordonnez, and M. Baron. Mathmouse : A mathematical models warehouse to handle both theoretical and numerical data. In *Actes de la conférence BDA 2017 Gestion de Données – Principes, Technologies et Applications*, Nancy, France, 2017.

[PBOB18] C. Ponchateau, L. Bellatreche, C. Ordonnez, and M. Baron. A mathematical database to process time series. *International Journal of Data Warehousing and Mining (IJDWM)*, 14(3) :1–21, 2018.

[Pec09] Rick Pechter. What's pmml and what's new in pmml 4.0 ? *SIGKDD Explor. Newsl.*, 11(1) :19–25, November 2009.

[PHM] Y. Perwej, M. Husamuddin, and F. Mazarbhuiya. An extensive investigate the mapreduce technology. *International Journal of Computer Sciences and Engineering*, 5 :218 – 225.

[PM08] L. Pujo-Menjouet. ntroduction aux équations différentielles et aux dérivées partielles. `http://math.univ-lyon1.fr/~pujo/coursintro-edo-edp.pdf`, 2008. [accessed : 04/11/2015].

[PP99] Chang-Shing Perng and D. Stott Parker. Sql/lpp : A time series extension of sql based on limited patience patterns. In Trevor J.M. Bench-Capon, Giovanni Soda, and A. Min Tjoa, editors, *Database and Expert Systems Applications*, pages 218–227, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[PTd16] T.-C. Phan, T.T.Q. Tran, and L. d'Orazio. Filtres pour jointures et requêtes récursives en mapreduce. In *Actes de la conférence BDA 2016 Gestion de Données – Principes, Technologies et Applications*, Poitiers, France, 2016.

[PTVF02] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C : The Art of Scientific Computing*. Cambridge University Press, second edition, 2002.

[PWB+11] U. Pieper, B. Webb, D. Barkan, D. Schneidman-Duhovny, A. Schlessinger, H. Braberg, Z. Yang, E. Meng, E. Pettersen, C. Huang, R. Datta, P. Sampathkumar, M. Madhusudhan, K. Sjolander, T. Ferrin, S. Burley, and A. Sali. Modbase, a database of annotated comparative protein structure models, and associated resources. *Nucleic Acids Research*, 39 :465 – 474, 2011.

[RCM+13] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. Addressing big data time series : Mining trillions of time series subsequences under dynamic time warping. *ACM Trans. Knowl. Discov. Data*, 7(3) :10 :1–10 :31, 2013.

[RDO01] E. Ramis, C. Deschamps, and J. Odoux. Courd de mathématiques : 4. séries et équations différentielles, 2001.

[RDR+98] Raghu Ramakrishnan, Donko Donjerkovic, Arvind Ranganathan, Kevin S. Beyer, and Muralidhar Krishnaprasad. Srql : Sorted relational query language. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, SSDBM'98, pages 84–95, Washington, DC, USA, 1998. IEEE Computer Society.

[RMA15] R. Rastogi, P. Mondal, and K. Agarwal. An exhaustive review for infix to postfix conversion with applications and benefits. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 95–100, 2015.

[Row06] Peter Rowlett. Mathml/xml series : What is mathml ? 6(1) :1–2, 2006.

[RS16] Franck Ravat and Jiefu Song. Unifying warehoused data with linked open data : A conceptual modeling solution. In *6th International Conference on Model and Data Engineering (MEDI)*, pages 245–259, 2016.

[RTB+06] Benjamin Ribba, Philippe Tracqui, Jean-Laurent Boix, El Manssouri Hanane, and Stephen Randall Thomas. Qxdb : A generic database to support mathematical modeling in biology. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 364 :1517–32, 2006.

[RTTZ07] Franck Ravat, Olivier Teste, Ronan Tournier, and Gilles Zurfluh. A conceptual model for multidimensional analysis of documents. In *26th International Conference on Conceptual Modeling*, pages 550–565, 2007.

[RWW+17] Sean Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. Littletable : A time-series database and its uses. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 125–138, New York, NY, USA, 2017. ACM.

[SBPR11] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The architecture of scidb. In *Proceedings of the 23rd International Conference on Scientific and Statistical Database Management*, SSDBM'11, pages 1–16, Berlin, Heidelberg, 2011. Springer-Verlag.

[SDDM95] Duri Schmidt, Angelika Kotz Dittrich, Werner Dreyer, and Robert W. Marti. Time series, A neglected issue in temporal database research ? In *Recent Advances in Temporal Databases, Proceedings of the International Workshop on Temporal Databases*, pages 214–232, 1995.

[SF13] P. Sadalage and M. Fowler. *NoSQL Distilled : A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, 2013.

[Soc17] The Open Math Society. *The Open Math Standard version 2.0*. 2017.

[SZ96] H. Shatkay and S Zdonik. Approximate queries and representations for large data sequences. In *Proceedings of the Twelfth International Conference on Data Engineering*, ICDE '96, pages 536–545, Washington, DC, USA, 1996. IEEE Computer Society.

[SZZA01] Reza Sadri, Carlo Zaniolo, Amir M. Zarkesh, and Jafar Adibi. A sequential pattern query language for supporting instant data mining for e-services. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 653–656, 2001.

[TYS+14] Wang Tuo, Sun Yunhua, Tian Song, Yu Liang, and Cui Weihong. Indoor air quality analysis based on hadoop. *IOP Conference Series : Earth and Environmental Science*, 17(1) :012260, 2014.

[Ull82] J. Ullman. *Database systems*. Computer software engineering series. Computer Science Press, Inc., 1982.

[Vas98] Panos Vassiliadis. Modeling multidimensional databases, cubes and cube operations. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, SSDBM '98, pages 53–62, Washington, DC, USA, 1998. IEEE Computer Society.

[w3c16] w3c. Extensible markup language (xml). `https://www.w3.org/XML/`, 2016. [accessed : 25/01/2018].

[w3s] w3schools.com. Xml dom tutorial. `https://www.w3schools.com/XML/dom_intro.asp`. [accessed : 28/03/2018].

[w3s18] w3schools. Introduction to sql. `https://www.w3schools.com/sql/sql_intro.asp`, 2018. [accessed : 29/01/2018].

[Wal70] Dr John Waldron. Reverse polish notation. `https://www.scss.tcd.ie/John.Waldron/3d1/rpn.pdf`, 1970. [accessed : 06/03/2018].

[WC06] Timothy W. Cole. Mathml in practice : issues and promise. 5 :209–218, 2006.

[WGH+14] S. Wimalaratne, P. Grenon, H. Hermjakob, N. Le Novere, and C. Laibe. Biomodels linked dataset. *BMC Systems Biology*, 8, 2014.

[WQS16] Mahdi Washha, Aziz Qaroush, and Florence Sedes. Impact of time on detecting spammers in twitter. In *Actes de la conférence BDA 2016 Gestion de Données – Principes, Technologies et Applications*, Poitiers, France, 2016.

[WSV96] Ulrich Wolf-Schumann and Stefan Vaillant. Timeview : A time series management system for gis and hydrological systems. pages 79 – 87, 01 1996.

[XBP08] Dung Nguyen Xuan, Ladjel Bellatreche, and Guy Pierra. Ontodawa, un système d'intégration à base ontologique de sources de données autonomes et évolutives. *Ingénierie des Systèmes d'Information*, 13(2) :97–125, 2008.

[YTL+14] F. Yang, E. Tscheltter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. Druid : A real-time analytical data store. In *SIGMOD '14 : Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2014. ACM.

Résumé – Les sciences expérimentales font régulièrement usage de séries chronologiques, pour représenter certains des résultats expérimentaux, qui consistent en listes chronologiques de valeurs (indexées par le temps), généralement fournies par des capteurs reliés à un système (objet de l'expérience). Ces séries sont analysées dans le but d'obtenir un modèle mathématique permettant de décrire les données et ainsi comprendre et expliquer le comportement du système étudié. De nos jours, les technologies de stockage et analyse de séries chronologiques sont nombreuses et matures, en revanche, quant au stockage et à la gestion de modèles mathématiques et leur mise en lien avec des données numériques expérimentales, les solutions existantes sont à la fois récentes, moins nombreuses et moins abouties. Or, les modèles mathématiques jouent un rôle essentiel dans l'interprétation et la validation des résultats expérimentaux. Un système de stockage adéquat permettrait de faciliter leur gestion et d'améliorer leur ré-utilisabilité. L'objectif de ce travail est donc de développer une base de modèles permettant la gestion de modèle mathématiques et de fournir un système de « requête par les données », afin d'aider à retrouver/reconnaître un modèle à partir d'un profil numérique expérimental. Dans cette thèse, je présente donc la conception (de la modélisation des données, jusqu'à l'architecture logicielle) de la base de modèles et les extensions qui permettent de réaliser le système de « requête par les données ». Puis, je présente le prototype de la base de modèle que j'ai implémenté, ainsi que les résultats obtenus à l'issu des tests de ce-dernier.

Mots clés : Bases de données relationnelles, Entrepôts de données, Bases de données–Conception, Modèles mathématiques–Bases de données, Séries chronologiques–Logiciels, Équations différentielles–Bases de donnees, Structures de données (informatique), Microservices, Comparaison série-équation.

Abstract – It is common practice in experimental science to use time series to represent experimental results, that usually come as a list of values in chronological order (indexed by time) and generally obtained via sensors connected to the studied physical system. Those series are analyzed to obtain a mathematical model that allow to describe the data and thus to understand and explain the behavior of the studied system. Nowadays, storage and analyses technologies for time series are numerous and mature, but the storage and management technologies for mathematical models and their linking to experimental numerical data are both scarce and recent. Still, mathematical models have an essential role to play in the interpretation and validation of experimental results. Consequently, an adapted storage system would ease the management and re-usability of mathematical models. This work aims at developing a models database to manage mathematical models and provide a "query by data" system, to help retrieve/identify a model from an experimental time series. In this work, I will describe the conception (from the modeling of the system, to its software architecture) of the models database and its extensions to allow the "query by data". Then, I will describe the prototype of models database, that I implemented and the results obtained by tests performed on the latter.

Keywords : Relational databases, Data warehousing, Mathematical models–Databases, Time-series analysis–Computer programs, Differential equations–Databases, Data structures (Computer science), Microservices, Series-equation comparison.

ENSMA, Téléport 2 – 1 avenue Clément Ader
86961 Futuroscope Chasseneuil Cedex