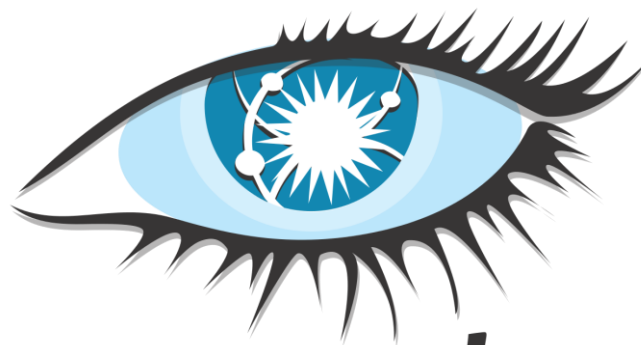




RAPPORT DE PROJET COMPILATION



cassandra

Réalisation d'un Interpréteur CQL Cassandra Query Language

Réalisé par :

- KANGA Dominique Bernard
- NIDABRAHIM Youssef
- CHOKRI Khaoula
- NAJI Kawtar

Encadré par :

- Mr. Karim BAINA

Remerciements

Avant d'entamer ce rapport de projet de Compilation, on tient à exprimer nos sincères gratitude envers tous ceux qui nous ont aidé ou ont participé au bon déroulement de ce projet.

Nos sincères remerciements à **Monsieur KARIME BAINA**, notre encadrant durant le déroulement de ce projet, pour sa générosité, sa disponibilité permanente, son guide et ses conseils et son aide inestimable.

Table des matières

Remerciements	2
Liste des abréviations.....	4
Table des figures	5
Introduction générale	6
CHAPITRE I : Présentation de Cassandra	7
1. Les bases de données NoSQL	8
2. Historique de Cassandra	8
3. Les concepts de Cassandra	9
CHAPITRE II : CQL : Cassandra Query Language	11
1. Présentation de CQL	12
2. Grammaire.....	12
2.1. Requêtes de base	12
2.2. Les collections.....	14
CHAPITRE III : Analyse	16
1. Analyseur lexical.....	17
1.1. Concepts	17
1.2. Les unités lexicales	17
2. Analyseur syntaxique	18
3. Analyseur sémantique.....	20
CHAPITRE IV : Interpréteur CQL.....	22
CHAPITRE V : Apport du projet.....	31
1. Apports scientifiques et techniques	32
2. Apport sur la formation pédagogique	32
3. Apport personnel	32
Conclusion et Perspectives.....	33

Liste des abréviations

Abréviation	Désignation
CQL	Cassandra Query Language
NOSQL	Not Only SQL
SQL	Structed Query Language
SGBD	Système de Gestion de Base de Données
JSON	JSON
RI	Représentation Intermédiaire

Table des figures

Figure 1: Grammaire STATEMENTS	12
Figure 2: Grammaire STATEMENT	13
Figure 3: Grammaire CREATE_KEYSPACE.....	13
Figure 4: Grammaire ALTER_TABLE	13
Figure 5: Grammaire CREATE_INDEX.....	14
Figure 6: Grammaire SELECT	14
Figure 7: Grammaire SET	15
Figure 8: Grammaire LIST	15
Figure 9: Grammaire MAPS	15
Figure 10: CREATE_KEYSPACE en C.....	19
Figure 11: Les erreurs sémantiques.....	21
Figure 12: Structure table.....	24
Figure 13: Structure Colonnes	24
Figure 14: Structure Keyspace	24
Figure 15: Représentation intermédiaire de dictionnaire de données.....	25
Figure 16: Représentation intermédiaire de lignes des tables.....	25
Figure 17: Représentation intermédiaire des erreurs.....	26
Figure 18: L'accueil	27
Figure 19: Test - USE	28
Figure 20: Test - SHOW	28
Figure 21: Test - DESCRIBE	29
Figure 22: Teste - SELECT.....	29
Figure 23: Teste - INSERT.....	30

Introduction générale

Le web connaît un succès tel que certains acteurs sont confrontés à une très forte audience qui continuera à croître de façon exponentielle si tout va bien pour eux, par exemple Google gère 700 000 recherches par seconde et 98 000 nouveaux tweets par seconde. Face à ces cas d'utilisations hors norme, des solutions technologiques sur mesure ont dû être créées par les grands du Web.

Les bases de données relationnelles n'ayant pas été créées pour cela, une catégorie de bases de données (le Not Only SQL) s'est développée en faisant le compromis d'abandonner certaines fonctionnalités classiques des SGBD relationnelles. En effet Cassandra est parmi ces bases de données NoSQL orientées colonnes.

Le projet qu'on présente, qui s'inscrit dans le cadre de notre projet académique de compilation, aborde précisément la réalisation d'un interpréteur CQL Cassandra Query Language.

On présente dans ce rapport les différentes étapes suivies pour la réalisation de ce projet. Il est axé sur quatre grandes parties. On a commencé par une étude préliminaire qui consiste à préciser de façon claire les objectifs du projet ainsi que son contexte, ensuite vient la phase de l'analyseur lexicale, à ce stade on a spécifié une liste des unités lexicales. Puis on a entamé la phase d'analyseur syntaxique afin de bien vérifier la syntaxe des requêtes CQL. Après on a réalisé un analyseur sémantique qui détecte tous les anomalies. Enfin vient la phase de l'interprétation des requêtes CQL qui permet la construction du projet.

CHAPITRE I

Présentation de Cassandra

Introduction

Cassandra est une base de données NOSQL orientée colonnes, destinée pour de grands volumes de données, hétérogènes, de structure et taille évolutives et hautement disponibles, sans compromettre la performance. Un très large panel d'entreprises, dont *Twitter*, *Netflix* et *eBay*, utilisent Cassandra.

1. Les bases de données NoSQL

Le NoSQL, pour Not only SQL, désigne les bases de données qui ne sont pas fondées sur l'architecture classique des bases de données relationnelles. Développé à l'origine pour gérer du big data, l'utilisation de base de données NoSQL a explosée depuis quelques années.

Lorsque l'on parle de NoSQL, on regroupe des systèmes de base de données qui ne sont pas relationnels, mais il faut savoir qu'il existe plusieurs types de bases de données NoSQL.

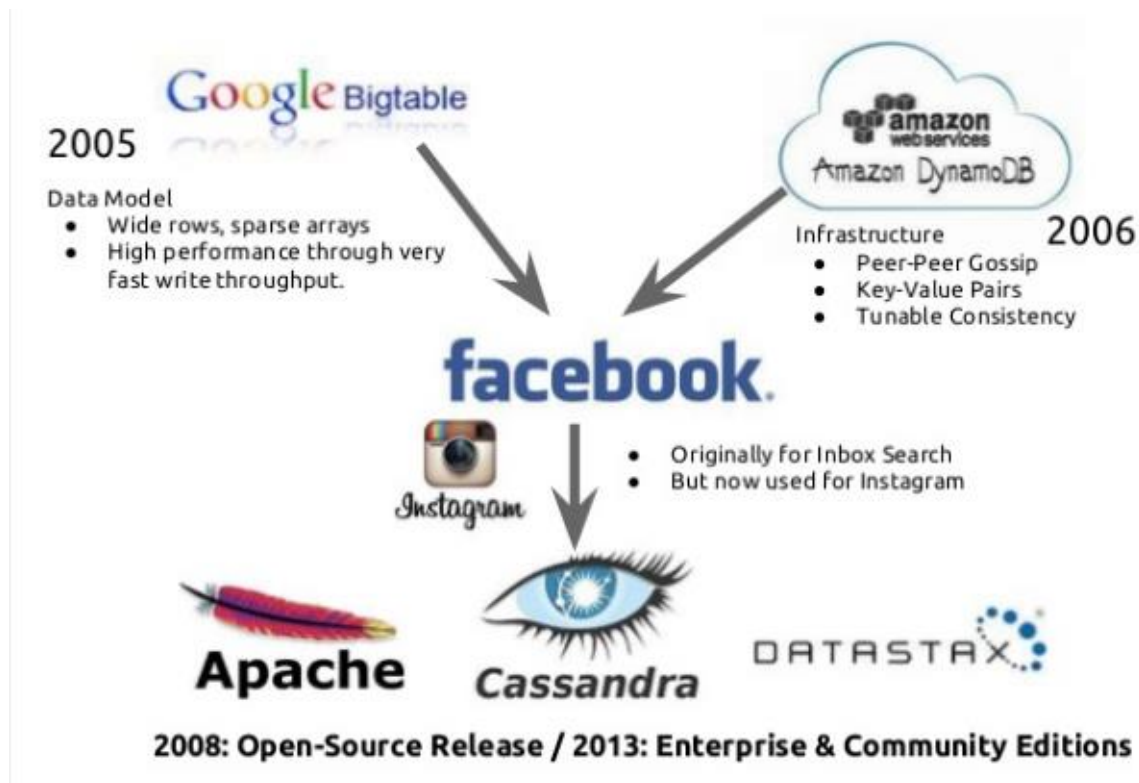
- Les bases clef/valeur, permettent de stocker des informations sous forme d'un couple clef/valeur où la valeur peut être une chaîne de caractère, un entier ou un objet sérialisé, par exemple *Redis*.
- Les bases orientées colonnes, ressemble aux bases de données relationnelles, car les données sont sauvegardées sous forme de ligne avec des colonnes, mais se distingue par le fait que le nombre de colonnes peut varier d'une ligne à l'autre. Les solutions les plus connues sont *HBase* ou *Cassandra*.
- Les bases orientées document, représente les informations sous forme d'objet XML ou JSON. L'avantage est de pouvoir récupérer simplement des informations structurées de manière hiérarchique. Les solutions les plus connues sont *Couch*, *RavenDB* et *MongoDB*.
- Les bases orientées graphe, présentent les données sous forme de nœud et de relation. Cette structure permet de récupérer simplement des relations complexes. Un exemple de base graphe est *Neo4J*.

2. Historique de Cassandra

Initialement développée par *Facebook*, l'application a été libérée dans l'espace open source en juillet 2008, et rapidement adoptée par des entreprises gérant d'importants volumes de données, telles que *Twitter*, *Netflix*, *digg.com* ou *Spotify*.

Jeff Hammerbacher, directeur de la division Data chez Facebook, a décrit Cassandra comme une application du type de BigTable servie par une infrastructure proche d'Amazon DynamoDB.

En 2015 Cassandra occupe la 8^e place dans le classement des systèmes de gestion de bases de données les plus populaires tous types confondus, et la deuxième place pour les systèmes de gestion de bases de données NoSQL.



3. Les concepts de Cassandra

Cassandra est un système permettant de gérer une grande quantité de données de manière distribuée. Ces dernières peuvent être structurées, semi-structurées ou pas structurées du tout.

Cassandra a été conçue pour être hautement scalable sur un grand nombre de serveurs, Cassandra fournit un schéma de données dynamique afin d'offrir un maximum de flexibilité et de performance.

Mais pour bien comprendre cet outil, il faut tout d'abord bien assimiler le vocabulaire de base.

- **Keyspace** : c'est l'équivalent d'une *database* dans le monde des bases de données relationnelles. À noter qu'il est possible d'avoir plusieurs *Keyspaces* sur un même serveur.

- **Colonne (Column)** : une colonne est composée d'un nom, d'une valeur et d'un timestamp (instant de création).
- **Ligne (Row)** : les colonnes sont regroupées en lignes. Une ligne est représentée par une clé et une valeur. Il existe deux types de lignes : les *wide row* permettant de stocker énormément de données, avec beaucoup de colonnes et les *skinny row* permettant de stocker peu de données.
- **Une famille de colonnes (Column family)** : c'est l'objet principal de données et peut être assimilé à une table dans le monde des bases de données relationnelles. Toutes les lignes sont regroupées dans les colonnes family.

CHAPITRE II

CQL : Cassandra Query Language

1. Présentation de CQL

Cassandra Query Language CQL est l'interface par défaut pour la communication avec la base de données Cassandra. C'est un langage SQL-Like, pour la manipulation des données dans la base.

La meilleure façon d'interagir avec Cassandra est d'utiliser CQL Shell, `cqlsh`. En utilisant `cqlsh`, on peut créer des keyspaces et des columns family (les tables), insérer et récupérer des données depuis les tables et plus encore.

2. Grammaire

Une grammaire est une collection de règles de réécriture qui définissent mathématiquement quand une suite de symbole d'un certain alphabet constitue un mot d'un langage.

La grammaire est disponible dans la documentation de Cassandra, mais elle n'est pas LL(1), pour la rendre LL(1), on a suivi ces étapes :

- Désambiguïsation de la grammaire.
- Élimination de la récursivité gauche.
- Rendre la grammaire prédictive.

Nous citons ci-dessous la grammaire qui correspond à quelques requêtes de base de CQL.

2.1. Requêtes de base

La règle qui va représenter la liste des requêtes et considérer comme la règle principale est la suivante :

```
STATEMENTS : STATEMENT ';' LIST_STATEMENTAUX  
LIST_STATEMENTAUX : STATEMENTS | epsilon
```

Figure 1: Grammaire STATEMENTS

La règle suivante va switcher sur une de ses dérivés en fonction de la requête :

```

STATEMENT
: DROP_KEYSPACE_STMT
| CREATE_KEYSPACE_STMT
| ALTER_KEYSPACE_STMT
| USE_STMT
| CREATE_TABLE_STMT
| ALTER_TABLE_STMT
| DROP_TABLE_STMT
| TRUNCATE_TABLE_STMT
| CREATE_INDEX_STMT
| DROP_INDEX_STMT
| INSERT_STMT
| UPDATE_STMT
| DELETE_STMT
| BATCH_STMT

```

Figure 2: Grammaire STATEMENT

Pour créer une Keyspace :

```

CREATE_KEYSPACE_STMT : alter keyspace idf with PROPERTIES
PROPERTIES : PROPERTY PROPERTIES_AUX
PROPERTIES_AUX
: and PROPERTIES
| epsilon
PROPERTY : PROPERTY_NAME '=' PROPERTY_VALUE
PROPERTY_NAME : idf
PROPERTY_VALUE
: idf
| CONSTANT
| MAP
CONSTANT
: string
| integer
| float
| BOOL
| uuid
| blob

```

Figure 3: Grammaire CREATE_KEYSPACE

Pour modifier une table (il est possible de remplacer le terme TABLE par COLUMNFAMILY) :

```

ALTER_TABLE_STMT : alter ALTER_TABLE_STMT_AUX
ALTER_TABLE_STMT_AUX
: table TABLE_NAME ALTER_TABLE_INSTRUCTION
| columnfamily TABLE_NAME ALTER_TABLE_INSTRUCTION

```

Figure 4: Grammaire ALTER_TABLE

Pour créer un index sur une colonne :

```

CREATE_INDEX_STMT : create CREATE_INDEX_STMTAUX
CREATE_INDEX_STMTAUX
    : index INDEX_STMTAUX
    | custom CUSTOM_STMTAUX
INDEX_STMTAUX
    : INDEX_NAMEAUX
    | IF_NOT_EXISTS INDEX_NAMEAUX
INDEX_NAMEAUX
    : INDEX_ONAUX
    | INDEX_NAME INDEX_ONAUX
INDEX_ONAUX : on TABLE_NAME '(' COLUMN_NAME ')'
CUSTOM_STMTAUX : INDEX_STMTAUX USING_AUX
USING_AUX : using INDEX_CLASS WITH_AUX
WITH_AUX
    : with INDEX_OPTIONS
    | epsilon

```

Figure 5: Grammaire CREATE_INDEX

Afficher les données insérées :

```

SELECT_TABLE_STMT : SELECT_CLAUSE FROM TABLE_NAME
SELECT_CLAUSE : COLUMN_NAME SELECT_CLAUSE_AUX
SELECT_CLAUSE_AUX : ',' SELECT_CLAUSE | epsilon

```

Figure 6: Grammaire SELECT

2.2. Les collections

Dans les bases de données relationnelles traditionnelles, il est fortement déconseillé d'utiliser des valeurs multiples dans un même champ. Pour cela, il faudrait créer une autre table, et faire une jointure. Dans les bases NOSQL, le but ultime est de minimiser au maximum les jointures, pour faciliter et accélérer la navigation. C'est pour cette raison que des collections comme les sets, listes et maps sont utilisés.

Attention à ne pas abuser de ce type de champs: il faut s'assurer que les collections stockent de petites quantités de données, car une requête Cassandra lit une collection en entier, c'est à l'utilisateur de les parcourir ensuite.

- **Sets** : Un ensemble (set) est un ensemble non ordonné de valeurs. En utilisant le type de données set, il est possible de résoudre le problème de champs multiples, comme les emails par exemple.

```

SET : '{' LIST_TERM '}'
LIST_TERM
  : epsilon
  | TERM LIST_TERMAUX
LIST_TERMAUX
  : ',' TERM LIST_TERMAUX
  | epsilon

```

Figure 7: Grammaire SET

- **Listes** : Une liste est utilisée quand l'ordre d'insertion des éléments compte, ou quand on veut pouvoir insérer la même valeur plusieurs fois.

```

LIST : '[' LIST_TERM ']'

```

Figure 8: Grammaire LIST

- **Maps** : Une map permet d'associer deux éléments, sous forme de clef/valeur. Elle peut être utilisée, par exemple, pour sauvegarder les horaires des différents événements dans un profil utilisateur. Chaque élément dans une Map est stocké dans Cassandra comme étant une colonne que vous pouvez modifier, remplacer et requêter.

```

MAP : '{' MAP_CONTENT '}'
MAP_CONTENT
  : TERM_AFF LIST_TERMAFF_AUX
  | epsilon
LIST_TERMAFF_AUX
  : epsilon
  | ',' MAP_CONTENT
TERM_AFF : term ':' term

```

Figure 9: Grammaire MAPS

CHAPITRE III

Analyse

Introduction

Cette partie est très intéressante, en effet c'est la première étape qu'on entame pour réaliser un interpréteur. Elle sert à comprendre les entrées d'une requête et cela en mettant en place un analyseur lexical qui s'occupera d'analyser les éléments lexicaux ou les lexèmes et un analyseur syntaxique pour tous ce qui concerne la forme et la structure de notre requête et à la fin on doit s'assurer qu'il y a un sens et une cohérence dans notre requêtes à travers un analyseur sémantique.

1. Analyseur lexical

1.1. Concepts

Le but de l'analyseur lexical est de reconnaître une séquence de mots appartenant à un langage défini à l'aide d'une expression régulière.

On utilise pour cela FLEX comme un générateur d'analyseur lexical. On part d'une expression régulière de la forme $e_1 | \dots | e_n$ où à chaque e_i est associé un lexème à produire.

Toutefois, l'analyse lexicale ne se contente pas de reconnaître les mots appartenant au langage défini par une certaine expression régulière, elle produit également une suite de lexèmes, un pour chaque mot reconnu, qui sera ensuite utilisée par l'analyseur syntaxique.

Une remarque c'est que la reconnaissance de la séquence de lexème peut être ambiguë. Par exemple, si le langage à reconnaître est b^+ , alors partant de bb on peut soit reconnaître un lexème bb , soit reconnaître une séquence de deux lexèmes b et b .

Dans les générateurs d'analyseurs lexicaux comme *FLEX*, cela est résolu de la façon suivante : on cherche par défaut à reconnaître le plus long préfixe de l'entrée possible, et si deux expressions régulières reconnaissent le même mot, c'est le lexème correspondant à celle écrite en premier qui est choisi.

1.2. Les unités lexicales

Comme vu précédemment, l'analyse lexicale produit des lexèmes ou des unités lexicales, en général au vol à la demande de l'analyseur syntaxique, qui les consomme. Donc l'analyseur demande le prochain lexème à l'analyseur lexical, qui le reconnaît et renvoie un code identifiant ce lexème ou cette classe de lexèmes : identificateurs, nombres . . . en général les lexèmes sont décrits par des expressions régulières.

Par exemples, un identificateur est en général une suite de lettres et chiffres : `[A-Za-z][A-Za-z0-9]*`

Parmi les unités lexicales de CQL qu'on peut citer :

- Identificateur : Le nom des tables, colonnes et keyspaces ...
- Mots clefs déjà réservés :

KEYSPACE	TABLE	TRUNCATE	USE	USING	WHERE
WITH	ASCII	BOOLEAN	COUNT	CREATE	DATE
DECIMAL	DELETE	DESC	DESCRIBE	DISTINCT	DOUBLE
DROP	EXISTS	FLOAT	FROM	FUNCTION	INT
INSERT	KEY	ADD	ALTER	AND	BIGINT
IF	INTO	MAP	MODIFY	NOT	OR

2. Analyseur syntaxique

L'analyseur syntaxique vérifie que l'ordre des tokens correspond à l'ordre définit pour le langage. On dit que l'on vérifie la syntaxe du langage à partir de la définition de sa grammaire. Il produit une représentation sous forme d'arbre de la suite des tokens obtenus lors de l'analyse lexicale.

Avant d'entamer l'analyse syntaxique, la grammaire doit être LL(1) on suivant ses étapes :

- Désambiguïsation de la grammaire.
- Élimination de la récursivité gauche.
- Rendre la grammaire prédictive.

On prend un exemple d'une requête de création de keyspace :

```
CREATE KEYSPACE jobnetwork
  WITH replication = {'class': 'SimpleStratgy', 'replication_factor' : 1};
  AND durable_writes = false;
```

La grammaire qui correspond à cette requête est la suivante :

```

CREATE_KEYSPACE_STMT : alter keyspace idf with PROPERTIES
PROPERTIES : PROPERTY PROPERTIES_AUX
PROPERTIES_AUX
    : and PROPERTIES
    | epsilon
PROPERTY : PROPERTY_NAME '=' PROPERTY_VALUE
PROPERTY_NAME : idf
PROPERTY_VALUE
    : idf
    | CONSTANT
    | MAP
CONSTANT
    : string
    | integer
    | float
    | BOOL
    | uuid
    | blob

```

Alors chaque requête de la création d'une keyspace doit commencer par le mot clefs CREATE suivi d'un autre mot clefs KEYSPACE puis le nom de keyspace qui doit être un identificateur c'est-à-dire correspond au token IDF_TOKEN ensuite vient le mot clefs WITH qui doit être suivi obligatoirement par une liste de propriétés PROPERTIES.

On voit clairement depuis la grammaire que chaque propriété PROPERTY est composée d'un terme gauche PROPERTY_NAME et un terme droite PROPERTY_VALUE.

L'implémentation en langage C de l'analyseur syntaxique se fait à l'aide des ifs imbriqués :

```

boolean _create_keyspace_stmt()
{
    boolean result;
    if(token==CREATE_TOKEN)
    {
        token=_liretoken();
        if(token==KEYSPACE_TOKEN)
        {
            token=_liretoken();
            if(token==If_TOKEN)
            {
                token=_liretoken();
                if(token==EXISTS_TOKEN)
                {
                    token=_liretoken();
                    if(_keyspace_name())
                    {
                        token=_liretoken();
                        if(token==WITH_TOKEN)
                        {
                            token=_liretoken();
                            if(_properties())
                            {
                                result=true;
                            }else
                            {
                                result=false;
                            }
                        }
                    }
                }else
                {
                    result=false;
                }
            }
        }else
        {
            result=false;
        }
    }
}

{
    result=false;
}
}else
{
    result=false;
}
}
}else
{
    result=false;
}
}
return result;
}
}

```

Figure 10: CREATE_KEYSPACE en C

3. Analyseur sémantique

Après avoir passé par l'analyseur syntaxique, Certaines expressions peuvent être syntaxiquement correctes mais ne pas avoir de sens par rapport à la sémantique du langage.

L'analyse sémantique étudie l'arbre de syntaxe abstraite produit par l'analyse syntaxique pour éliminer au maximum les requêtes qui ne sont pas corrects du point de vue de la sémantique. Par exemple, on vérifie l'existence des tables et keyspaces et la cohérence des propriétés.

Lors de la création ou modification d'une keyspace, on doit vérifier premièrement si elle est déjà créer ou pas puis on vérifie les propriétés :

Propriété	Type	Valeur
replication	map	class : 'SimpleStrategy', 'replication_factor' : {int} Rq : replication possède comme valeur des entier
		class: 'NetworkTopologyStrategy', data-centre : {int} , ...
durable_writes	boolean	Soit TRUE ou FALSE

Dans le cas de la propriété replication qui possède comme valeur un Map, si la valeur de class est 'NetworkTopologyStrategy' alors les éléments suivants doivent être obligatoirement une liste des data-centre.

Les options d'une table sont bien énumérer :

Option	Type	Option	Type
comment	<i>string</i>	gc_grace_seconds	<i>int</i>
read_repair_chance	<i>float</i>	bloom_filter_fp_chance	<i>float</i>
dclocal_read_repair_chance	<i>float</i>	default_time_to_live	<i>int</i>
compaction	<i>map</i>	compression	<i>map</i>
caching	<i>map</i>		

Les colonnes d'une table suivent des règles sémantique à savoir les noms de ces colonnes doivent être unique dans la table c'est-à-dire on ne doit pas avoir une duplication de nom de colonne. Le mot clé *static* ne doit être utilisé :

- En même temps sur les colonnes *primary key*
- Si la table utilise option COMPACT STORAGE

- Si la table ne contient pas clustering columns, on doit avoir une seule colonne qui est primary key

Pour insérer des données dans une table, on doit s'assurer que la table existe déjà et on vérifie également les colonnes est ce que ça correspond aux colonnes de cette table.

Donc à partir de ces règles sémantique, on a énumérer toutes les erreurs qu'on peut avoir :

```
//TYPES ERREURS SEMANTIQUES
typedef enum { REPLICATIONERROR_IDf , REPLICATIONERROR_CONST , DURABLE_WRITESERROR_NOTBOOL , INVALIDKEYSPACEPROPERTY , INVALIDKEYSPACECTERM ,
INVALIDKEYSPACEFIRSTTERM , INVALIDKEYSPACESTRATEGY , REPLICATIONFACTOR_NEEDED , INVALIDSIMPLESTRATEGY , INVALID_DATACENTER ,
INVALID_DATACENTER_FACTOR , INVALID_REPLICATION_FACTORVALUE , COMMENTVALUE_NOTCOMPATIBLE , READREPAIRVALUE_NOTCOMPATIBLE ,
DCLOCALREADVALUE_NOTCOMPATIBLE , BLOOMFILTERVALUE_NOTCOMPATIBLE , GCGRACEVALUE_NOTCOMPATIBLE , TIMETOLIVEVALUE_NOTCOMPATIBLE ,
COMPRESSIONVALUE_NOTCOMPATIBLE , LACHINGVALUE_NOTCOMPATIBLE , COMPACTIONVALUE_NOTCOMPATIBLE , INVALIDTABLEOPTIONS , PLUSPRIMARYKEY , STATICWITHPRIMARYKEY ,
UNKNOWNCOLUMN , TABLNAMEFOUND , TABLNAMEINVALID , KEYSACENAMENOTFOUND , TABLENOTFOUND , COLONNENOTFOUND , KEYSACENAMEFOUND , COLUMNNAMEFOUND ,
VIOLATIONCONSTRAINTINTG } OptionsErrorType;
```

Figure 11: Les erreurs sémantiques

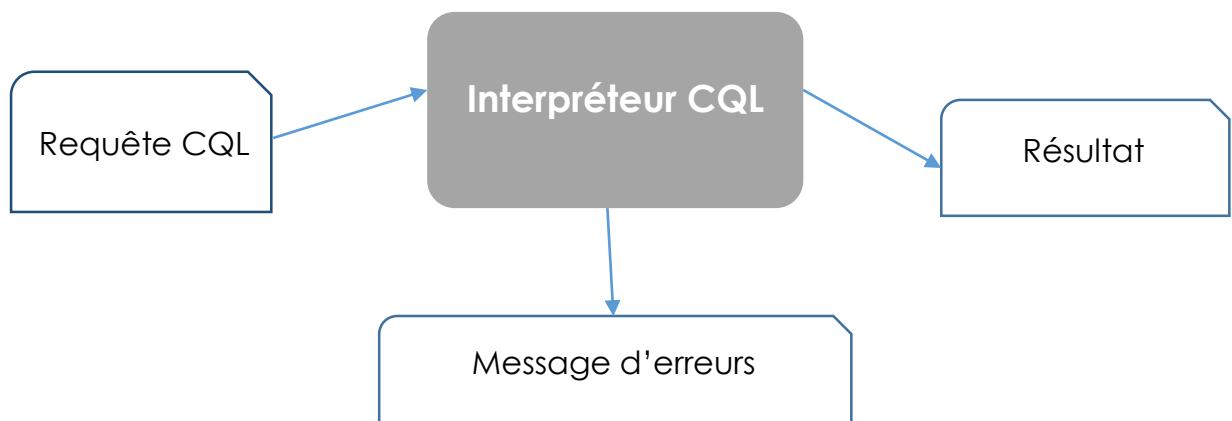
CHAPITRE VI

Interpréteur CQL

Introduction

Après la mise en place d'un analyseur lexical, syntaxique et sémantique, on est assuré que les requêtes sont sans erreurs, respecte la syntaxe et la sémantique de la grammaire CQL, ce qui reste maintenant c'est l'interprétation de ces requêtes et c'est ce qu'on va voir dans cette partie.

Un interpréteur est un logiciel qui prend en entrée un programme dans un langage source et produit en sortie les résultats de l'exécution de ce programme.



1. Représentation intermédiaire

1.1. Dictionnaire de données

Le dictionnaire de données ou métadonnées est une forme de représentation de données qui contient la structure de nos tables et keyspaces, ses champs, clefs, contraintes et tous ce que la base de données aura besoin.

Ce dictionnaire de données est matérialisé par une liste des tables et une autre liste des keyspaces qu'on a met en place et qui seront stockées dans un fichier binaire.

En effet, on stocke pour chaque table son nom, le nom de keyspace dont elle appartient, la liste de ses colonnes et liste des propriétés. Voici ci-dessous la structure utilisée pour le stockage d'une table :

```

/*=====+*
 *      STRUCTURES  TABLES      *
 *=====+*/
typedef struct tables
{
    char Nom_table[30];
    char Nom_keyspace[30];
    ColonneDefinitions colonneDefs; //liste des colonnes tables
    Properties proprietes;           //liste des proprietes tables
    struct tables *suiv;
}table;

```

Figure 12: Structure table

```

/*=====+*
 *      STRUCTURES  LISTE COLONNE TABLE      *
 *=====+*/
typedef struct listColonnes{
    char nom_colonne[30];
    int type_colonne;
    int valeurStatic;
    int valeurPrimary;
    struct listColonnes * suiv;
} colonneDefinitions;

typedef colonneDefinitions* ColonneDefinitions;

```

Figure 13: Structure Colonnes

Chaque table possède une liste de colonne en précisant pour chaque colonne son nom, le type ainsi si elle est primary key ou pas ou si elle est static. Concernant les keyspaces, voici sa structure de stockage :

```

/*=====+*
 *      STRUCTURES  KEYSPPACES      *
 *=====+*/
typedef struct keyspaces
{
    char Nom_keyspace[30];
    Properties proprietes; //liste des proprietes keyspaces
    struct keyspaces *suiv;
}keyspace;

typedef keyspace* Keyspace;

```

Figure 14: Structure Keyspace

Comme il est mentionné avant, les tables et les keyspaces forment des listes chaînées qu'on les enregistre dans des fichiers binaires.

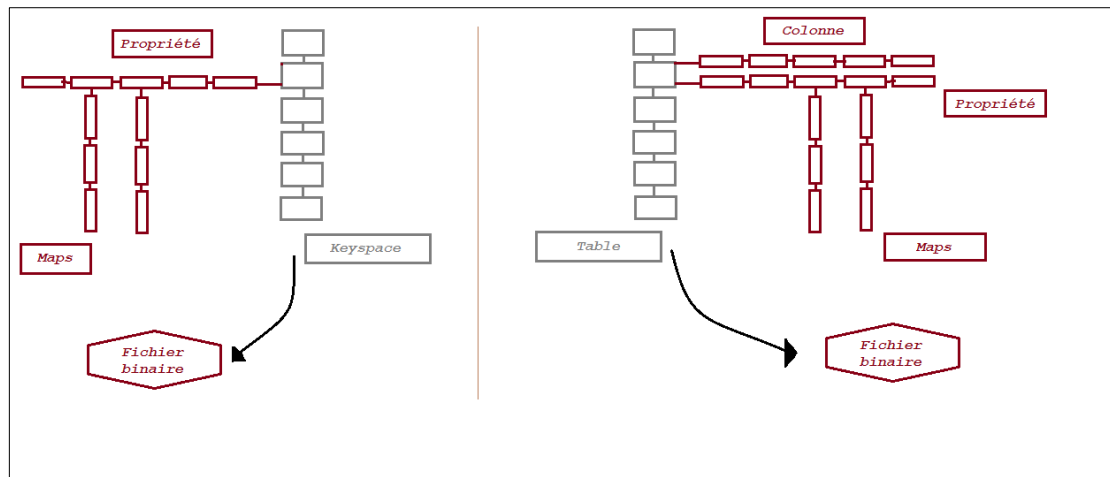


Figure 15: Représentation intermédiaire de dictionnaire de données

1.2. RI des requêtes

Chaque requête saisie doit être stockée quelque part pour qu'elle soit interprétée facilement après.

On a mis en place des structures pour chaque type de requête à savoir :

- Structure RI qui stocke une table pour les requêtes CREATE TABLE et ALTER TABLE.
- Structure RI pour les keyspace lors des requêtes CREATE KEYSPACE et ALTER KEYSPACE.
- Structure RI qui stocke les informations sur une table et la ligne à insérée lors de la requête INSERT INTO.
- Structure RI dont on stocke les lignes récupérées d'une table lors de la requête SELECT.

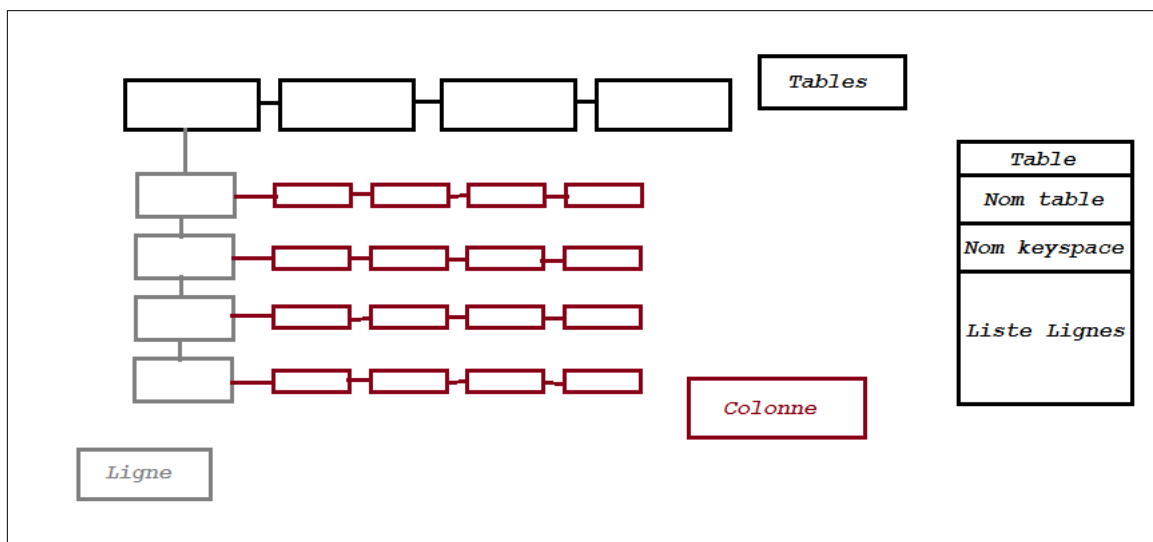


Figure 16: Représentation intermédiaire de lignes des tables

1.3. RI gestionnaires des erreurs

Bien évidemment la détection des erreurs sont parmi les fonctionnalités d'un interpréteur. Pour cela lorsqu'une erreur est détectée, elle doit être stockée dans une liste chaînée qui sera affiché juste après l'analyse de la requête. Les erreurs qui seront stocké sont de type sémantique, voici la structure :

```
//STRUCTURE D'ERREURS SEMANTIQUES
typedef struct {
    char nomErreur[30];
    int linenum;
    OptionsErrorType error;
} optionsError;

//NOMBRE D'ERREUR
extern int nbrErrSm;

//TABLE DES ERREURS
optionsError * tableErr[100];
```

Figure 17: Représentation intermédiaire des erreurs

2. Requête de définition de données

Les requêtes de définition de données sont : CREATE KEYSPACE, ALTER KEYSPACE, DROP KEYSPACE, CREATE TABLE, ALTER TABLE, DROP TABLE ...

La liste des tables et keyspaces c'est-à-dire le dictionnaire de données seront chargées.

Lorsqu'on détecte une requête de création d'une keyspace :

- On vérifie si le keyspace n'existe pas.
- On la stocke dans RI Keyspace et on l'insère dans la liste des keyspace chargée.
- On crée un dossier qui va représenter cette keyspace.

On se qui concerne la création d'une table :

- On vérifie si la table n'existe pas et si éventuellement le keyspace l'appartient.
- On la stocke dans RI table et on l'insère dans la liste des tables déjà chargée.
- Puis on crée un fichier JSON dans le dossier relatif à son keyspace.

3. Requête de manipulation de données

Les requêtes de manipulation de données sont : SELECT, INSERT, UPDATE, DELETE

La liste des tables avec les lignes de données seront chargées automatiquement en parcourant chaque fichier JSON dans chaque dossier.

Pour les requêtes de l'insertion :

- On vérifie si la table existe.
- On récupère la ligne à insérer et on vérifie si elle respecte les contraintes à savoir la clé primaire qui doit être unique ...
- On insère la ligne dans la liste des lignes de la table concerné déjà chargée.
- A la fin de l'exécution, cette liste sera stockée en la parcourant et en sauvegardant la liste des lignes de chaque table dans le fichier JSON qui lui correspond.

On ce qui concerne la sélection des données :

- On récupère la table concerné et on vérifie si elle existe à l'aide de dictionnaire de données déjà chargées.
- On parcourt la liste des tables avec la liste des lignes de chaque table chargées depuis les fichiers JSON.
- On affiche le résultat de la recherche.

4. Test de l'exécution de l'interpréteur CQL

Voici ci-dessous quelque interface de notre interpréteur CQL :

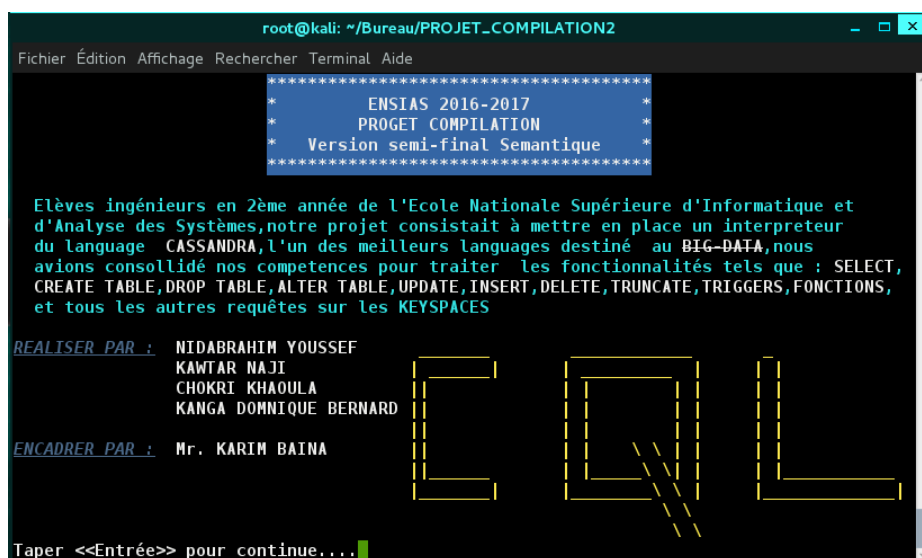
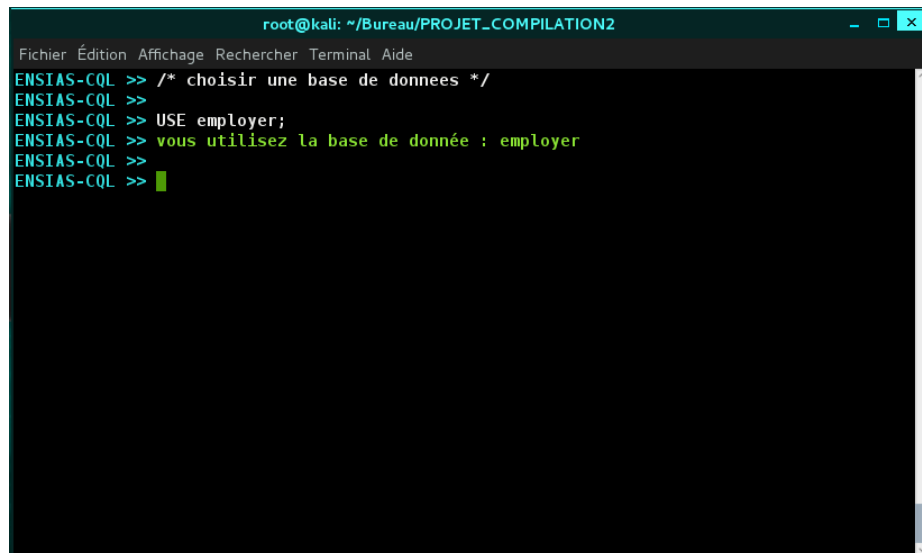
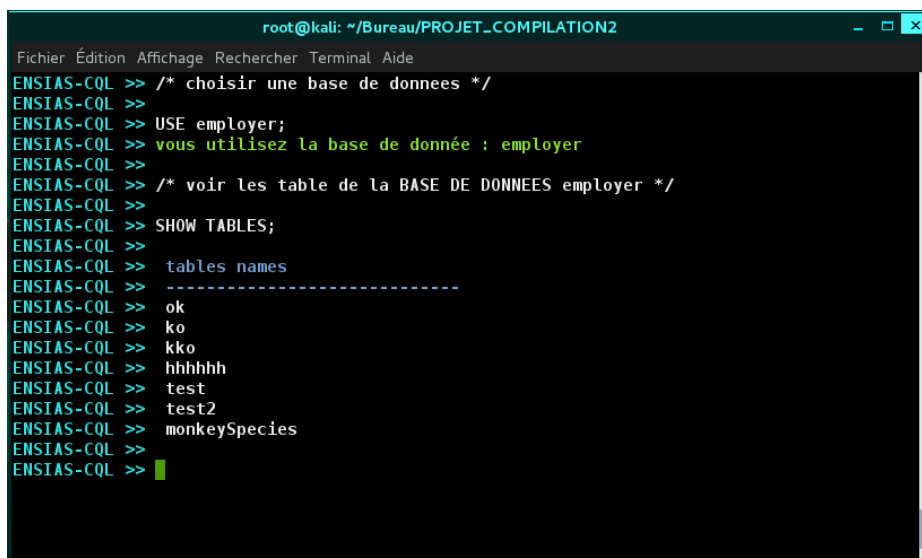


Figure 18: L'accueil



```
root@kali: ~/Bureau/PROJET_COMPILATION2
Fichier Édition Affichage Rechercher Terminal Aide
ENSIAS-CQL >> /* choisir une base de donnees */
ENSIAS-CQL >>
ENSIAS-CQL >> USE employer;
ENSIAS-CQL >> vous utilisez la base de donnée : employer
ENSIAS-CQL >>
ENSIAS-CQL >> █
```

Figure 19: Test - USE



```
root@kali: ~/Bureau/PROJET_COMPILATION2
Fichier Édition Affichage Rechercher Terminal Aide
ENSIAS-CQL >> /* choisir une base de donnees */
ENSIAS-CQL >>
ENSIAS-CQL >> USE employer;
ENSIAS-CQL >> vous utilisez la base de donnée : employer
ENSIAS-CQL >>
ENSIAS-CQL >> /* voir les table de la BASE DE DONNEES employer */
ENSIAS-CQL >>
ENSIAS-CQL >> SHOW TABLES;
ENSIAS-CQL >>
ENSIAS-CQL >> tables names
ENSIAS-CQL >> -----
ENSIAS-CQL >> ok
ENSIAS-CQL >> ko
ENSIAS-CQL >> kko
ENSIAS-CQL >> hhhhhh
ENSIAS-CQL >> test
ENSIAS-CQL >> test2
ENSIAS-CQL >> monkeySpecies
ENSIAS-CQL >>
ENSIAS-CQL >> █
```

Figure 20: Test - SHOW

```
root@kali: ~/Bureau/PROJET_COMPILATION2
Fichier Édition Affichage Rechercher Terminal Aide
ENSIAS-CQL >> /* voir les table de la BASE DE DONNEES employer */
ENSIAS-CQL >>
ENSIAS-CQL >> SHOW TABLES;
ENSIAS-CQL >>
ENSIAS-CQL >>      tables names
ENSIAS-CQL >> -----
ENSIAS-CQL >>      ok
ENSIAS-CQL >>      ko
ENSIAS-CQL >>      kko
ENSIAS-CQL >>      hhhhhh
ENSIAS-CQL >>      test
ENSIAS-CQL >>      test2
ENSIAS-CQL >>      monkeySpecies
ENSIAS-CQL >>
ENSIAS-CQL >> /* voir la description de la TABLE test */
ENSIAS-CQL >> DESCRIB test;
ENSIAS-CQL >>
ENSIAS-CQL >>      column name      type column
ENSIAS-CQL >> -----
ENSIAS-CQL >>      num              INT
ENSIAS-CQL >>      nom              TEXT
ENSIAS-CQL >>
ENSIAS-CQL >>
```

Figure 21: Test - DESCRIBE

```
root@kali: ~/Bureau/PROJET_COMPILATION2
Fichier Édition Affichage Rechercher Terminal Aide
ENSIAS-CQL >>
ENSIAS-CQL >> DESCRIB test;
ENSIAS-CQL >>
ENSIAS-CQL >>      column name      type column
ENSIAS-CQL >> -----
ENSIAS-CQL >>      num              INT
ENSIAS-CQL >>      nom              TEXT
ENSIAS-CQL >>
ENSIAS-CQL >> /* requete select sur la TABLE test */
ENSIAS-CQL >>
ENSIAS-CQL >> SELECT * FROM test;
ENSIAS-CQL >>
ENSIAS-CQL >>      num      nom
ENSIAS-CQL >> -----
ENSIAS-CQL >>      1      ensias
ENSIAS-CQL >>      2      op
ENSIAS-CQL >>      3      inpt
ENSIAS-CQL >>      4      emi
ENSIAS-CQL >>
ENSIAS-CQL >> /* Insertion dans la TABLE test */
ENSIAS-CQL >>
ENSIAS-CQL >> INSERT INTO test(num,nom) VALUES(5,'insea');
ENSIAS-CQL >> ligne insérée
ENSIAS-CQL >>
```

Figure 22: Teste - SELECT

```
root@kali: ~/Bureau/PROJET_COMPILATION2
Fichier Édition Affichage Rechercher Terminal Aide
ENSIAS-CQL >> num      nom
ENSIAS-CQL >> -----
ENSIAS-CQL >> 1      ensias
ENSIAS-CQL >> 2      op
ENSIAS-CQL >> 3      inpt
ENSIAS-CQL >> 4      emi
ENSIAS-CQL >>
ENSIAS-CQL >> /* Insertion dans la TABLE test */
ENSIAS-CQL >>
ENSIAS-CQL >> INSERT INTO test(num,nom) VALUES(5,'insea');
ENSIAS-CQL >> ligne insérée
ENSIAS-CQL >>
ENSIAS-CQL >> /* Verification de l insertion dans la TABLE test */
ENSIAS-CQL >>
ENSIAS-CQL >> SELECT * FROM test;
ENSIAS-CQL >>
ENSIAS-CQL >> num      nom
ENSIAS-CQL >> -----
ENSIAS-CQL >> 1      ensias
ENSIAS-CQL >> 2      op
ENSIAS-CQL >> 3      inpt
ENSIAS-CQL >> 4      emi
ENSIAS-CQL >> 5      insea
ENSIAS-CQL >>
```

Figure 23: Teste - INSERT

CHAPITRE VII

Apport du projet

Ce projet long de 14 semaines est une bonne expérience, il nous a apporté beaucoup de choses tant au niveau technique qu'en terme de gestion de projet.

1. Apports scientifiques et techniques

Ce projet nous a permis de découvrir le monde du développement informatique, un monde complexe mais passionnant de par les débouchés auxquels il mène. Nous avons également acquis de nouvelles notions sur les bases de données NoSQL. Les acquis de ce projet nous serviront dans le futur.

2. Apport sur la formation pédagogique

Ce projet nous a permis de consolider nos connaissances sur le langage C et de découvrir le langage CQL. Nous sommes satisfaits de travailler ce sujet.

3. Apport personnel

Chaque membre de l'équipe a eu des apports personnels durant le projet.

- NIDABRAHIM YOUSSEF :

« Expérience intéressante qui m'a permis de coréaliser un projet abouti et qui m'a appris à être plus à l'écoute des autres. Que du positif, expérience à renouveler »

- KANGA DOMINIQUE BERNARD :

« Bon travail, bonne ambiance de groupe. Que d'émotion et de souvenir inoubliable, très bonne expérience à renouveler »

CONCLUSION

Nous sommes en général satisfaits de tout ce que nous avons appris à travers ce projet.

Conclusion et Perspectives

L'objectif visé à travers ce rapport est de présenter notre solution réalisée qui s'inscrit dans le cadre de notre projet académique de compilation, ce projet consiste à concevoir et à développer un interpréteur CQL Cassandra Query Language.

Au cours de ce travail, on a consacré, dans un premier temps, nos réflexions à l'étude de l'existant et la spécification des besoins. Cette étude nous a permis de déterminer les grands axes qu'on va suivre pour concevoir cette solution. En effet, on a pu développer un interpréteur CQL qui analyse les requêtes lexicalement, syntaxiquement et sémantiquement et qui les interprète.

De l'avis général, nous avons consolidé nos connaissances générales et appris à manipuler et faire des traitements sur des structures plus complexes et gérer les fichiers en langages C. Nous sommes globalement satisfaits de ce que nous avons réalisé.

Au niveau de la gestion du projet en équipe, nous avons réussi à bien nous répartir les tâches afin de réaliser nos objectifs dans les temps et l'ambiance générale du groupe était très bonne. Une bonne expérience à renouveler.