BIG DATA
GESTION DES DONNÉES SEMI-STRUCTURÉES

REF.: 20161017-01 OCTOBRE 2016 - Version 1.0

FICHE DE SYNTHÈSE

De tout temps, l'efficacité du stockage des données a été un objectif majeur poursuivi par la communauté informatique.
Jusqu'à aujourd'hui, les Systèmes de Gestion de Bases de Données Relationnelles (SGBDR) ont constitué la réponse la plus largement répandue. Ces SGBDR ont introduit une modélisation des données sous une forme dite « normale » introduisant un découpage technique de « tables » contenant des « enregistrements » et devant être réconciliées par des « jointures ».
Or, la plupart des applications métiers manipulent des « documents » qui sont des entités autonomes ayant une signification métier.
Par exemple,

- une application de gestion de loterie remet aux clients un ticket, qui est un document ; elle ne délivre pas des enregistrements d'une collection de tables jointurées sur telles ou telles clés.
- un virement bancaire est un document, et pas seulement deux enregistrements dans une table de mouvements, l'un créditant un compte, l'autre le débitant.

Mais les contraintes techniques ont habitué la communauté informatique à raisonner sous cet angle particulier, à « mapper » des objets sur des tables, à développer du code pour automatiser ces mappings, … et à perdre ainsi la simplicité du document.
Aujourd'hui pourtant, les Systèmes d'Information se transforment pour faire face à de nouveaux enjeux digitaux. On voit apparaître les architectures micro-services, dont l'un des principes est d'échanger des documents entre micro-services spécialisés. La notion de document arrive donc en force sur le devant de la scène.
De ce fait, sont apparus de nouveaux besoins de stockage de ces documents, besoins qui sont adressés par diverses solutions :

- Pour la partie transactionnelle : les bases NoSQL spécialisées
- Pour la partie analytique : le Big Data

Il est aujourd'hui impossible d'apporter une réponse unique permettant d'avoir le même niveau de performance pour les accès « transactionnels » et pour les accès « batchs » sur un volume de données important.
Nous avons voulu traiter ici de la gestion de ces documents (composés de données hétérogènes dites « semi-structurées ») par les solutions Big Data.
Cet article vise ainsi à démontrer qu'avec une adaptation du langage SQL (basée sur la norme SQL++), il est possible de gérer efficacement ces documents en termes de :

- Performance (changement d'échelle du temps de réponse)
- Volumétrie (optimisation du stockage de la donnée utile)

Sébastien Barnoud - CTO

FICHE D'ÉVOLUTION

VERSION	DATE	OBJET DE L'ÉVOLUTION	JUSTIFICATIF	VERSION
0.1	17 octobre 2016	Version draft	Version initiale S.Barnoud	17 octobre 2016
1.0	2 novembre 2016	Version 1.0	Après relecture collégiale	2 novembre 2016

SOMMAIRE

1 INTRODUCTION

1.1 AVANT-PROPOS

Cette présentation présume que toutes les notions techniques de base sont connues du lecteur. Néanmoins, pour ceux qui disposent de la version électronique, des liens hypertextes permettent d'en définir ou d'en approfondir certaines.

1.2 RAPPELS ET DÉFINITIONS

1.2.1 DONNÉES SEMI-STRUCTURÉES

Les données semi-structurées sont un type de donnée, qui ne se conforme pas au modèle de structure associé aux bases de données relationnelles.
Elles contiennent des tags (ou d'autres marqueurs) permettant de séparer les éléments sémantiques entre eux et ainsi de hiérarchiser des structures et des champs au sein même de la donnée.
Les données semi-structurées sont aussi connues sous le nom de données auto-décrites.
Exemples avec ou sans schéma :
XML, JSon sont des formats de documents auto-décrits qui peuvent être très « lourds » (notamment en XML), c'est-à-dire, que les tags peuvent représenter plus d'octets que la donnée proprement dite.
Il est possible d'associer à ces formats un schéma : cela revient à décrire l'ensemble des tags possibles, de les typer et d'appliquer des restrictions sur le domaine de valeur de chacun d'entre eux.
La connaissance du schéma n'est toutefois pas obligatoire pour interpréter la donnée. Le schéma permet par contre de la valider.
Afin de limiter le poids des tags, il existe des formats qui permettent d'externaliser la description de la donnée, par exemple Avro ou Protocol Buffer. Pour ces formats, la connaissance du schéma est obligatoire pour décoder un document.

1.2.2 BASES DE DONNÉES RELATIONNELLES

Les bases de données relationnelles contiennent des données structurées : chaque donnée est dans un champ fixe.
Un champ fixe peut contenir une donnée semi-structurée, mais alors le mode de stockage et le langage (en l'occurrence SQL) pour accéder à cette donnée semi-structurée ne sont plus adaptés.

1.2.3 FORME NORMALE

Une forme normale désigne un type de relation particulier entre les entités (données structurées).
Elle sert en particulier à éviter la redondance de l'information, et est très utilisée dans les bases de données relationnelles. Le langage SQL est particulièrement adapté (et même conçu) pour gérer des données qui respectent les formes normales.
La forme normale peut également être utilisée, toujours pour éviter la redondance d'information, dans les documents semi-structurés afin d'en diminuer le poids.

Exemple en finance : Analyse de risques
Une analyse de risques est un ensemble de calculs (agrégats) sur différentes hypothèses de variation de plusieurs paramètres d'un ensemble de contrats.
Un contrat donné intervient donc dans plusieurs résultats d'une même analyse. Le contrat est une structure de données riche qu'on ne veut pas répéter à chaque agrégat. Pour cela, on peut définir une forme normale, c'est-à-dire, modéliser à son niveau un tableau de référence de clés vers le tableau des contrats :

```
struct riskAnalysis {
        array[struct{int id,Struct deal}]  deals,
        array[{struct scenario}] scenario
}

struct scenario {
        double value,
        …
        array[int idref] deals
}
```

1.3 LES ACCÈS TRADITIONNELS À LA DONNÉE

L'utilisation des SGBDR a été jusqu'à ce jour largement répandu pour stocker les données nécessaires au fonctionnement des applications.
Le SQL, langage de prédilection des bases de données relationnelles, permet d'exprimer relativement simplement tout type de requête d'accès à des données structurées.
Si l'on souhaite stocker des informations semi-structurées dans une base de données et y accéder aisément, il est alors nécessaire de construire un modèle de base normalisé (en séparant les entités).
Dans l'exemple précédent, le schéma devra nécessairement contenir :

- La table des deals
- La table des scenarii
- La table modélisant la relation 1:n entre les tables des scénarii et des deals
- Les index permettant d'obtenir des requêtes performantes

L'insertion d'un document dans une base de données avec un tel schéma nécessite alors de mettre à jour de façon atomique ces différentes entités (tables et index), ce qui est résolu par leurs propriétés ACID.

1.4 POURQUOI EN EST-ON ARRIVÉ AUX LIMITES DU SYSTÈME ET COMMENT LES A-T-ON CONTOURNÉES ?

D'après le théorème de CAP, une telle solution basée sur des gestionnaires de base de données relationnelles ne peut pas « scaler » horizontalement, c'est-à-dire ne peut pas résoudre les problèmes d'augmentation de puissance de traitement linéaire uniquement par l'ajout de machines.
Afin de s'affranchir de cette limite, le marché a vu éclore progressivement des bases de données dites « NoSQL » (Not only SQL) qui permettent, entre autres, de stocker les données semi-structurées en tant que document. Le langage de requêtage est adapté pour pouvoir rechercher un document ayant certaines caractéristiques.
Parallèlement, l'architecture des systèmes a considérablement évolué, passant des architectures « 2 tiers » aux architectures micro-services. Pour ces dernières, l'échange d'informations entre les composants utilise massivement des formats semi-structurés (JSon en particulier).
La naissance de ces architectures micro-services a généré le besoin de traitement de données semi-structurées (stockage et requêtage), non seulement au niveau transactionnel mais également au niveau « analytique ».

Le Big Data adresse ces nouveaux besoins de stockage et de « requêtage » analytique, et une de ses implémentations est Hadoop et son écosystème.

2 COMMENT LE BIG DATA GÈRE LES DONNÉES SEMI-STRUCTURÉES ET POURQUOI L'ACCÈS EFFICACE AUX DONNÉES SEMI-STRUCTURÉES N'EST PAS TOTALEMENT RÉSOLU ?

2.1 UN SYSTÈME DE FICHIER POUR LEVER LES LIMITES DU THÉORÈME DE CAP

HDFS (Hadoop Distributed File System) est le système de fichiers, base de l'écosystème Hadoop. Il permet de distribuer massivement des données sur des machines membres d'un même cluster en utilisant leurs disques locaux. Ainsi, HDFS permet de multiplier à l'infini les axes de lecture/écriture.
Associé au framework « MapReduce », les traitements et les données peuvent être colocalisés ; ceci réduit le déplacement de données sur le réseau entre les membres du cluster.
La scalabilité est alors assurée quasiment à l'infini.

2.2 DES FORMATS DE FICHIER POUR OPTIMISER LA PERFORMANCE ET LA VOLUMÉTRIE

Il existe de nombreux formats de fichiers basés sur HDFS, et différents moyens de les écrire/lire

- soit directement par des API
- soit par un moteur SQL.

Pour les données semi-structurées, le plus utilisé est le format SequenceFile, qui

- via un gestionnaire de sérialisation/désérialisation (SerDe) spécifique au format (Avro,JSon, XML, …)
- et/ou via des fonctions définies par l'utilisateur (UDF) (XPath, JsonPath, …),

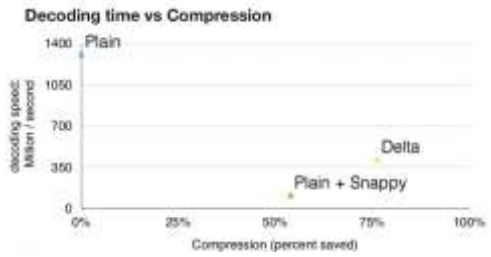
permet d'accéder simplement aux documents.

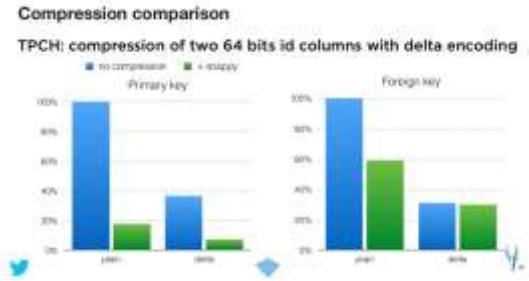
Parmi ces différents formats, Parquet a été spécialement conçu pour stocker, en mode colonne, des données semi-structurées.

Outre un excellent taux de compression (delta encoding, …), Parquet permet le « Predicate Push Down », c'est-à-dire de filtrer, dès le stockage des blocs, les données à lire (ou à ne pas lire), par l'intermédiaire :

- D'une mini structure d'index et de statistiques par blocs de valeurs d'un attribut (min, max)
- D'un dictionnaire (adapté aux faibles cardinalités de valeurs)
- D'un filtre de Bloom (à venir)

Il existe donc de puissants moyens de stockage des données semi-structurées prévus à la fois pour faire des économies tant en termes de stockage (taux de compression atteignant 95% selon les cas d'usages), qu'en termes de puissance CPU pour accéder à la donnée en lecture (voir cette présentation)





2.3 DANS CE NOUVEAU PARADIGME, COMMENT MAINTENANT ACCÈDE-T-ON À L'INFORMATION ?

Dans cet écosystème, HIVE est aujourd'hui le composant qui permet le plus simplement d'accéder à ces informations.
HIVE est un interpréteur du langage HQL qui provient du langage SQL.
HIVE a la possibilité de gérer l'ensemble des formats de fichiers (basés sur HDFS) qui permettent, en fonction de leurs spécificités, de traiter de façon plus au moins efficace les données semi-structurées.
Toutefois, fonctionnellement, le HQL est indépendant du format de fichiers. Par contre, les performances et le taux de compression dépendront fortement du format de fichier utilisé.
HIVE permet également d'accéder à HBase (base NoSQL « clé valeur » standard de l'écosystème Hadoop), SolR (base NoSQL documentaire, non nécessairement dans Hadoop) …

Le HQL (Hive Query Langage) supporte nativement les types « array » et « struct » qui permettent de stocker des données semi-structurées en tant que champs, ainsi que les « LATERAL VIEW » qui, utilisées conjointement avec les UDF inline et explode, permettent de requêter les données semi-structurées dans une syntaxe proche du SQL.
Un exemple de requête utilisant ces éléments syntaxiques est donné au paragraphe **3.2**.

2.4 QU'EN EST-IL DES ACCÈS TRANSACTIONNELS ?

La base NoSQL la plus répandue dans cet écosystème est HBase qui est une base « clé valeur » apte à supporter une charge transactionnelle. HBase permet de stocker des documents semi-structurés, mais est en général directement accédée sous forme d'API par les applications, ce qui veut dire que le traitement de ces documents a lieu dans les applications, dans un langage « évolué », comme Java, C#, … (et pas en SQL).
Il est pourtant possible de faire du SQL sur HBase, via Phoenix. Le problème d'accès aux données semi-structurés est alors le même que pour les bases de données relationnelles classiques.
Le développeur est à nouveau confronté à la difficulté d'accéder aisément et efficacement aux données qu'il souhaite traiter.

3 UNE OUVERTURE POSSIBLE : LE SQL++

3.1 LE SQL++ … MAIS DE QUOI S'AGIT-IL ?

Le SQL++ est issu d'une étude qui propose une évolution sémantique du SQL afin de lui permettre la manipulation des documents semi-structurés.
Ces extensions permettent (entre autres) :

- De « naviguer » dans la structure d'un document (a.b.c)
- De définir une nouvelle notion : la valeur « missing » (à distinguer de la valeur « null »)
- De réaliser des jointures dont la portée est limitée au document

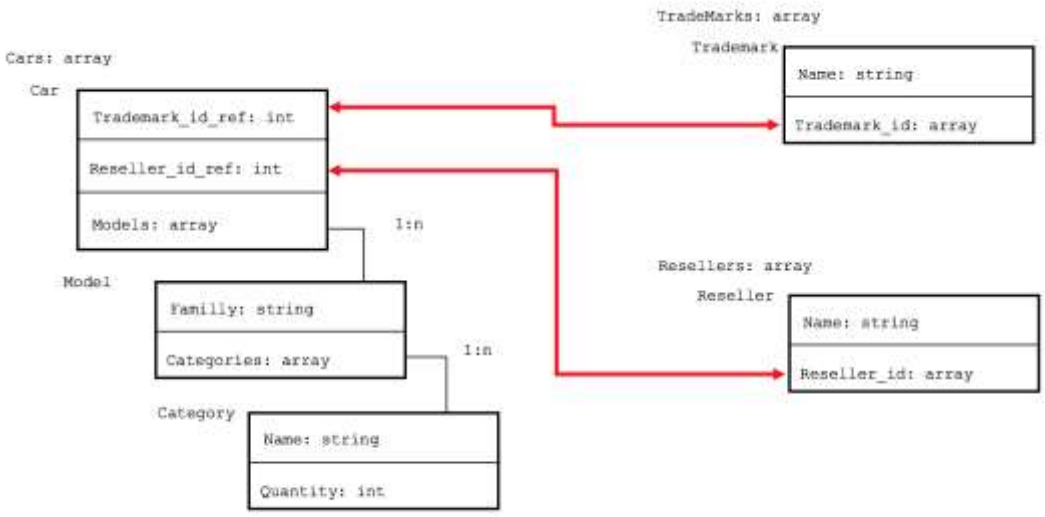
C'est précisément cette dernière fonctionnalité, qui, intégrée à l'optimiseur SQL, peut permettre de bénéficier pleinement des avancées dans le domaine du stockage des données semi-structurées.

3.2 LES LIMITES ACTUELS DU HQL

En effet, le HQL actuel oblige à « dénormaliser » à la volée le « result set », c'est-à-dire à le transformer en vue tabulaire.
Dans le cas des tableaux à grandes dimensions, cette opération est lourde, et ne permet pas de profiter du « predicate pushdown » offert par le stockage (l'évaluation des prédicats n'est alors réalisée que sur la vue tabulaire).
Ainsi, une requête Hive sur un document semi-structuré n'est syntaxiquement absolument pas naturelle pour un utilisateur classique du langage SQL.

3.2.1 SIMPLICITÉ SYNTAXIQUE

A l'inverse, le SQL++ permet une expression beaucoup plus naturelle d'accès aux données.
Exemple :



On peut créer une table Hive pour stocker de tels documents ainsi :

```
create table car (
cars
        array   <struct <
                        trademark_id_ref:int,
                        reseller_id_ref:int,
                        models: array   <struct<
                                                family:string,
                                                category:array<struct<name:string,quantity:int>>
                                                >
                                        >
                        >
                >,
trademarks
        array   <struct<
                        name:string,
                        trademark_id:int
                        >
                >,
resellers
        array   <struct<
                        name:string,
                        reseller_id:int
                        >
                >
)
stored as parquet;
```

Une requête HQL sera :

```
select
        category.quantity as quantity,
        category.name as name,
        models.family as family,
        trademarks.name as trademark,
        resellers.name as reseller
from car
lateral view inline(cars) cars
lateral view inline(cars.models) models
lateral view inline(models.category) category
lateral view inline(trademarks) trademarks
lateral view inline(resellers) resellers
where
        cars.trademark_id_ref = trademarks.trademark_id
        and cars.reseller_id_ref = resellers.reseller_id;
```

La même requête en SQL++ :

```
select
        cars.models.category.quantity as quantity,
        cars.models.category.name as name,
        cars.models.family as family,
        trademarks.name as trademark,
        resellers.name as reseller
from car as c
correlate c.trademarks as trademarks  on c.cars.trademark_id_ref = trademarks.trademark_id
correlate c.resellers as resellers  on c.cars.reseller_id_ref = resellers.reseller_id ;
```

Remarques :
La syntaxe du CORRELATE permet de définir une jointure interne ou externe, droite ou gauche (exactement comme le JOIN classique).
Cette dernière requête est syntaxiquement exactement la même que celle qu'on écrirait avec un SGBDR classique sur une forme normale (sauf que l'on utiliserait JOIN à la place de CORRELATE).

3.2.2 PERFORMANCE

Les moteurs actuels traitent la « lateral view » (qui transforme la forme normale en vue tabulaire) avant les prédicats.
Ceci revient à faire les produits cartésiens de chacun des tableaux, ce qui est particulièrement anti-performant quand il y a un critère de jointure.
De même, s'il y a un prédicat sur l'un des attributs internes, le moteur est incapable de « profiter » du « predicate pushdown » utilisé lors du stockage.
Dans notre exemple la where clause trademarks.name = 'Renault' est syntaxiquement incorrecte car trademarks est un tableau : le développeur SQL est obligé de dénormaliser avec la lateral view avant de l'appliquer.

3.3 UNE SOLUTION : L'EXTENSION DU HQL PAR L'IMPLÉMENTATION DE LA NORME SQL++

L'implémentation du SQL++ et sa bonne intégration dans le moteur, doivent permettre de bénéficier de l'ensemble des algorithmes optimisés de jointure existant (Map Join, Sort Merge Join, …) ainsi que du « predicate pushdown ».

Afin de démontrer le bien-fondé de cette démarche, nous avons développé une fonction UDTF[1] (un type d'UDF qui transforme la vue tabulaire) qui permet de faire la jointure avec un Map Join.

Voici le récapitulatif des temps de réponse obtenus pour différentes valeurs de taille des tableaux. Dans l'ensemble de ces tests N désigne la taille du tableau 'cars', les tableaux 'trademarks' et 'resellers' ont alors une taille N/100.
Les temps mesurés sont en secondes.

N	1 000	10 000	100 000
Lateral view	7	45	62 558
UDTF	6	6	8

1- Il y a 3 types de fonctions définies par l'utilisateur (UDF) :
- Les UDF simples : elles transforment un champ (par exemple le formatage d'une date)
- Les UDAF : elles permettent de faire une agrégation (par exemple max)
- Les UDTF : elles permettent de transformer la vue tabulaire (par exemple explode ou inline)

3.4 EXEMPLES D'APPLICATION

3.4.1 FINANCES

Gains en performance

Dans une grande banque d'investissement, nous avons testé une requête sur des données réelles semi-structurées en Protocol Buffer d'analyse de risques.

La requête, de calcul d'un cube à 35 dimensions, est régulièrement lancée déjà sur un SGBD classique où les données sont stockées sous une forme normale (avec les index adéquats). La requête dure 8 heures.

La requête HQL sur les données semi-structurées stockées en Parquet dure :

- 53 heures avec les lateral view (presque 7 fois moins rapide que le SGBDR)
- 80 secondes avec l'UDTF (360 fois plus rapide que le SGBDR)

Ces résultats montrent que, si l'on veut profiter pleinement des apports du stockage d'objets semi-structurés, il est nécessaire d'intégrer dans l'optimiseur de plan d'exécution la gestion des données semi-structurées.
La sémantique du SQL++ se prête mieux à cette intégration que celle du HQL actuel en apportant de plus une plus grande lisibilité des requêtes.

Remarque : Il est souvent préconisé en architecture Big Data de dénormaliser la donnée quitte à la dupliquer (le coût du stockage sur HDFS étant faible par rapport à celui des SGBDR). Nous n'avons pas évalué les performances de HIVE sur les données dénormalisées. Il nous paraît de toute façon illusoire de penser que la dénormalisation soit l'unique solution à tous les problèmes. Le stockage semi-structuré permet donc, en plus, d'économiser le coût du développement (et de maintenance) de la dénormalisation.

Gains en stockage

De plus le volume de données stockées en Parquet est inférieur de plus de 95% par rapport à celle du SGBD classique (data + index).

3.4.2 GÉNOMIQUE

La génomique pourrait s'avérer être un champ d'application très intéressant.

PROLOGISM

Groupe SI-LOGISM

Tour Gamma B
193 Rue de Bercy
75012 PARIS

Tél. : + 33 (0)1 45 75 63 48
Fax : + 33 (0)1 45 75 63 58

www.prologism.fr

REF.: 20161017-01 OCTOBRE 2016 - Version 1.0