Université de Bretagne Occidentale

MASTER SIC SIAM

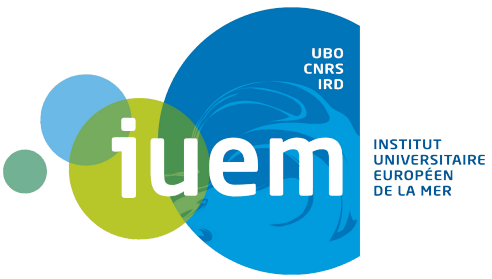
Système Informatique Complexe, Systèmes Informatiques et Applications Marines

2ème année

Rapport de Stage

Gestion des séries de données scientifiques.

COATANEA Brendan



Institut Universitaire Européen de la Mer
Technopôle Brest Iroise
Rue Dumont d'Urville
29200 Brest
France

Maître de stage :

Jonathan Schaeffer

Enseignant tuteur :

Jean-Philippe Babau

Année universitaire 2016-2017

Remerciements

Je tiens à remercier mon maître de stage Jonathan SCHAEFFER, responsable du service informatique au sein de l'IUEM, qui durant ces 6 mois de stage m'a permis d'apprendre énormément de chose sur des technologies, des outils ou des méthodes de travail que je ne connaissais pas. Mais aussi pour sa confiance en moi et en mes capacités à accomplir les différents missions du stage. Il fut d'une aide précieuse dans les moments les plus délicats.
Je remercie également les responsables scientifiques : Christine DAVID-BEAUSIRE et Peggy RIMMELIN-MAURY, pour leur aide à la compréhension du contexte scientifique, leur définitions des besoins des possibles utilisateurs du projet, .
J'en sais gré aux membres du FEIRI pour l'accueil chaleureux en leur sein, leurs aides et leurs conseils. Enfin je souhaite remercier L'Université de Bretagne Occidentale qui m'a permis d'effectuer ce stage ainsi que Gildas ROUDAULT pour avoir partagé mon CV au sein de son réseau du Technopôle de Plouzané.

Table des matières

N.B : Tous les mots écrit en italiques sont expliqués dans l'index. [1]

1. A la page 31

Introduction

Durant l'année de Master Système Informatique Complexe, Systèmes Informatiques et Applications Marines il est nécessaire d'effectuer un stage en entreprise d'une durée minimale de 24 semaines. Il permet d'attester la validation des acquis de l'année en cours ainsi que d'affirmer les capacités de l'étudiant à s'adapter au monde du travail, et à ses difficultés inhérentes.
J'ai donc effectué mon stage au sein de l'Institut Universitaire Européen de la Mer, dans la ville de Brest, du 03 Janvier 2017 au 30 Juin 2017.
J'ai pu prendre connaissance de l'existence de ce stage grâce à l'aide de Gildas ROUDAULT qui a partagé mon CV au sein de son réseau au Technopôle. Le sujet de ce stage m'a intéressé au plus haut point, car il était question d'un environnement de travail au sein du milieu de la recherche. Environnement que je n'avais jamais fréquenté et qui m'intéressait énormément, de par son aspect innovant et utile ainsi que de lien entre de nombreux cœurs de métier. Mon intérêt était aussi du au fait que le stage était basé, en grande partie, sur le framework Ruby on Rails que je ne connaissais pas encore et qui pouvait peut-être me réconcilier avec la programmation web.
L'objectif de ce stage était la création d'une interface de visualisation de données scientifiques, permettant leur exploration nécessaire à leur qualification et leur validation. L'IUEM possède en effet des séries de mesures effectuées en Antarctique, à la base scientifique française de Dumont d'Urville, depuis 1997 mais dont les résultats n'ont jamais été exploités.
Afin de vous présenter le résultat de cette expérience je vous rendrais compte de la situation actuelle de l'établissement d'accueil au sein duquel j'ai travaillé, puis je vous ferais part de l'objectif de la mission qui m'a été confiée, ainsi que du travail que j'ai effectué lors de ce stage.

1. Structure d'accueil

1.1 Présentation de l'entreprise

L'IUEM ou Institut Universitaire Européen de la Mer est un institut de recherche dédié à l'océan et au littoral fondé en 1997 par Paul Tréguer. Il regroupe de nombreuses disciplines, toutes tournées vers un même but la découverte des mystères du monde marin, depuis les simples vagues des côtes jusqu'au plus profondes abysses des grands fonds. Ses principaux objectifs sont les suivants :

- Accroître la connaissance du monde marin,
- Étudier et observer les interactions de ce monde marin avec l'atmosphère et les espaces continentaux,
- Former des chercheurs et des cadres dans ces domaines,
- Contribuer à l'observation des modifications, naturelles ou causées par l'homme dans ce milieu.

Ces objectifs ont été regroupés en 3 principales missions que l'institut effectue au jour le jour :

- Enseignement supérieur
- Recherche
- Observation

Pour ce qui est de l'enseignement l'IUEM possède le statut d'Ecole interne de l'UBO ainsi il assure des formations de master et de doctorat, seul en France, dans le domaine des sciences de la mer. Il existe au sein de l'institut deux masters, "Sciences de la mer et du littoral" ainsi que depuis 2010 "Energies marines renouvelables", et l'Ecole Doctorale des Sciences de la Mer. Les doctorants ont donc accès à des enseignements de haut niveau (cours, colloques, séminaires, écoles thématiques,...) mais de par sa présence au sein de l'institut également aux différents laboratoires de recherche.

Ces laboratoires s'occupent quant à eux de la mission de recherche de l'institut. En effet il s'agit d'un acteur majeur de la recherche française et européenne en sciences de la mer, de part son environnement regroupant la moitié du potentiel national dans ce domaine, ses équipes sont présentes sur toutes les grandes problématiques de recherche et travaillent sur tous les océans. La pluridisciplinarité étant un atout principal de l'IUEM, la recherche sur le milieu marin est effectuée sous tous les angles possibles comme : la physique, la chimie, la biologie, la biogéochimie, la géologie, la géographie, etc ...

L'Observatoire Marin de l'IUEM (Observatoire des Sciences de l'Univers – OSU) s'occupe lui de l'acquisition des données scientifiques nécessaires à la compréhension des états des milieux marins côtiers et hauturiers et de leurs réponses face aux changements globaux. Les membres

de celui-ci mesurent dans les milieux marins côtiers et hauturiers des paramètres biologiques, chimiques et physiques. Ainsi l'observatoire est d'une grande aide à la recherche de l'IUEM, mais aussi nationale, car son travail permet d'obtenir de nombreux résultats concrets et de tirer des conclusions sur l'évolution passée et future du milieu marin.

La composition de l'IUEM est assez complexe de part ses différentes missions, et regroupe de nombreuses entités en tous genres. Ainsi l'institut est une école interne de l'UBO, mais il regroupe par la même occasion l'ensemble de ses laboratoires dont l'objet de recherche est en lien avec la mer et les littoraux. L'IUEM est également un Observatoire des Sciences de l'Univers (OSU), dépendant de l'Institut National des Sciences de l'Univers (INSU) qui lui appartient au Centre National de la Recherche Scientifique (CNRS). L'Institut de Recherche pour le Développement (IRD) et l'Institut français de recherche pour l'exploitation de la mer (Ifremer) sont deux autres tutelles de l'IUEM qui interviennent au sein de l'observatoire ainsi que des laboratoires de recherches. En synthèse l'IUEM c'est :

- Des Unité Mixte de Recherche (UMR) ou laboratoires, qui sont aux nombre de six (LOPS, LEMAR, GEOMER, AMURE, LM2E, GEOSCIENCES OCEAN)
- Un laboratoire associé (LBCM)
- Trois Laboratoires Mixtes Internationaux (LMI)
- Une Unité Mixte de Service, tutellée par : l'UBO, le CNRS et l'IRD

L'IUEM est placé au site de la Pointe du Diable, au cœur du Technopôle Brest-Iroise, il dispose de plus de 10.000 m^2 de bureaux, laboratoires et salles de cours. Mais il est également présents au sein de l'UBO et du centre de l'Ifremer. Au niveau personnel, étant l'un des plus importants centres français de recherche marine, il possède plus de 400 chercheurs, enseignants-chercheurs, ingénieurs et techniciens permanents et environ 70 contractuels. Enfin il assure la formation d'environs 400 étudiants en master et doctorat.

1.2 L'environnement de travail

1.2.1 UMS3113

Mon stage s'est déroulé dans le cadre de la mission d'observation de l'IUEM, j'ai donc intégré l'UMS. L'Unité Mixte de Service regroupe les différentes professions permettant la mission d'observation de l'institut ainsi que l'aide à celle de recherche. Son activité comprends le service d'observation, l'administration centrale et la scolarité, la relation entreprise, les relations internationales, le service communication et enfin le service informatique. La demande de mon stage était principalement basée sur les besoins de l'UMS, en effet il s'agit du service qui récolte les données d'observations et qui nécessitait la création d'une interface de visualisation et de qualification.

1.2.2 FEIRI

Le FEIRI (Fédération d'Équipements Informatiques et Réseau à l'IUEM) est le service informatique de l'IUEM, grâce à l'aide de la DSI de l'UBO il met à disposition de multiples services aux utilisateurs de l'institut. Les différents services du FEIRI sont les suivants : [1]

- L'Assistance Informatique de Proximité

1. Vous pouvez trouver un schéma avec plus de détails en annexe à la page 35

— Le service web (création et mise à jour des sites, hébergement de sites, ...)
— Soutiens aux missions de l'IUEM (services métiers pour la recherche, l'enseignement et l'observation) : calcul scientifique et gestion des données scientifiques
— Gestion du parc micro-informatique
— Infrastructure des systèmes d'information (salles serveurs, serveurs, équipements réseau)
— Service interne

Le service est composé de sept personnes appartenant toutes à différentes structures en plus de l'UMS, comme l'UBO ou le CNRS. De plus chaque laboratoire possède un correspondant informatique qui sert d'intermédiaire entre le FEIRI et le laboratoire en question.

2. Présentation du sujet

2.1 Objectifs

L'objectif de ce stage était l'implémentation d'une interface web de visualisation, d'importation et de qualification des données scientifiques. Le service d'observation possède des séries de mesures de multiples paramètres effectuée en Antarctique durant une vingtaine d'années (1997 à 2016). Le matériel de mesure ayant évolué au cours des années les différents fichiers contenant les données n'avaient pas le même format, le même nombre de paramètres et le même degré de précision. Ainsi il était également nécessaire d'effectuer un premier travail de définition d'un format de fichiers commun à tous les fichiers qui simplifiera par la suite l'import. L'interface de visualisation devait permettre d'afficher différents paramètres et informations à propos des mesures (graphe des données, lieu, nombre de mesures, ...). La qualification des données quant à elle consiste en la définition d'un code qualité aux mesures, ce qui au niveau informatique demandait une interaction possible sur les données afin de modifier un attribut en base. Et enfin l'import des données est inscrit dans une démarche de continuité du site, en effet l'observatoire souhaite pouvoir l'utiliser dans le cadre d'autres séries de mesures, un moyen d'ajouter d'autres données dans la base était donc nécessaire.

2.2 Présentation de l'existant

Comme je l'ai dit plus haut ce stage était basé sur un projet déjà existant. Le projet a pour nom Heuliad (signifie série en breton), il s'agit d'un site web ainsi qu'une base de données associée destinée à accueillir, visualiser et analyser les différentes mesures scientifique effectué par l'observatoire de l'IUEM. La base de données était déjà existante et contenait des données scientifiques de campagnes de mesures, vous pouvez trouver en annexe un schéma de la base (à la page 36), dont je vais détailler les principaux objets. Il est nécessaire de différencier deux cas de mesures, les missions d'observation et les séries temporelles. Les missions d'observations sont le plus souvent sur des courtes durées (en semaines), il s'agit, en général, d'un bateau qui part naviguer dans une zone et va prendre des échantillons à plusieurs endroits. Alors que les séries temporelles sont effectuées sur de longues durées (à l'année), elles consistent en un placement d'une sonde à un endroit fixe qui mesure différents paramètres à un intervalle régulier. Il était donc nécessaire d'avoir un schéma permettant de regrouper ces différentes formes de séries de mesures qui restait clair dans les deux cas. Le schéma résultant

— Une série : Représente une série de mesures, que ce soit une missions d'observation ou une série temporelle. Elle est définie par un nom et une description.

— Une campagne : Sous-ensemble de mesures qui est pratique dans le cas d'une mission d'observation, car elle permet de différencier les différents jours ou passage du bateau. Pour ce qui est des séries temporelle il n'est pas très utile on ne crée qu'une unique campagne.
— Une station : Un lieu ou une sonde a été posée afin d'effectuer une mesure, définie principalement par : une lagitude, une latitude et une élévation ou la profondeur a laquelle a été immergée la sonde.
— Un paramètre : Un paramètre mesuré défini par un nom et son unité de mesure.
— Une mesure : Une mesure effectuée pour un paramètre donnée, défini par son paramètre, sa valeur, la date à laquelle elle a été effectuée et son code qualité.

La base contenait déjà les résultats d'une mission d'observation tandis que les données que je devais intégrer été celles d'une série temporelles, il s'agissait donc de l'opportunité de tester la modélisation de la base et savoir si des modifications seraient nécessaires. En plus de cette base de données existante, un début d'interface de visualisation était déjà présent. Elle permettait de naviguer parmi les différentes mesures et d'afficher quelques informations sur les différentes objets principaux listés plus haut. Enfin pour ce qui était de l'existant avant le début de mon stage il y avait les fichiers de données. Ils étaient réçus tout les ans et stockés sur un serveur de l'IUEM en attente de traitement. Comme dit précédemment différentes sondes ont été utilisés aux cours la série temporelle il y avait donc différents formats de fichier.

2.3 Langages utilisés

Dans cette partie je vais vous présenter les principaux langages utilisés durant mon stage. Ces présentations sont composées de deux parties, une définition du langage puis son utilité dans le contexte du stage.

Ruby & Ruby on Rails

Le principal langage utilisé au cours de ce stage aura été le Ruby. Il s'agit d'un langage fortement orienté objet se rapprochant du paradigme objet de Smalltalk :

— Toute donnée est un objet, y compris les types ;
— Toute fonction est une méthode ;
— Toute variable est une référence à un objet ;

C'est un langage très pratique dans le cadre d'un projet car il possède de nombreuses bibliothèque permettant d'effectuer de nombreuses tâches variées toujours avec le même langage (du web à la base de données par exemple). Ruby on Rails est un framework web libre écrit en Ruby, il est basé sur le motif d'architecture logicielle modèle-vue-contrôleur (MVC). Ce langage est celui utilisé pour coder le site d'Heuliad, il permet d'avoir un visuel des données assez rapidement et permet une gestion des différents classes de la base sous forme d'objets. De plus la communauté autour de ce langage est très active et de nombreuses gemmes (noms des extensions très facile à implementer dans un projet) permettent d'ajouter de nouvelles fonctionnalités à un site.

PostgreSQL

PostgreSQL est un système de gestion de base de données relationnelle et objet distibué sous licence BSD en tant que logiciel libre. Il a un comportement proche d'Oracle tout en restant libre et possède une communauté très active. PostgreSQL est largement reconnu pour son comportement stable, ainsi que pour ses possibilités de programmation étendues, directement dans le moteur de la base de données, via PL/pgSQL, pouvant être couplé à d'autres modules externes compilés dans d'autres langages. La base de données en place était gérée par ce système, toutes les interactions avec elle au cours du stage ont donc été effectuées par ce biais ainsi que par Ruby.

Javascript et CoffeeScript

JavaScript est un langage de programmation de scripts principalement employé dans les pages web interactives mais aussi du coté des serveurs. C'est un langage orienté objet à prototype, c'est à dire que les bases du langage et ses principales interfaces sont fournies par des objets qui ne sont pas des instances de classes, mais qui sont chacun équipés de constructeurs permettant de créer leurs propriétés, et notamment une propriété de prototypage qui permet de créer des objets héritiers personnalisés. En outre, les fonctions sont des objets de première classe. Le CoffeeScript quant à lui améliore la brièveté et la lisibilité du JavaScript, il fonctionne sur le même principe que Python en hiérarchisant les blocs de code selon l'indentation (comme l'HAML également qui est présenté plus bas).
Ces langages m'ont permis de modifier le comportement de différentes pages du site ainsi que d'en modifier le contenu, de façon dynamique.

HAML

Le Haml (HTML Abstraction Markup Language) est un langage de balisage léger pour templates. Il est utilisé pour générer des documents XHTML sans utiliser la syntaxe HTML de façon simpliste et minimaliste. Comme en Python l'indentation est importante, c'est elle qui définit la hiérarchie du document et permetl la fermeture automatique des balises et blocs de code. Ce langage simplifie énormément le code car il permet d'enlever de nombreux symboles HTML (<, >, id="", class=""…). L'indentation stricte rend également la gestion d'erreurs plus rapide,le document HTML final ne se génèrant pas s'il y a une erreur syntaxique. Ce langage étant de base utilisé dans les projets Rails et optimisant le développement web, j'ai jugé bon de l'utiliser pour les différentes pages du site.

Foundation

Foundation est un framework d'interface, ou framework front-end. Il propose un ensemble de composants d'interface cohérenet qui permet de simplifier le codage d'une interface web. Le projet Heuliad avait déjà été commencé à l'aide de ce framework j'ai donc continué dans cette lancée.

LaTeX

LaTeXest un langage et un système de composition de documents principalement utilisé pour la rédaction de textes scientifiques. La rédaction s'effectue en deux étapes identiques à la pro-

grammation informatique, l'écriture du texte puis la compilation de celui-ci qui genére la mise en page du document. Il permet la mise en forme de nombreux langages de programmation ainsi que d'équation mathématiques. Je l'ai personnellement utilisé pour la rédaction de mon rapport et de mes diapositives.

2.4 Logiciels utilisés

Lors de mon stage j'ai utilisé de nombreux logiciels et autres programmes, je vais donc vous présenter les principaux de la même façon que les langages précédemment (en deux parties, une définition du logiciels / programmes puis son utilité dans le contexte du stage).

SSH

Il s'agit d'un protocole de communication sécurisé entre ordinateurs. Il permet de se connecter à un ordinateur distant depuis son propre ordinateur. Durant mon stage ce protocole était constamment utilisé afin d'accéder aux ressources des serveurs de OVH. Ainsi toutes les instances de routages et les programmes nécessitant de gros calculs sur les fichiers OpenStreetMap était accessible et modifiables.

Git

Git est un logiciel de gestion de versions décentralisé et open source conçu pour manipuler de petits ou grands projets avec vitesse et efficacité. Comme la plupart des projets informatiques de nos jours, le projet était suivi par un logiciel de gestion de versions. L'utilisation d'un tel outil permet la conservation d'un historique de chaque modification des fichiers et offre la possibilité de revenir en arrière en cas de problèmes mais aussi le développement en parallèle de plusieurs utilisateurs ou encore de plusieurs fonctionnalités (système de branches).

Capistrano

Capistrano est un outil open source pour exécuter des scripts sur plusieurs serveurs POSIX. Son utilisation principale est le déploiement d'applications web. Il automatise le processus de création d'une nouvelle version d'une application disponible sur un ou plusieurs serveurs Web, y compris le soutien des tâches telles que la modification des bases de données. Capistrano est écrit dans le langage Ruby et est distribué à l'aide du canal de distribution RubyGems. Il est donc principalement utilisé pour le déploiement d'application web développée à l'aide du framework Ruby on Rails mais peut tout à fait l'etre dans le cas d'autre langages comme PHP par exemple. Ce logiciel était utilisé pour lancer en production l'application sur les serveurs de l'IUEM. A chaque nouvelle version, après une gestion des erreurs principales, une mise en production été effectuée par celui-ci.

Redmine

Redmine est une application web libre de gestion de projets complète en mode web, développée en Ruby on Rails. Il permet de regrouper de nombreuses taches utiles au suivi d'un projet, dont voici quelques une utilisées aux cours de mon stage :

— gestion des droits utilisateurs définis par des rôles,
— rapports de bogues (bugs), demandes d'évolutions,
— Wiki multi-projets,
— news et tickets accessibles par RSS / ATOM,
— notifications par courriel (mail),
— gestion de feuilles de route, GANTT, calendrier,
— saisie du temps passé (sans possibilité de modifier une saisie antérieure),
— intégration avec divers suivis de versions : SVN, CVS, Mercurial, Git, Bazaar et Darcs,
— support de plusieurs bases de données : MySQL, MariaDB, PostgreSQL, SQLite ou SQL Server.

Cet outil a été crucial au bon déroulement du stage, il permetait d'avoir un suivi des features à developper, des bugs à résoudre ainsi que de planifier les futures étapes du projet.

GNU Emacs

GNU Emacs est une des deux principales versions de l'éditeur de texte Emacs. Comme Ruby on Rails il possède une très large communauté qui développe de nombreuses extensions ce qui le rends très complet. Je l'ai donc utilisé pour le développement d'Heuliad mais aussi pour la gestion d'un carnet de bord grâce à l'extension Org mode (permet de maintenir des listes TODO, de planifier des projets, ...) et la rédaction de mon rapport.

3. Présentation du travail

3.1 Organisation

Dans cette partie je vais présenter l'organisation de mon travail, des différents outils utilisé pour gérer le projet et je présenterais un planning simplifié du déroulement du stage. Le stage a donc débuté le 03 Janvier 2017, après une première phase d'initiation à Ruby et Rails. En effet suite à mon entretien au sein de l'IUEM en décembre 2016 il m'a été conseillé de me former un minimum sur ses différents langages. J'ai donc effectué quelques tutoriels avant le début de celui ci.
La première semaine de stage a été principalement orientée sur la découverte du sujet ainsi que de l'existant et la mise en place des différentes méthodes de travail. Le projet était guidé par les principes de développement Agiles. A l'aide d'un site de gestion de version (basé sur les principes de Redmine) nous définissions avec Jonathan (mon maître de stage) la version suivante du site, les fonctionnalités à développer et les bugs à résoudre. Par exemple voici une capture d'une roadmap du développement :

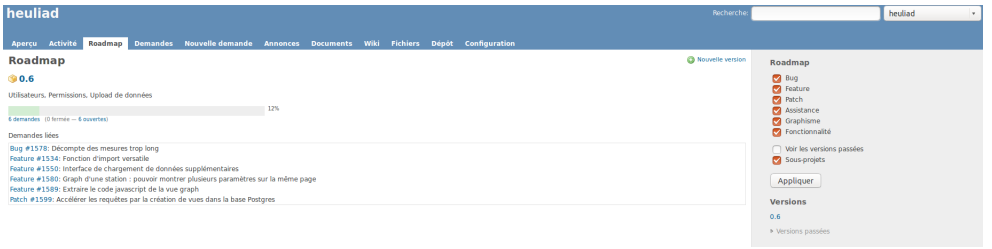


Figure 3.1 – Roadmap v0.6

De plus le FEIRI organise une réunion bi-hebdomadaire ayant pour but de résumer son activité passée et ses futurs projets. Ainsi tout le service se réunit et chaque pôle résume son activité sur les deux dernières semaines ainsi que ses perspectives sur les prochaines à venir. Ceci permettait d'avoir une idée des différentes activités effectuées par le service mais aussi de rendre compte des différentes interactions avec d'autres personnes de l'IUEM pouvant intéresser le service. De plus c'était une occasion de faire part des problèmes rencontrés et peut-être d'obtenir une solution, ou du moins de connaître l'existence de celui ci.

Période	Missions effectuées
01/01	Découverte contexte du stage
	Regroupement des fichiers
	Uniformisation des formats
16/01	Import en base
16/01	Amélioration des vues
	Recherche sur les différentes libraires de graphe
	Test de Dygraph, D3 et Highcharts
13/02	v 0.3
13/02	Graphe sur un ensemble limité
	Passage sur Highstock
27/02	Construction JSON par PostgreSQL
27/02	Mise en place de l'agrégation
	Gestion des codes de qualité
20/03	v 0.4
20/03	Limitation du nombre de requêtes
	Menu de navigation
10/04	v 0.5
10/04	Multigraphe
	Import interactif
22/05	v 0.6
22/05	Mise en place des indexes
12/06	Mise en place des vues

Table 3.1 – Agenda simplifié du stage

Enfin au cours de mon stage j'ai tenu un journal de bord qui m'a permis d'obtenir ce planning qui présente de manière simplifiée le déroulement de mon stage :

3.2 Fichier de données et imports

3.2.1 Format de fichiers

La première étape de mon stage a été de gérer les différents formats des fichiers contenant les données scientifiques. L'utilisation de différents capteurs au fur et à mesure de l'évolution de la technologie de capture, de plus les fichiers étant passer entre les mains de plusieurs scientifiques ont subit différentes transformations. J'ai donc été obligé de passer par une étape d'identification des différents formats récupérés, voici le résultat obtenu [1] :

1. Une version plus détaillée des formats est disponible en annexe page 36

Période	Intervalle de mesure	Nombre de paramètres
1997-2016	1 heure	3
1997-2005	30 minutes	4
2006-2010	30 minutes	4
2007-2008 et 2011-2012	30 minutes	3
2012	2 minutes	10
2014 à aujourd'hui	2 minutes	12

Table 3.2 – Description des différents formats

Et voici un exemple de contenu des fichiers :

```
DUMONT D'URVILLE HF LAT=66 39.7 S LONG=140 00.6 E STEP=120
(2(i2,1x),i4,3(1x,i2),10f10.3)
 day       hour       bot    twat  cwat  baro    par       oconc
13/01/2006 23:12:00 1462.70 -1.12 27.53 976.90 99999.000 99999.000
13/01/2006 23:14:00 1464.63 -1.12 27.53 976.90 99999.000 99999.000
13/01/2006 23:16:00 1464.17 -1.12 27.53 976.80 99999.000 99999.000
13/01/2006 23:18:00 1464.90 -1.08 27.53 976.90 99999.000 99999.000
13/01/2006 23:20:00 1460.30 -1.08 27.53 976.90 99999.000 99999.000
```

Figure 3.2 – Exemple de format de fichier

Une ligne de fichier contient donc une date et une heure de mesure ainsi que les différentes valeurs mesurées pour les paramètres. Suite à la découverte de ces informations, il a été convenu qu'une étape d'uniformisation des fichiers allait être nécessaire avant l'import en base de données.

3.2.2 Uniformisation

Une fois les formats définis je me suis donc occupé de les uniformiser en un unique format adapté à l'import en base. J'ai également questionné les différents responsables scientifique si le format utilisé actuellement était susceptible de changer encore. Il m'a été confirmé que ce format devrait rester celui de référence pour un certain moment. Je suis donc partis sur le développement de méthodes d'uniformisation des formats plus ancien au format actuel. J'ai ainsi pu m'essayer au langage Ruby en créant des scripts qui lisent les fichiers, récupèrent les différents champs dignes d'intérêt et les réorganisent dans un fichier de sortie. Le Ruby permet assez facilement ce genre de tâche, il contient différentes fonctions et classes permettant de gérer les fichiers CSV et les différents formats horaire. J'ai également crée une classe Mesure représentant une ligne d'un fichier, elle est donc composée d'une date, d'une heure et des différentes valeurs mesurées pour chaque paramètre. De plus j'ai écrit une fonction de sérialisation de la classe permettant de rendre la mesure au format actuel en chaîne de caractère. Le fonctionnement du script d'uniformisation était le suivant :

- Ouverture du fichier
- Boucle sur les lignes
- Création d'un objet Mesure à l'aide des champs de la ligne actuelle
- Stockage de la mesure dans un tableau de mesures

— Une fois atteint la fin du fichier, le tableau de mesures est écrit dans un fichier de sortie
Enfin une fois les différents fichier uniformisés je les ai fusionnés dans un unique fichier rangé par ordre chronologique.

3.2.3 Volumétrie de la donnée

C'est à ce moment que la principale problématique du stage s'est révélée. En effet une fois les données regroupées il a été possible d'avoir une idée concrète de la taille qu'elles représentaient. Les mesures de 1997 à 2016 à raison d'une mesure par heure n'était pas problématique mais celles de 2006 à 2016 à raison d'une mesure toutes les deux minutes elles posaient problème. En effet nous avions environ trois millions de lignes de mesures, soit plus de trois cent méga-octets. Un tel nombre de données allait nécessiter énormément d'optimisations que ce soit à la mise en base ou à son questionnement.

3.2.4 Base de données

Lorsque les fichiers ont été unifiés je me suis intéressé à la façon dont je pourrais les importer dans la base de données de L'IUEM. Il s'agit d'une base postgres existante ayant déjà été utilisée dans le cadre du projet heuliad pour des séries de mesures effectuées sous forme de campagne.Il était donc question d'ajouter des données d'une série temporelle dans ce modèle de base de données, quitte à le modifier, le but étant de regrouper toutes les futures séries de données quelle que soient leurs formes. Le schéma de la base de données est disponible en annexe page 36.

3.2.5 Import

L'import des données au sein d'un projet Rails se fait par le biais de Rake. Il s'agit d'un moteur de production semblable à make, il permet de définir des tâches courantes pouvant être effectuée dans le cadre du projet. Il était donc important d'effectuer une tâche d'import aussi générique que possible permettant d'être réutilisée au cours du projet heuliad. De plus il fallait optimiser le code d'import à cause du trop grand nombre de données, un parcours trivial du fichier suivi d'un import ligne par ligne s'effectuait en plusieurs jours ce qui n'était pas envisageable.

Une des premières optimisations effectuée a été d'instancier des objets Mesure, classe présentée plus haut, ce qui a permis d'alléger un petit peu l'import mais pas suffisamment. Le problème venait principalement de l'allocation mémoire nécessaire pour un tableau de mesures une fois arrivé en fin de fichier. J'ai donc simplifié l'import en l'effectuant en plusieurs étapes arrivé à un certain nombre d'éléments dans le tableau (de manière arbitraire : 10 000 éléments), mais ce n'était pas encore suffisant. La solution est venue de la découverte de l'existence de la gem "active-record import", elle permet de simplifier les requêtes à partir du modèle des données. Ainsi une simple ligne comme :

```
result = Measure.import columns, measureArray, validate : false
```

Measure étant la classe du modèle correspondant à la table Mesures en base, la gem permet d'utiliser une méthode "import" sur cette classe, méthode à laquelle on passe deux paramètres. Un tableau de données, dans notre cas les mesures, une chaîne de caractère contenant les différentes colonnes correspondant aux champs du tableau. La méthode permet l'utilisation de multiples options dont validate qui défini si l'on effectue une phase de validation avant les imports afin de

respecter les différentes contraintes d'unicités et autres de la base. Dans notre cas les données ayant déjà été traitées et afin d'optimiser le temps de calcul il a été décidée de passer cette étape (d'où le validate : false, l'affectation se faisant par le symbole ' :' en Ruby). Une fois l'entête récupérée dans le tableau "columns", les mesures rentrées sous forme de chaîne de caractère (plus besoin d'utiliser la classe Ruby et d'instancier d'objets) dans "measureArray" cette méthode importe un tableau de 10 000 mesures en 36 millisecondes, soit environ 3 heures pour les 3 millions de mesures. A tout cela s'ajoutait un problème de représentation des données. Dans le cas d'une mesure inexistante (pour diverses raisons) il était fréquent de trouver la valeur '99 999.00' pour une mesure, mais avoir de telles valeurs en base n'était pas spécialement pratique, surtout une fois l'étape de visualisation arrivée. Suite à un entretien avec les responsables scientifiques il a été convenu que l'utilisation de la "valeur" nil serait plus appropriée quitte à définir, à l'import, ses mesures par un code qualité : "Mesure absente".

3.3 Interface web

Cette partie est dédiée à tous les développements effectuées dans le cadre de mon stage afin d'améliorer l'interface du site heuliad ainsi qu'aux nouvelles fonctionnalités de visualisation et d'import que j'ai ajouté.

3.3.1 Documentation de l'installation du projet

La première étape a été en toute logique d'installer le projet sur ma machine. Mais n'ayant jamais été utilisée par d'autres personnes que mon maître de stage il ne disposait pas de documentation. J'ai ainsi eu pour première tâche d'ajouter au wiki du Feiri une documentation d'installation détaillant les différentes étapes devant être effectuées, ceci au fur et à mesure de ma propre installation du projet.

3.3.2 Visualisation des données

Modèle et informations

La version initiale du site heuliad été utilisée dans le cadre de séries d'observation sous forme de campagne. Il y avait donc une campagne, appartenant à une série, qui contenait de nombreuses stations correspondant à chaque point géographique pour lequel le bateau de la campagne s'était arrêté. Ainsi dans la vue d'une station on trouvait une liste des mesures effectuées à ce point, ce qui était pour le moment de l'ordre d'une centaine de valeurs. Mais avec l'ajout des données de Dumont d'Urville affiché la liste des mesures n'était plus envisageable. Ma première mission a donc été de modifier cette vue. J'ai opté pour un résumé des informations importantes relatives à la station, comme par exemple :

- Le nombre de mesures effectuées sur cette station.
- Les paramètres qui ont été mesurées
- La période couverte par les mesures.

Afficher ces informations a permis de limiter le nombre de requêtes ainsi que le temps de réponse de la page. Afin de récupérer les informations suffisantes, j'ai utilisé des fonctions PostgreSQL, qui sont également disponible en Ruby grâce à ActiveRecord, comme Count, First et Last.

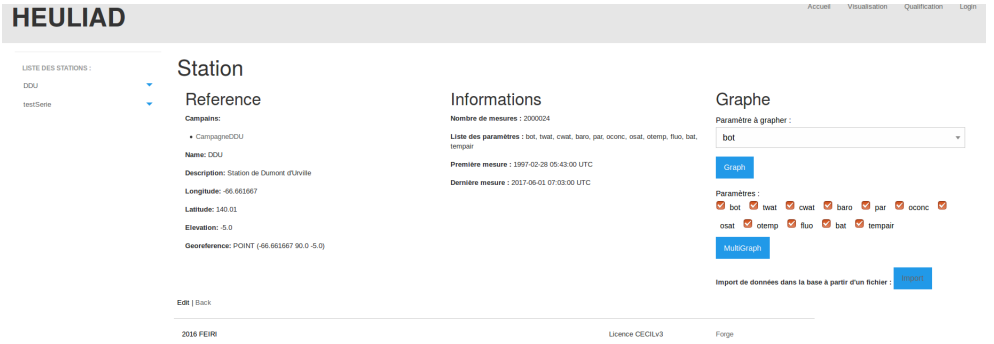


Figure 3.3 – Vue d'une station

Graphes des données

Différentes librairies

Les scientifiques n'ayant pour le moment que récupéré les fichiers contenant les données scientifiques n'avaient qu'une idée très vague de celles-ci, il était donc nécessaire de développer un outil permettant d'évaluer de façon rapide et efficace la qualité des mesures et l'évolution des paramètres mesurés. Pour ceci l'outil le plus adapté et fréquent est le graphe j'ai donc effectué une première étape de recherches sur les différentes bibliothèques de graphes existantes basées sur différents critères liés au cas de mon stage. La bibliothèque devait être la plus efficace possible avec d'importants ensemble de données, la plus paramétrable, facilement interactive afin de pouvoir interagir avec les données et si possible facilement adaptable à un projet Rails. Mes recherches m'ont menées vers trois bibliothèques de graphes différentes :

— Dygraph : C'est une bibliothèque Javascript open-source qui permet assez facilement à l'aide d'un bloc de code que l'on peut paramétrer et dans lequel on rentre les données d'obtenir un graphe.

— D3.js : D3 est quant à elle une bibliothèque faite pour la formulation de document basé sur les données. En effet elle ne se limite pas aux graphes mais permet de construire tout type de schémas comme des tableaux, des diagrammes, ... Elle est basée sur HTML, CSS et SVG et permet une approche orientée donnée des manipulations DOM (Document Object Model) d'HTML et XML.

— Highcharts / Highstock : Il s'agit d'une bibliothèque propriétaire, développé par Highsoft. Elle est disponible gratuitement pour une utilisation non commerciale. Elle dispose de multiples types de graphes, d'options diverses et différents paramétrages.

Dans un premier temps j'ai testé les différentes bibliothèques avec des données simples permettant d'afficher un graphe basique. Ceci permettait d'avoir une idée du fonctionnement de la bibliothèque mais également de sa simplicité d'intégration dans un projet Rails. Différentes gems existaient justement afin de simplifier l'intégration de celles-ci à un projet, mais la plupart n'étant pas tenues à jour, je me suis contenté d'intégrer le code des bibliothèques dans le dossier adapté. Une fois les différentes libraires testées j'ai pu essayer d'utiliser un sous-ensemble des données de

mesures et c'est à ce moment que D3 à montrer des problèmes. En effet la bibliothèque était très puissante mais demandé un investissement assez important avant d'avoir un résultat intéressant. D3 permettant de définir tout type de schémas basé sur des données, elle nécessite de définir tout les outils et bases que l'on souhaite utiliser, ainsi il fallait définir un point, les axes (abscisses et ordonnées), ... Cette bibliothèque aurait été très intéressante à explorer mais le graphe n'étant pas le cœur du stage il n'était pas possible de s'attarder autant dessus, la visualisation n'étant qu'une partie de la mission, j'ai donc opté sur une solution moins libre mais plus avancée. Le choix de libraire s'est effectué sur le point névralgique du stage, l'importance de l'ensemble de données a grapher. C'est au moment ou j'ai testé les deux autres bibliothèques sur quasiment un million de mesures que j'ai pu apercevoir les limites d'une d'entre elles. Dygraph était facilement implémentable mais arrivé avec un ensemble important de mesure les temps de réponses devenaient problématique, il fallait plusieurs minutes avant d'avoir une quelconque information de rétroaction de la page. Highcharts n'était pas non plus bien meilleure, mais une deuxième version de la libraire existe et pallie ce problème.

Highstock

Dans le cas de graphe de millions de points Highsoft a développé une version de sa libraire nommée Highstock. Elle permet l'affichage de différents type de graphes comme Highcharts et possèdent également de multiples fonctionnalités, mais la plus intéressante est le data-grouping. Cela permet de limiter le nombre de points à grapher lorsque un nombre trop important est demandé, en effet il n'est pas nécessaire d'essayer d'afficher 100 000 points sur un graphe d'environs 800 pixels cela ne se verra même pas.

L'utilisation de cette bibliothèque a permis d'avoir les premières visualisations des données même si elles n'étaient pas encore acceptable d'un point de vue temps de réponse. Le fonctionnement initial était le suivant :

1. On choisit le paramètre que l'on souhaite grapher depuis la vue de la station.
2. Le modèle effectue une requête à la base afin de récupérer la liste des mesures liées à cette station pour ce paramètre.
3. Le JsonBuilder renvoie le résultat de la requête au format json de l'url.
4. Le contrôleur récupère le json et le passe à la vue par le biais d'une variable
5. La vue construit le graphe en passant le json dans le champ "data" de la déclaration en Javascript du graphe.

Voici la déclaration Javascript d'un graphe avec Highstock (ici la température) :

```
$.getJSON('#{url_for(controller: :measures, action: :highstock, format:
:json, station_id: @station.id, parameter_id: @param,
quality_id: @qualite)}&callback=?', function (data) {
Highcharts.stockChart('container',{
title : {
text : "Graphe de la température",
  x : -20 //center
},
...
navigator : {
  adaptToUpdatedData : false,
  series : {
```

```
      data : data
    }
  },
  xAxis : {
    type : 'datetime', title : 'Time',
    events : {
      afterSetExtremes : afterSetExtremes
    },
    minRange : 1800 * 1000 // une demie heure
  },
  yAxis : {
    title : {
      text : "degré(°C)"
    },
    plotLines : [{
      value : 0,
      width : 1,
      color : '#808080'
    }]
  },
  ...
  series : [{
    name : "Température",
    data : data,
    lineWidth : 0,
    dataGrouping : {
      enabled : false
    },
    marker : {
      enabled : true,
      radius : 2
    }
  }]})
  });
```

Le code est assez simple, la fonction jQuery getJSON est lancé avec pour paramètre une fonction définissant le contenu d'un JSON, ici la fonction highstock du contrôleur measures, ainsi que les paramètres de la dite fonction (station_id, parameter_id, quality_id). Le résultat de la fonction passée en paramètre est retourné dans la variable data. Celle ci est passée, en cas de succès, à la fonction de rappel (callback) qui crée le graphe. Dans la fonction de rappel on peut voir l'appel à la libraire Highstock "Highschart.stockchart()" et les différents paramètres nécessaires à la création du graphe sous forme de bloc. Par exemple le paramètre title qui définit le titre du graphe, les paramètres yAxis et xAxis auxquels on donne leurs nom ainsi que leurs unités, etc... Les paramètres les plus important étant navigator et series. Le premier est la frise temporelle que l'on trouve en bas du graphe et qui permet de naviguer, comme son nom l'indique, parmi les données. Le second correspond aux courbes qui seront affichées, c'est dans celui-ci que l'on retrouve la variable data créée ultérieurement.

Cette version a été fonctionnelle jusqu'à la mise en production. Sur ma version de test elle devait grapher environ 300 000 mesures pour un paramètre mais une fois la base de donnée de production utilisée il était question de 2 millions de mesures. La construction du JSON était bien trop lente, de même pour le re-dessin du graphe lorsque l'on souhaitait zoomer, d'où les optimisations qui sont présentées dans le chapitre suivant. En plus du graphe des mesures d'une station pour un paramètre donné, d'autres fonctionnalités était nécessaire. À commencer par la prise en compte du code de qualité, en effet toute mesure passe à un moment donné par une phase de qualification. Celle ci consiste en la définition d'un code, ou coefficient, de qualité associé à la

mesure qui permet d'avoir un indice sur l'état de la donnée, vous pouvez trouver un annexe la liste des différents code de qualité possible et leur signification (page 37. À l'import j'ai convenu avec les scientifiques de définir le code "Non défini" par défaut, j'ai ensuite mis en place le choix du code qualité a afficher en plus du paramètre voulu. Ce qui au final ne se résume qu'à l'ajout d'une condition sur la requête à la base, le code qualité étant un attribut d'une mesure. Il a également été nécessaire d'effectuer de la gestion d'erreurs afin de solidifier le comportement de la page, comme redirigé vers la page de la station quand : un paramètre est manquant, un des identifiants passés (station, paramètre ou code qualité) ne correspond à aucun identifiant en base, un identifiant de paramètres qui n'appartient pas à la station, etc ... De plus il y a eu de multiples optimisations a effectués afin d'avoir un graphe fonctionnelle mais celles-ci sont développées toutes ensembles dans la partie suivante.

Multigraph

Une autre fonctionnalité qui intéressait les scientifiques était la possibilité de comparer l'évolution de deux paramètres sur une période de temps donnée. Le plus simple était d'afficher deux paramètres sur le même graphe mais cela posait un problème de grandeur. En effet si l'on affiche la pression en même temps que la température le graphe devient illisible, la température avoisinant les 0 degrés et la pression étant de l'ordre des milliers de millibars, la courbe de température se résumait à une droite. De plus il devait être possible d'afficher plus de deux paramètres à la fois. Il a donc été décidé de dessiner les graphes séparément mais cela a soulevé un autre problème. Chaque graphe ayant sa propre frise, permettant de naviguer parmi les données, il était nécessaire de pouvoir synchroniser les différents graphes, donc frises, afin de comparer sur une période équivalente. Un tel comportement était envisageable grâce à l'utilisation d'Ajax, pour modifier le comportement et le contenu de code Javascript au gré des actions effectuées sur la page. J'ai mis en place un bouton de synchronisation pour chaque graphe qui permet d'appliquer la période observée sur le graphe courant à tous ceux de la page. Cette fonctionnalité est, au final, à moitié fonctionnelle, en effet la synchronisation ne s'effectue uniquement sur les graphes se trouvant en dessous du graphe courant. Il a été décidé de mettre en pause celle ci pour se concentrer sur la suivante qui était plus prioritaire.

Import de données interactif

Afin de rendre le projet heuliad utilisable et permettre d'avoir des retours d'utilisateurs le plus rapidement possible il était nécessaire de permettre à ceux-ci d'ajouter eux même leur mesures à la base de données. Pour cela j'ai développé une interface d'import de données interactif, dans le but de simplifier au maximum la tâche d'import sans avoir à s'occuper de l'aspect base de donnée. L'import devait s'effectuer en deux étapes, une première de chargement du fichier vers le serveur heuliad et une seconde de vérification du fichier et de configuration de l'import. Afin de faciliter la tâche il a été décidé de convenir d'un format de fichier a télécharger, j'ai donc ajouté à la première page une liste de conditions que le fichier devait remplir pour que l'import ai lieu.

Figure 3.4 – Interface d'import d'un fichier

La phase de chargement était plutôt facile, Rails contient parmi ses Helpers la fonction `file_field()`, équivalent de `input type= "file"` en HTML, qui crée dans la vue un champ de saisie pour un fichier. Une fois le fichier choisi il est télécharger et sauvegarder temporairement sur le serveur afin de simplifier les différents traitement à effectuer dessus. Plusieurs tests sont alors effectués afin de savoir si le fichier rempli les conditions (l'extension étant testée dés l'annonce du nom du fichier à téléchargé), si l'une d'entre elles n'est pas valide l'utilisateur est redirigé vers la page précédente et informé par un message Javascript personnalisé suivant l'erreur. Dans le cas ou toutes les conditions sont remplies la page de vérification du fichier et de configuration de l'import est affichée :

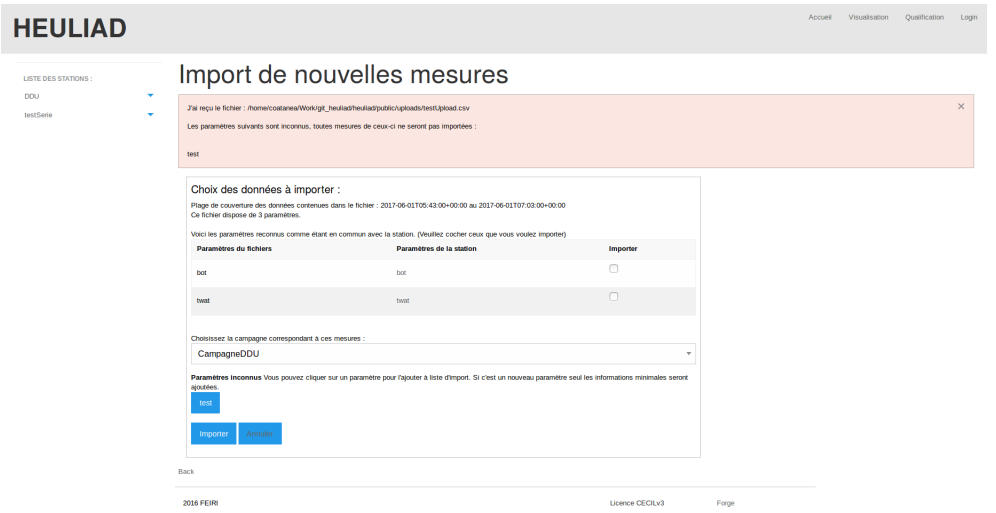


Figure 3.5 – Interface de configuration de l'import

On peut y trouver :

— Une alerte JavaScript confirmant le nom du fichier téléchargé et annonçant si des para-

mètres de l'en-tête non pas été reconnus.
— Des informations sur le fichier (première et dernière mesure, nombre de paramètres)
— Un tableau des paramètres reconnus associés à ceux existant en base et un case à cocher
— Une liste des campagnes liées à la station actuel

Ainsi une fois le fichier téléchargé plusieurs traitements sont effectués afin d'afficher cette page. Une comparaison de l'en-tête du fichier avec la liste des paramètres appartenant à la station permet de connaître lesquels existent déjà et ceux qui ne sont pas encore dans la base. Les paramètres non reconnus sont affichés dans une fenêtre d'informations comme n'allant pas être importés car de trop nombreux cas était à prévoir (mauvaise orthographe, ...).

Une lecture de l'en-tête, de la seconde et dernière ligne permet de retirer quelques informations utiles à l'utilisateur (période parcourue, nombre de paramètres, ...). Le tableau des paramètres reconnus avec les case à cocher est destiné à la configuration de l'import, en effet il n'est pas toujours souhaité d'importer toutes les mesures d'un fichier. Si l'utilisateur souhaite importer seulement un paramètre spécifique il coche la ligne de celui ci, par exemple si l'on veut ajouter des mesures sur une période ultérieurement. Ensuite, une mesure ayant un lien vers une campagne dans le modèle, l'utilisateur doit choisir la campagne liée aux mesures qu'il souhaite importer ce qui se fait par le choix parmi une liste déroulante construite en parcourant les campagnes liées à la station.

Une fois toutes ces étapes de configuration passées il est possible d'effectuer quasi semblable à celui fait à l'origine. L'unique différence est la nécessité d'appliquer une validation, dans un soucis de temps elle avait été passée lors de l'import original mais ici sachant qu'il y a peu de chances d'avoir un import de millions de mesures elle est possible. L'ajout de cette phase implique la possibilité que des mesures ne soient pas importés car elles pourraient violer les contraintes de validité crées dans le modèle de base de données. Pour garder une trace de ces mesures non importés j'ai utilisé le résultat du `Measures.import()` présenté ultérieurement, celui ci garde de coté toutes les requêtes d'importation qui ont échouées (`result.failed_instances()`). Grâce à ces résultats j'ai pu créer un fichier log listant toutes les lignes du fichier ayant posé problème.

3.4 Optimisations effectuées

Dans cette partie je vais développer la thématique de mon stage qui est revenue lors de chaque tentative de développement de nouvelles fonctionnalités, l'optimisation du à la gestion d'un important ensemble de données.

3.4.1 Utilisation de tableaux à la place de modélisation d'objets

Dans les premières versions du graphe la récupération des mesures nécessaires au dessin du graphe été effectué par ActiveRecord et renvoyé sous la forme d'un JSON (la construction de celui-ci est également passée par différentes étapes mais n'a pas influencé de façon notable les résultats). L'utilisation d'ActiveRecord s'est révélée être problématique une fois le projet mis en production pour la première fois. J'ai découvert suite à des recherches sur la construction d'un JSON dans un projet Rails, le travail d'un certain Dan McClain [2] à propos de l'intérêt d'éviter

2. L'article de son blog(https ://frama.link/xsjxHMux)

ActiveRecord pour cette tâche et de lui préférer PostgreSQL. Il y défini le comportement des deux méthodes et effectue des comparatifs des résultats :

Fonctionnement d'ActiveRecord :

1. Rails reçoit la requête d'un JSON par le navigateur
2. Rails mets en œuvre la logique applicative et construit une requête par le biais d'ActiveRecord
3. ActiveRecord sérialise la requête et l'envoie à PostgreSQL
4. PostgreSQL effectue la requête et sérialise le résultat dans son protocole de format.
5. ActiveRecord dé-sérialise le résultat en un ensemble d'objets Lignes (de base de données)
6. ActiveRecord convertit ces Lignes en instance d'objets du modèle (en Mesures dans notre cas)
7. Rails convertit ces objets en un JSON.
8. Rails renvoie le JSON au navigateur

Fonctionnement de PostgreSQL :

1. Rails reçoit la requête d'un JSON par le navigateur
2. Rails mets en œuvre la logique applicative et construit une requête par le biais d'ActiveRecord
3. ActiveRecord sérialise la requête et l'envoie à PostgreSQL
4. PostgreSQL effectue la requête, sérialise le résultat en JSON puis sérialise celui-ci dans son protocole de format.
5. ActiveRecord dé-sérialise le protocole de format en un JSON.
6. Rails renvoie le JSON au navigateur

La plus longue partie d'ActiveRecord est effectué dans les étapes 6 et 7. Rails doit dé-sérialiser un format, le stocker en mémoire uniquement pour le sérialiser encore une fois dans un format différent. PostgreSQL étant capable de rendre le résultat de la requête en JSON, il est tout aussi intéressant de l'envoyer directement sous ce format à ActiveRecord qui n'aura plus qu'à le dé-sérialisé, au lieu de passer par des objets qu'il devra convertir et sérialiser. Ainsi PostgreSQL effectue deux étapes deux moins, qui sont le cœur du problème, ce qui permet de limiter le nombre d'objets Ruby crées et donc l'espace mémoire utilisé par ceux-ci ainsi que par la conversion en JSON. Afin de tester l'efficacité de PostgreSQL j'ai mis en place un fichier de test Ruby :

```
startdate = "1997-02-28 05:43:00"
enddate = "2016-01-12 11:02:00"
parameter = 169
station = 711
qualite = 2

Benchmark.bm do |x|
  x.report("ActiveRecords Relation : ") { val = Measure.where(station : station )
  .where(parameter : parameter ).where(quality : qualite)
  .where(date : startdate..enddate).limit(1000000).to_a}
  x.report("Array of values [1/3]:  ") { Measure.datatable(startdate, enddate,
  parameter, station, qualite)}
  x.report("Array of values [2/3]:  ") { Measure.datatable(startdate, enddate,
  parameter, station, qualite)}
  x.report("Array of values [3/3]:  ") { Measure.datatable(startdate, enddate,
  parameter, station, qualite)}
end
```

A l'aide du module Ruby Benchmark il est possible de tester le temps d'exécution d'un code

limit	type	real	request
50 000	ActiveRecord	2.124812	875.2ms
Facteur : 1.58	PostgreSQL	1.341667	748.7ms
100 000	ActiveRecord	3.879179	1212.2ms
Facteur : 1.32	PostgreSQL	2.942919	1219.2ms
500 000	ActiveRecord	24.273150	8098.5ms
Facteur : 1.98	PostgreSQL	12.278865	4930.4ms
1 000 000	ActiveRecord	52.887709	12362.1ms
Facteur : 1.99	PostgreSQL	26.573625	10701.6ms

Table 3.3 – Résultats optimisations PostgreSQL

Ruby. Ce code effectue les deux types de requêtes, celle à l'aide d'ActiveRecord et celle avec PostgreSQL. J'ai également ajouté plusieurs instances de la deuxième versions afin de vérifier si PostgreSQL utilisait un cache permettant de retourner plus rapidement le résultat ce qui pouvait être pratique dans le cadre du graphe de données. De plus il a fallut mettre en place une limite au nombre de résultats pouvant être retournés par l'algorithme, le fait que ces tests étaient effectués sur la base de production impliquait la possibilité de retourner environs 2 millions de mesures ce qui représentait une trop grande charge de mémoire pour l'ordinateur que j'utilisais. Les résultats de ce test sont disponibles dans leur version complète en annexe (à la page 37 mais voici une version simplifiée contenant les conclusions essentielles.

Avec ces résultats il est possible de comparer le temps d'exécution (colonne real) et celui de la requête (colonne request) afin définir le facteur d'amélioration que possède PostgreSQL en comparaison à ActiveRecord. Ici il s'élève à 2, mais sur les différents tests effectués il peut atteindre 6 (soit 6 fois plus rapide). Il est également possible de distinguer une tendance avec ces quatre résultats, qui se confirme avec les autres, il semblerait que plus le nombre de résultats à renvoyer est important plus PostgreSQL est efficace.

3.4.2 Agrégation

Le principal soucis de l'outil de graphe a été le nombre de données récupérées par les requêtes sur les mesures. Ceci a donné naissance à la problématique suivante : "Quel intérêt y a-t-il à récupérer plusieurs centaines de milliers de points afin de grapher une courbe d'environ 800 pixels", en effet la vue n'atteindra jamais une taille de plusieurs milliers de pixels. Il était donc nécessaire de limiter le nombre de points récupérés quitte à perdre un peu de précision, et il se trouve qu'Highstock possède déjà une fonctionnalité similaire mais qui n'était pas suffisante dans notre cas. La bibliothèque même à l'aide de ses fonctions de data-grouping ne permettait pas de contrebalancer la lourdeur de la requête à la base et le formatage des données. La solution a été découverte en même temps que la fonction `date_trunc` qui permet de garder uniquement une partie d'une date comme par exemple le jour, la semaine ou le mois. En soit elle n'a pas énormément d'intérêt mais une fois utilisée avec une clause `GROUP BY` et la fonction de moyenne `avg()` son potentiel se découvre. En effet si l'on applique un tronquage par jour sur un groupe de mesures on obtient (dans le cas d'une mesure toutes les 2 minutes) 720 mesures pour la même date et si l'on ajoute à cela une moyenne des valeurs on obtient une mesure pour un jour. Nous avons diviser par 720 le nombre de résultats obtenu pour une requête. C'est avec cette logique

Niveau d'agrégation	Intervalle de temps	Nb mesures
Une mesure par jour	> 6 mois	2464
Une mesure par heure	6 mois < > 1 semaine	57639
Données brutes	< 1 semaine	500016

Table 3.4 – Différents niveaux d'agrégations

		Agrégation par heure (113488 mesures)		Agrégation par jour (4870 mesures)	
limit	type	real	request	real	request
50 000	ActiveRecord	2.168300	757.3ms	1.649282	525.9ms
	PostgreSQL	7.996041	7804.9ms	7.599660	7585.0ms
100 000	ActiveRecord	4.003584	1075.4ms	4.908916	2209.4ms
	PostgreSQL	8.233604	7811.4ms	9.013775	8999.3ms
500 000	ActiveRecord	27.515003	8576.7ms	21.963939	6061.9ms
	PostgreSQL	10.590090	9927.2ms	7.913364	7897.5ms
1 000 000	ActiveRecord	59.503445	10927.1ms	49.235981	9854.7ms
	PostgreSQL	9.227083	8844.3ms	7.874495	7858.4ms

Table 3.5 – Comparaison d'ActiveRecord et de PostgreSQL avec agrégation

que j'ai essayé de définir des intervalles les plus optimisés pour obtenir un minimum de résultats, ce qui a donné : Certains chiffres restent encore très important car il faut prendre en compte que la fonction `date_trunc()` ne permet pas de tronquer selon énormément de paramètres, les plus exploitables étant heure, jour, semaine et mois. Mais le graphe de plusieurs centaine de milliers de points n'est pas un problème pour Highstock et reste dans des temps tout à fait raisonnable. Une fois les intervalles définis il a été possible de modifier la construction de la requête des mesures à envoyer à Highstock et une requête basique comme celle la :

```
SELECT date, value FROM measures WHERE(parameter_id = 1 AND station_id= 1 )
AND quality_id = 2 GROUP BY date ORDER BY date ;
```

est devenue ceci :

```
SELECT extract(epoch from date_trunc('day',date))*1000 as ts,avg(value)
FROM measures WHERE(parameter_id = 1 AND station_id= 1 )
AND quality_id = 2 GROUP BY ts ORDER BY ts ;
```

L'agrégation étant une limitation du nombre de résultats retournés à l'interface le gain de temps a été compliqué a évaluer. L'utilisation du tronquage des dates et du calcul de la moyenne ajoute des opérations a effectués et laisse imaginer que le résultat va être plus long à récupérer. Pourtant, à ma grande surprise, l'exécution immédiate de requêtes PostgreSQL en place d'ActiveRecord une fois atteint un certain volume de données reste plus rapide même avec le calcul de l'agrégation.

A l'aide de ce tableau on peut observer que ActiveRecord est plus rapide que PostgreSQL et son agrégation tant que le nombre de mesures a traitées ne dépasse pas 500 000. Arriver au seuil du million on peut voir que PostgreSQL est passé devant même si il doit effectuer tout les calculs liés à l'agrégation des mesures.

Le plus grand gain de temps, lié à cette optimisation, a tout de même été remarqué au niveau

du graphe des données. Highstock recevant quelques milliers de mesures au lieu de plusieurs centaines de milliers, voire des millions, affiche le graphe beaucoup plus rapidement.

3.4.3 Limitation du nombre de requêtes

Une fois l'agrégation mise en place une autre idée d'optimisation m'est venue à l'esprit. Ayant développé, pour l'agrégation, une fonction permettant de savoir quel niveau d'agrégation correspondait à deux dates, par exemple 01/01/2017 => 01/07/2017 = 'day', je pouvais savoir à quel moment exact le niveau allait changé. De plus le comportement du graphe faisait qu'à chaque fois que l'on zoomait ou dé-zoomait à l'aide de la frise toutes les mesures étaient rechargées, ce qui n'était pas nécessaire tant que la période à grapher n'était pas plus importante que la précédente ou ne représentait pas un nouveau niveau d'agrégation. J'ai donc créé une fonction permettant de définir à partir de 4 dates, l'intervalle précédent et le nouvel intervalle allant être affiché, si il était nécessaire d'effectuer une requête afin d'obtenir de nouvelles mesures ou si l'ensemble à grapher n'était pas un sous-ensemble de l'actuel.
Cette optimisation a été difficile à mesurer de manière quantitative car elle annulait toute nécessité de rechargement du graphe, le rendant obligatoire uniquement lors d'important zoom ou dé-zoom ou lorsque l'utilisateur passe d'une extrémité à l'autre de la frise temporelle. Mais je pense qu'au final cela a été la plus marquante concernant la fluidité de navigation sur le graphe. Une solution de test imaginée était de faire tester l'interface de base à des scientifiques et d'étudier leurs comportement, lister le nombre de zooms effectués, quelle taille d'intervalles était le plus demandé, etc ... Ceci aurait ensuite permis de peut-être mieux définir les intervalles d'agrégation et savoir à quels moments il était intéressant de recharger ou non les mesures.

3.4.4 Améliorations en base de données

Deux optimisations ont été possibles afin d'améliorer les performances de la base de données : les indexes et les vues. J'ai donc premièrement mis en place des indexes afin de simplifier l'accès aux mesures. Les indexes sont en quelques sortes des filtres pouvant être appliqués sur la base, il fonctionne sur le principe des algorithmes de tri (l'index par défaut utilisant un B-tree). Si l'on souhaite récupérer de manière fréquente les mesures ayant une valeur supérieur à 1000, un exemple d'indexe serait :

```
CREATE INDEX measures_bigger_than_1000 ON Measures WHERE value > 1000;
```

Une fois les indexes mis en place ils sont mis à jour pour toutes insertions ou suppressions en base, ils sont également utilisés de façon automatique par PostgreSQL dés lors qu'ils peuvent améliorer le temps de réponse d'une requête. La plupart des requêtes du projet heuliad sont effectuées durant le graphe des mesures, pour simplifier celles ci j'ai mis en place des indexes permettant de les accélérer. Pour chaque requête on cherche les mesures :

- Appartenant à une station particulière.
- Pour un paramètre donné.
- De la plus ancienne à la plus récente.

J'ai donc mis en place les deux indexes suivants :

```
CREATE INDEX measure_station_id ON Measures (station_id);
CREATE INDEX measure_parameter_id ON Measures (parameter_id);
```

Ceux-ci créent des indexes B-tree sur les colonnes station_id et parameter_id dans la table des mesures. Et l'index :

```
CREATE INDEX date_order_id ON Measures (date ASC);
```

Celui-ci permet d'avoir un accès plus rapide aux mesures ordonnées de façon croissante suivant les dates. Les indexes sont intéressants sur des requêtes fréquentes mais plus particulièrement lors de requêtes sur des petits ensembles perdus parmi de nombreuses données. Ainsi les différents tests effectués avec des requêtes avant et après la mise en place des indexes n'offrent pas de résultats très concluants, mais une fois les indexes mis en place sur la base de données de production j'ai observé une amélioration des temps de réponses des petites stations existants avant l'import de celle de Dumont d'Urville. Je n'ai malheureusement pas effectué de fichier de log à ce moment et ne peux donc pas donner un facteur d'amélioration de cette optimisation.

La seconde optimisation dédiée à la base de données était l'utilisation de vues. Les vues sont des requêtes SELECT gardées en mémoire qui permettent d'effectuer des opérations sur leurs résultats. Elles peuvent être utilisées pour donner accès à des colonnes ou des lignes spécifiques d'une table, de garder le résultat de jointures sans devoir les calculer à chaque fois, ou encore de personnaliser l'affichage des données. Dans le cadre de mon stage elles ont été utilisées dans le même but que les indexes, simplifier l'accès aux mesures d'un paramètre pour une station donnée. Nous avons décidé (avec mon maître de stage) que la meilleure utilisation des vues serait de créer une vue regroupant toutes les mesures de chaque paramètre d'une station, ce qui donne de façon généralisée (avec P étant un paramètre et S une station) :

```
CREATE OR REPLACE VIEW measures_S.id_P.id AS SELECT * FROM MEASURES
WHERE station_id = S.id AND parameter_id = P.id;
```

En appliquant cette requête à tous les paramètres de la station Dumont d'Urville j'ai obtenu toutes mes vues et j'ai pu effectuer des tests.

De plus dans le cadre de pérennisation du projet il était nécessaire de pouvoir gérer les vues de façon automatique, en effet lorsque une station ou un paramètre est créé ou supprimé il est nécessaire de mettre à jour celles-ci. J'ai pu mettre cela en place grâce aux Callbacks et Association Callbacks d'ActiveRecord, il s'agit de fonctions de rappel qui peuvent être définies dans le modèle d'un objet et qui peuvent être déclenchées durant le cycle de vie d'objets et de collections de la base. Voici un exemple du modèle des stations :

```
class Station < ActiveRecord : :Base
  has_and_belongs_to_many  :campains, uniq : true
  has_and_belongs_to_many  :parameters, after_add :  :create_view, uniq : true
  belongs_to  :comment
  has_many  :measures, dependent :  :destroy
  before_destroy  :delete_views

  def create_view(parameter)
    self.parameters.each do |p|
      sql = 'CREATE OR REPLACE VIEW measures_'+self.id.to_s+'_'+p.id.to_s+'
      AS SELECT * FROM MEASURES WHERE station_id = '+self.id.to_s+'
      AND parameter_id = '+p.id.to_s+' ;'
      ActiveRecord : :Base.connection.execute(sql)
    end
  end
```

```
    def delete_views
      self.parameters.each do |p|
        sql = 'DROP VIEW IF EXISTS measures_'+self.id.to_s+'_'+p.id.to_s+' ;'
        ActiveRecord : :Base.connection.execute(sql)
      end
    end
  end
```

Ainsi en utilisant `after_add` et `before_destroy` j'ai pu définir :

— A l'ajout d'un nouveau paramètre : La création d'une vue des mesures du nouveau paramètre de la station.

— A la suppression de la station : La suppression de toutes les vues liées à cette station.

J'ai également implémenté un comportement identique pour les paramètres.

3.5 Perspectives

Le stage arrivant à son échéance le développement de nouvelles fonctionnalités a été arrêté afin de laisse place à différents tests de performances. Tout au long du projet des ajustements ont été mis en place et différentes idées ont été abandonnées ou repoussées ainsi que de nombreuses perspectives futures ont été envisagées mais ne seront malheureusement pas effectuées dans le cadre du stage. La plus importante des modifications a été l'abandon de la tâche de qualification. Une fois le graphe des paramètres mis en place, le web s'est révélé être plus complexe que prévu lorsqu'il est question d'interaction avec la librairie. Highstock ne permet pas énormément de fonctionnalités à ce niveau, cela aurait donc demandé d'écrire énormément de Javascript en plus. Il serait plus envisageable de s'orienter vers un autre langage, plus adapté, afin de développer un outil de qualification qui utiliserait la base de donnée heuliad.

D'autres idées d'optimisation et de fonctionnalités se sont ajoutées au fur et à mesure du développement, mais on été mises de coté par soucis de temps et/ou de priorités. Par exemple dans cadre du regroupement des données scientifiques dans la base heuliad il a été imaginé d'ajouter une gestion de comptes utilisateurs. Ainsi un scientifique aurait accès aux données mais aussi les permissions de les modifier, alors qu'une personne extérieure aurait accès uniquement à la visualisation de la données. L'import interactif quant à lui est assez tribal et des améliorations ont été pensées, comme la possibilité d'ajouter à la base les paramètres non reconnus dans l'entête du fichier ou encore le téléchargement du fichier de log contenant les différentes mesures n'ayant pas pu être importées.

Enfin une solution possible à la synchronisation de plusieurs graphes avait été apportée par Mr.Babau lors de sa visite à l'IUEM, cela consistait à n'utiliser qu'une seule frise temporelle pour tous les graphes. C'était envisageable, mais cela nécessitait une importante re-factorisation du code d'Highstock et limitait les possibilités offertes par la fonctionnalité. Et ainsi de suite pour de nombreuses améliorations, le projet était vraiment digne d'intérêt et pourrait devenir outil très utile à la recherche.

Conclusion

Ce stage m'a permis d'avoir une idée plus concrète du travail dans le milieu de la recherche. Le fait d'être en compagnie des chercheurs et donc des futurs utilisateurs permettait d'avoir un retour immédiat ainsi que des demandes spécifiques à leur besoins.

J'ai pu découvrir de nouvelles technologies que je ne connaissais pas comme Ruby et Rails qui permettent la construction d'un projet web (site plus bdd) très simplement. L'utilisation d'ActiveRecord a permis la simplification de nombreuses tâches et d'arriver à un résultat assez rapidement mais malheureusement au prix de beaucoup de performances. A de multiples reprises j'ai été dans l'obligation de contourner son utilisation afin d'obtenir un comportement convenable avec un ensemble important de données. De plus, avec le recul du stage, Rails et le web par extension se révèle être un outil très pratique pour la visualisation mais dès que des interactions sont nécessaires la tâche se complique énormément.

J'ai également pu apprécier la grande utilité que représente git, que j'ai beaucoup trop sous-estimé en début de stage. Je me suis rendu compte au moment de la rédaction de mon rapport et des phases de tests qu'il permet, lorsqu'il est bien utilisé, de retracer très facilement l'évolution d'un projet et possède un tas d'outils et de métriques facilitant ce travail. Une autre découverte de ce stage a été l'application web Redmine qui couplée avec le dépôt git a permis une gestion du projet parfaite. La possibilité d'enregistrer des fonctionnalités et bugs à développer et résoudre laisse, en tout moment, une idée de l'orientation que prends le projet ainsi que du travail restant.

Le stage ayant une importante partie dédiée à la gestion de gros ensemble de données j'ai pu découvrir en quoi l'optimisation pouvait changer du tout au tout le comportement d'un algorithme. Le simple fait de modéliser les mesures en objet ne me paraissait pas si grave, mais s'est révélé doublé le temps de certains affichages. Ainsi la première solution trouvée à un problème était rarement la bonne.

Glossaire

ActiveRecord : Il s'agit d'un patron de conception utilisé afin de lire les données d'une base de données. Les attributs d'une table ou d'une vue sont encapsulés dans une classe. Ainsi l'objet, instance de la classe, est lié à un tuple de la base. Après l'instanciation d'un objet, un nouveau tuple est ajouté à la base au moment de l'enregistrement. Chaque objet récupère ses données depuis la base ; quand un objet est mis à jour, le tuple auquel il est lié l'est aussi.

Data-grouping : Le data-grouping remplace une séquence de points de données d'une série avec un point groupé. Les valeurs de chaque point groupé sont calculées à partir des valeurs d'origine de chaque point utilisé.

Framework : Un framework ou structure logicielle est un ensemble cohérent de composants logiciels structurels, qui sert à créer les fondations ainsi que les grandes lignes de tout ou d'une partie d'un logiciel (architecture). Les frameworks sont conçus et utilisés pour modeler l'architecture des logiciels applicatifs, des applications web, des middlewares et des composants logiciels.

Gemmes : Les gemmes sont les librairies ou bibliothèques du langage Ruby. RubyGems est un gestionnaire de paquets pour le langage de programmation Ruby qui permet de distribuer des programmes et des bibliothèques Ruby, c'est un outil conçu pour gérer facilement l'installation de gemmes et un serveur pour les distribuer.

Helpers : Dans la programmation orientée objet, une classe Helpers est utilisée pour fournir certaines fonctionnalités, ce qui n'est pas l'objectif principal de l'application ou de la classe dans laquelle elle est utilisée. Une instance d'une classe helper s'appelle un objet helper. Dans le cadre de Ruby il peut s'agir de fonction permettant d'écrire des formulaires HTML sans en écrire une seule ligne.

Json : JSON (JavaScript Object Notation) est un format de données textuelles dérivé de la notation des objets du langage JavaScript. Il permet de représenter de l'information structurée comme le permet XML par exemple. Il sert à faire communiquer des applications dans un environnement hétérogène. Il est notamment utilisé comme langage de transport de données par AJAX et les services Web.

LaTeX : LaTeXest un langage créé pour séparer le fond de la forme lors de la création d'un document,la structure du texte est effectuée grâce à des mots-clés et des commandes propres à LaTeX. Il donne les moyens d'obtenir des documents mis en page de façon professionnelle sans avoir à se soucier de leur forme. La priorité est donnée à l'essentiel : le contenu.

Librairie : Une librairie, également appelée bibliothèque logicielle, est une collection de fonctions prête à l'utilisation. Le plus souvent se sont des projets déjà réalisés qui sont ajoutés à notre programme afin d'éviter de perdre du temps à les réécrire.

Package : C'est une archive de fichiers qui de la même façon que les bibliothèques informatiques apporte des fonctions et des variables à un programme. Cela permet l'utilisation de nouvelles fonctions et d'utiliser des parties de projet qui peuvent tout à fait concerner le vôtre.

Structure : En informatique, on parle de structure de données, elles servent à contenir des données et leur donnent une organisation permettant de simplifier leur traitement.

Liste des figures

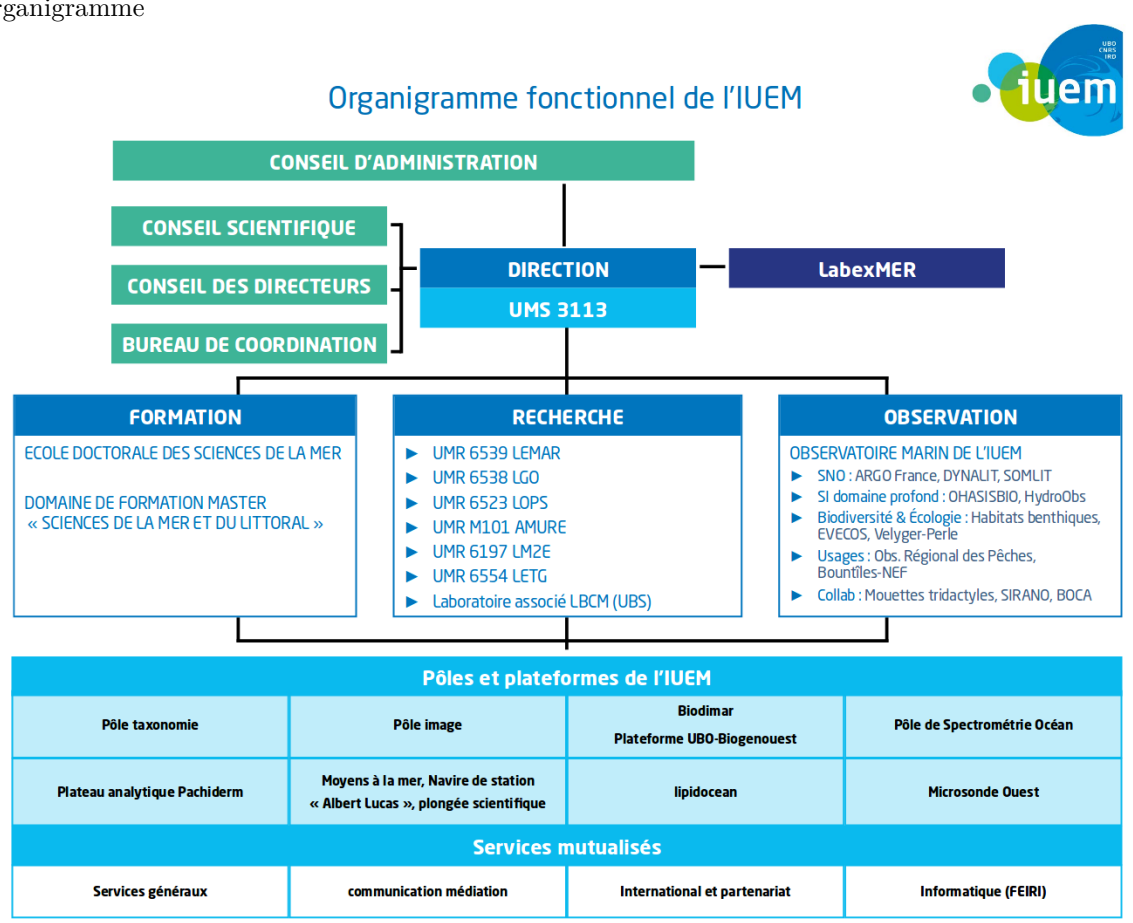
Liste des tableaux

Table des annexes

Annexes

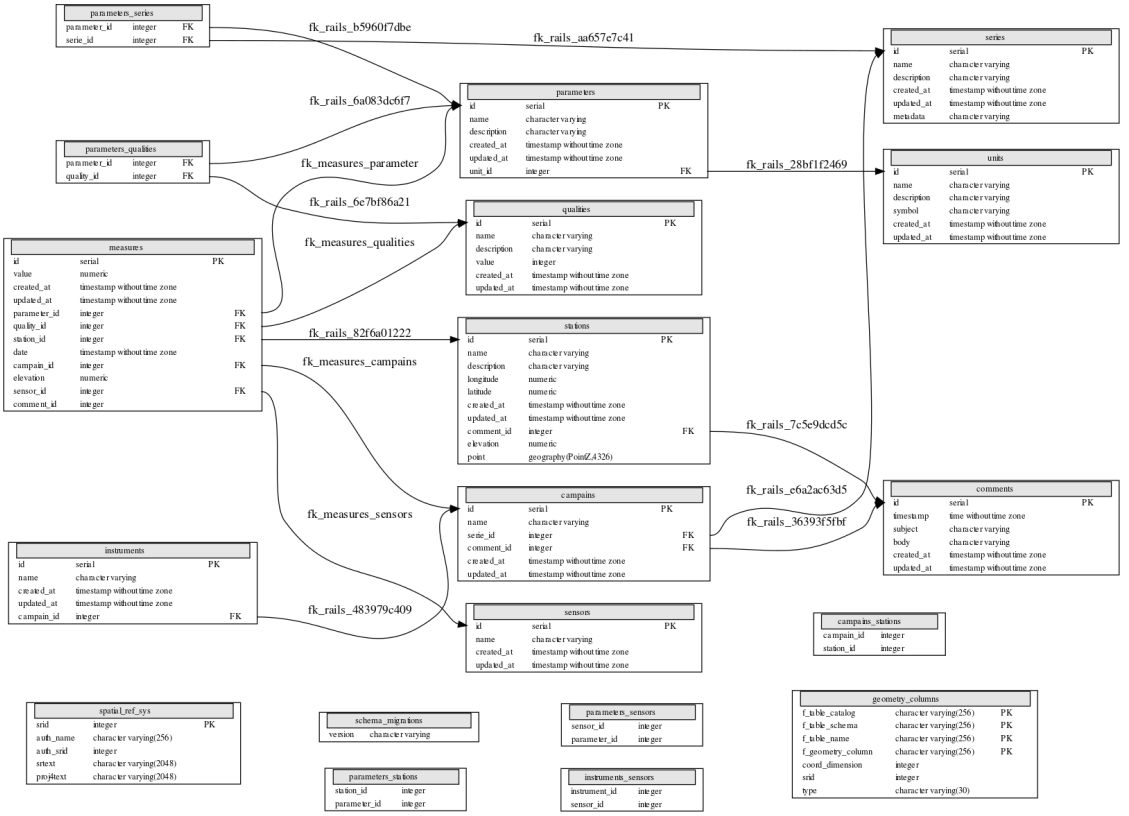
3.6 Organigramme

Organigramme



3.7 Base de données

Base de données



3.8 Fichiers et leurs formats

Fichiers et leurs formats

3.8.1 Format 1 1997-2016

Une mesure toutes les heures. Un fichier par paramètre (twat bot cwat), twat et bot de février 1997 à juillet 2016. Cwat janvier 2006 à juillet 2016. En 2006 et 2012, le marégraphe n'est pas installé tout à fait à la même hauteur. En 2008, le site de mesure n'est pas tout à fait au même endroit.

```
28/02/1997 06h00mn    -0.300
28/02/1997 06h00mn  1634.600
09/01/2006 06h00mn   27.300
```

3.8.2 Format 2 1997-2005

Une mesure toutes les 30 minutes.

```
id    date       heure        twat     bot          cwat      baro    flags
14351 28/02/1997 05:13:00     -0.400   1647.300     13.400    984.100 16 15 16 16
```

3.8.3 Format 3 2006-2010

Une mesure toutes les 2 minutes.

```
date    heure      twat      bot       cwat      baro
13/01/2006 23:12:00     -1.12   1462.70     27.53    976.90
```

3.8.4 Format 4 2007-2008 2011-2012

Une mesure toutes les 30 minutes.

```
date    heure      twat      bot       cwat
24/02/2007 08:00:00     -1.517   2663.195   27.089
```

3.8.5 Format 5 2012

Une mesure toutes les 2 minutes. Un fichier par jour.

```
aaaammjj hh:mm:ss tics rescueflag therm Vbat Conduc Temp Press Par Oxconc Oxsat Oxtemp Fluo
20120117 08:42:00 17002 0 11.84 13.16 -0.0185 5.967 1.018523 181.712 320.08 91.81 10.44 0.02
```

3.8.6 Format 6 2014-2016

Une mesure toutes les 2 minutes Format utilisé actuellement.

```
date   heure bot     twat    cwat    baro    par     oconc   osat    otemp   fluo    bat
17/01/2014 05:32:00 1504.546   -1.590   26.889  997.919  164.110  366.430  76.930   -1.500    0.1
```

3.9 Codes Qualité

Codes Qualité

code	description
0	Données en dessous de la limite de détection
1	Mesure bonne, échantillon non répliqué
2	Données en dessous de la limite de détection
3	Mesure douteuse
4	Mesure mauvaise
5	Prélèvement effectué, mais valeur pas encore reportée
6	Mesure bonne (moyenne de plusieurs réplicats)
7	Mesure bonne (valeur acquise hors protocoles SOMLIT)
8	Donnée non qualifiée
9	Echantillon non prélevé

3.10 Résultats ActiveRecord VS Tableau de caractères

Résultats ActiveRecord VS Tableau de caractères

Pas agrégation pas de vues

ActiveRecords :

```
Measure Load (875.2ms)  SELECT  "measures".* FROM "measures"
WHERE "measures"."station\_id" = 711 AND "measures"."parameter_id" = 169
AND "measures"."quality_id" = 2 AND ("measures"."date" BETWEEN '1997-02-28 05:43:00'
AND '2031-01-12 11:02:00') LIMIT 50000
```

PostgreSQL :

```
SELECT * FROM measures WHERE (date BETWEEN '1997-02-28 05:43:00' AND '2031-01-12 11:02:00' )
AND (parameter_id = 169 AND station_id= 711 )
AND quality_id = 2 LIMIT 50000 ;
```

limit	type	user	system	total	real	request
50 000	ActiveRecord	1.160000	0.080000	1.240000	2.124812	875.2ms
	PostgreSQL 1/3	0.830000	0.050000	0.880000	1.355225	604.9ms
	PostgreSQL 2/3	0.920000	0.020000	0.940000	1.364945	573.9ms
Facteur : 1.58	PostgreSQL 3/3	0.700000	0.030000	0.730000	1.341667	748.7ms
100 000	ActiveRecord	2.750000	0.130000	2.880000	3.879179	1212.2ms
	PostgreSQL 1/3	1.890000	0.110000	2.000000	2.942919	1219.2ms
	PostgreSQL 2/3	1.370000	0.070000	1.440000	2.268550	1082.2ms
Facteur : 1.32	PostgreSQL 3/3	1.420000	0.060000	1.480000	2.392262	1181.8ms
500 000	ActiveRecord	16.790000	0.640000	17.430000	24.273150	8098.5ms
	PostgreSQL 1/3	8.390000	0.250000	8.640000	12.278865	4930.4ms
	PostgreSQL 2/3	9.410000	0.210000	9.620000	13.147484	4833.7ms
Facteur : 1.98	PostgreSQL 3/3	7.000000	0.150000	7.150000	10.560596	4701.8ms
1 000 000	ActiveRecord	41.830000	1.120000	42.950000	52.887709	12362.1ms
	PostgreSQL 1/3	19.550000	0.640000	20.190000	36.406959	16708.6ms
	PostgreSQL 2/3	20.440000	0.500000	20.940000	28.742027	10517.2ms
Facteur : 1.99	PostgreSQL 3/3	18.210000	0.420000	18.630000	26.573625	10701.6ms

Pas agrégation des vues

ActiveRecords :

```
SELECT  "measures".* FROM "measures" WHERE "measures"."station_id)" = 711
AND "measures"."parameter_id" = 169 AND "measures"."quality_id" = 2
AND ("measures"."date" BETWEEN '1997-02-28 05:43:00' AND '2031-01-12 11:02:00') LIMIT 50000
```

PostgreSQL :

```
SELECT * FROM measures_711_169)
WHERE (date BETWEEN '1997-02-28 05:43:00' AND '2031-01-12 11:02:00' )
AND (parameter_id = 169 AND station_id= 711 )AND quality_id = 2 LIMIT 50000 ;
```

limit	type	user	system	total (s)	real (s)	request (ms)
50 000	ActiveRecord	1.380000	0.080000	1.460000	2.535235	1006.6
	PostgreSQL 1/3	0.730000	0.050000	0.780000	1.648142	992.0
	PostgreSQL 2/3	1.040000	0.020000	1.060000	1.679074	755.3
Facteur : 1.54	PostgreSQL 3/3	0.630000	0.010000	0.640000	1.213347	695.3
100 000	ActiveRecord	2.630000	0.120000	2.750000	4.477940	1923.1
	PostgreSQL 1/3	2.080000	0.070000	2.150000	3.277471	1379.1
	PostgreSQL 2/3	1.530000	0.070000	1.600000	3.052062	1706.5
Facteur : 1.47	PostgreSQL 3/3	1.600000	0.030000	1.630000	2.995076	1579.8
500 000	ActiveRecord	17.520000	0.580000	18.100000	26.484757	9515.7
	PostgreSQL 1/3	8.360000	0.290000	8.650000	13.036433	5616.5
	PostgreSQL 2/3	8.790000	0.240000	9.030000	13.442934	5633.4
Facteur : 1.97	PostgreSQL 3/3	7.990000	0.180000	8.170000	12.646635	5590.9
1 000 000	ActiveRecord	42.250000	1.170000	43.420000	51.730069	10367.1
	PostgreSQL 1/3	19.610000	0.640000	20.250000	70.038761	22982.5
	PostgreSQL 2/3	20.390000	0.580000	20.970000	33.535705	13672.1
Facteur : 1.98	PostgreSQL 3/3	18.620000	0.270000	18.890000	26.119893	9762.0

Agrégation hour pas de vues

ActiveRecord :

```
SELECT  "measures".* FROM "measures" WHERE "measures"."station_id" = 711
AND "measures"."parameter_id" = 169 AND "measures"."quality_id" = 2
AND ("measures"."date" BETWEEN '1997-02-28 05:43:00' AND '2031-01-12 11:02:00') LIMIT 50000
```

PostgreSQL :

```
SELECT extract(epoch from date_trunc('hour', date))*1000 as ts, avg(value)
FROM measures WHERE (date BETWEEN '1997-02-28 05:43:00' AND '2031-01-12 11:02:00' )
AND (parameter_id = 169 AND station_id= 711 ) AND quality_id = 2 GROUP BY ts LIMIT 50000 ;
```

limit	type	user	system	total	real	request
50 000	ActiveRecord	1.210000	0.080000	1.290000	2.919160	1608.8ms
	PostgreSQL 1/3	0.410000	0.020000	0.430000	20.207478	19781.0ms
	PostgreSQL 2/3	0.210000	0.000000	0.210000	11.712617	11508.2ms
Facteur : 0.28	PostgreSQL 3/3	0.240000	0.000000	0.240000	10.402933	10158.5ms
100 000	ActiveRecord	2.630000	0.120000	2.750000	3.812271	1160.2ms
	PostgreSQL 1/3	0.730000	0.020000	0.750000	10.136425	9425.6ms
	PostgreSQL 2/3	0.430000	0.020000	0.450000	8.204135	7805.5ms
Facteur : 0.46	PostgreSQL 3/3	0.390000	0.010000	0.400000	9.164421	8783.6ms
500 000	ActiveRecord	17.400000	0.530000	17.930000	27.284832	9919.2ms
	PostgreSQL 1/3	0.340000	0.060000	0.400000	16.343373	15837.3ms
	PostgreSQL 2/3	1.550000	0.020000	1.570000	15.909153	14238.4ms
Facteur : 2.77	PostgreSQL 3/3	0.650000	0.010000	0.660000	9.858007	9226.0ms
1 000 000	ActiveRecord	42.490000	1.380000	43.870000	62.820855	10516.1ms
	PostgreSQL 1/3	0.570000	0.040000	0.610000	11.736969	10641.4ms
	PostgreSQL 2/3	0.540000	0.040000	0.580000	12.197318	11707.2ms
Facteur : 5.35	PostgreSQL 3/3	0.460000	0.010000	0.470000	17.957433	17548.8ms

Agrégation hour des vues

ActiveRecord :

```
SELECT  "measures".* FROM "measures" WHERE "measures"."station_id" = 711
AND "measures"."parameter_id" = 169 AND "measures"."quality_id" = 2
AND ("measures"."date" BETWEEN '1997-02-28 05:43:00' AND '2031-01-12 11:02:00') LIMIT 50000
```

PostgrSQL :

```
SELECT extract(epoch from date_trunc('hour', date))*1000 as ts, avg(value)
FROM measures_711_169) WHERE (date BETWEEN '1997-02-28 05:43:00'
AND '2031-01-12 11:02:00' ) AND (parameter_id = 169 AND station_id= 711 )
AND quality_id = 2 GROUP BY ts LIMIT 50000;
```

limit	type	user	system	total	real	request
50 000	ActiveRecord	1.220000	0.070000	1.290000	2.168300	757.3ms
	PostgreSQL 1/3	0.410000	0.020000	0.430000	8.471523	8049.5ms
	PostgreSQL 2/3	0.190000	0.010000	0.200000	7.996041	7804.9ms
Facteur : 0.27	PostgreSQL 3/3	0.240000	0.000000	0.240000	8.250650	7994.1ms
100 000	ActiveRecord	2.850000	0.130000	2.980000	4.003584	1075.4ms
	PostgreSQL 1/3	0.800000	0.020000	0.820000	10.029438	9176.8ms
	PostgreSQL 2/3	0.450000	0.020000	0.470000	9.128613	8619.9ms
Facteur : 0.49	PostgreSQL 3/3	0.420000	0.020000	0.440000	8.233604	7811.4ms
500 000	ActiveRecord	18.530000	0.730000	19.260000	27.515003	8576.7ms
	PostgreSQL 1/3	0.360000	0.040000	0.400000	12.298775	11893.9ms
	PostgreSQL 2/3	1.730000	0.030000	1.760000	15.347704	13473.7ms
Facteur : 2.60	PostgreSQL 3/3	0.670000	0.010000	0.680000	10.590090	9927.2ms
1 000 000	ActiveRecord	42.680000	1.230000	43.910000	59.503445	10927.1ms
	PostgreSQL 1/3	0.580000	0.030000	0.610000	8.983180	8260.3ms
	PostgreSQL 2/3	0.520000	0.020000	0.540000	9.340969	8843.8ms
Facteur : 6.45	PostgreSQL 3/3	0.420000	0.010000	0.430000	9.227083	8844.3ms

Agrégation day pas de vues

ActiveRecord :

```
SELECT  "measures".* FROM "measures" WHERE "measures"."station_id" = 711
AND "measures"."parameter_id" = 169 AND "measures"."quality_id" = 2
AND ("measures"."date" BETWEEN '1997-02-28 05:43:00' AND '2031-01-12 11:02:00') LIMIT 50000
```

PostgreSQL :

```
SELECT extract(epoch from date_trunc('day', date))*1000 as ts, avg(value)
FROM measures WHERE (date BETWEEN '1997-02-28 05:43:00' AND '2031-01-12 11:02:00' )
AND (parameter_id = 169 AND station_id= 711 ) AND quality_id = 2 GROUP BY ts LIMIT 50000;
```

limit	type	user	system	total	real	request
50 000	ActiveRecord	1.390000	0.050000	1.440000	2.124401	588.4ms
	PostgreSQL 1/3	0.010000	0.000000	0.010000	8.068408	8050.3ms
	PostgreSQL 2/3	0.020000	0.000000	0.020000	8.301157	8284.5ms
Facteur : 0.29	PostgreSQL 3/3	0.020000	0.000000	0.020000	7.368104	7349.0ms
100 000	ActiveRecord	2.670000	0.160000	2.830000	4.144292	1534.4ms
	PostgreSQL 1/3	0.010000	0.000000	0.010000	10.035129	10020.7ms
	PostgreSQL 2/3	0.020000	0.000000	0.020000	9.110425	9096.1ms
Facteur : 0.45	PostgreSQL 3/3	0.020000	0.000000	0.020000	10.760077	10745.4ms
500 000	ActiveRecord	16.650000	0.550000	17.200000	21.113447	5200.1ms
	PostgreSQL 1/3	0.020000	0.000000	0.020000	8.089898	8073.8ms
	PostgreSQL 2/3	0.010000	0.010000	0.020000	7.055021	7039.1ms
Facteur : 2.99	PostgreSQL 3/3	0.020000	0.000000	0.020000	7.235653	7220.0ms
1 000 000	ActiveRecord	40.700000	1.330000	42.030000	50.883941	9989.7ms
	PostgreSQL 1/3	0.010000	0.010000	0.020000	12.309679	11605.4ms
	PostgreSQL 2/3	0.010000	0.010000	0.020000	10.933771	10917.8ms
Facteur : 4.65	PostgreSQL 3/3	0.010000	0.010000	0.020000	16.945297	16928.8ms

Agrégation day des vues

ActiveRecord :

```
SELECT  "measures".* FROM "measures" WHERE "measures"."station_id" = 711
AND "measures"."parameter_id" = 169 AND "measures"."quality_id" = 2
AND ("measures"."date" BETWEEN '1997-02-28 05:43:00' AND '2031-01-12 11:02:00') LIMIT 50000
```

PostgreSQL :

```
SELECT extract(epoch from date_trunc('day', date))*1000 as ts, avg(value)
FROM measures_711_169) WHERE (date BETWEEN '1997-02-28 05:43:00'
AND '2031-01-12 11:02:00' ) AND (parameter_id = 169
AND station_id= 711 ) AND quality_id = 2 GROUP BY ts LIMIT 50000 ;
```

limit	type	user	system	total	real	request
50 000	ActiveRecord	1.140000	0.080000	1.220000	1.649282	525.9ms
	PostgreSQL 1/3	0.020000	0.000000	0.020000	7.882485	7868.1ms
	PostgreSQL 2/3	0.020000	0.000000	0.020000	7.599660	7585.0ms
Facteur : 0.22	PostgreSQL 3/3	0.010000	0.010000	0.020000	8.110927	8096.3ms
100 000	ActiveRecord	2.680000	0.160000	2.840000	4.908916	2209.4ms
	PostgreSQL 1/3	0.010000	0.00000	0.010000	13.212089	13197.8ms
	PostgreSQL 2/3	0.010000	0.010000	0.020000	12.650778	12635.6ms
Facteur : 0.54	PostgreSQL 3/3	0.020000	0.00000	0.020000	9.013775	8999.3ms
500 000	ActiveRecord	16.580000	0.610000	17.190000	21.963939	6061.9ms
	PostgreSQL 1/3	0.020000	0.000000	0.020000	8.587509	8571.7ms
	PostgreSQL 2/3	0.020000	0.000000	0.020000	8.123731	8108.3ms
Facteur : 2.78	PostgreSQL 3/3	0.010000	0.000000	0.010000	7.913364	7897.5ms
1 000 000	ActiveRecord	40.650000	1.320000	41.970000	49.235981	9854.7ms
	PostgreSQL 1/3	0.010000	0.000000	0.010000	7.287568	7271.8ms
	PostgreSQL 2/3	0.020000	0.010000	0.030000	7.874495	7858.4ms
Facteur : 6.25	PostgreSQL 3/3	0.020000	0.000000	0.020000	7.722069	7702.3ms

3.11 Évolution temps affichage vue station DDU

Évolution temps affichage vue station DDU

Les temps sont en millisecondes.

Commit	bdd	activerecord	vue	rendu client	total
6e190e17	30010.7	30077.7	425630.9	454891.9	455714
6ekrrr	26027.8	26046.6	430077.3	455400.1	456129
7fef3312	25859.4	25909.2	413267.5	438930.2	486623
f73fe0d2 (Affichage de métriques)	9325.3	9579.4	288.8	9838.8	9873
ac5edf56 (ActiveRecord => PostgreSQL)	4402.4	4418.4	129.8	4520.1	4600
37187f2f	4011.0	4015.3	13.5	4025.6	4034
a315184c	3537.5	3542.6	16.7	3555.4	3564
e1e10a20	6805.2	6829.1	75.9	6896.5	6939
bd1db8aa	3645.5	3666.8	124.8	3663.3	3797
87f22ec1	3573.4	3593.1	129.3	3602.7	3728
master_no_agg	8949.5	8991.2	148.2	8982.4	9144
master_no_view_no_agg	6296.6	6321.2	136.7	6329.6	6465
master_no_view	8137.1	8240.7	308.3	8301.3	8650
master	4305.1	4351.7	257.9	4417.4	4702