Doctrine Query Language Symfony2

- Retour à la liste des documents
- Doctrine Query Language Symfony2 -

Table des matières

Référence
Types de requêtes DQL
Les requêtes SELECT

- DQL clause SELECT
- Jointure
- Paramètres nommés et positionnels
- Exemples DQL SELECT
- Syntaxe objet partiel
- Utilisation de INDEX BY
- Requêtes UPDATE
- Requêtes DELETE
- Fonctions, opérateurs, Granulats

Interrogation héritées classes

- Tableau simple
- Classe de l'héritage de table

La classe Query

- Formats des résultats des requêtes
- Résultats pures et mixtes
- Comparaison multiple des Entités
- Modes d'hydratation
- Hydratation objet
- Itération Résultat de grands ensembles
- Fonctions

EBNF

- La syntaxe du document:
- Terminals
- Query Language
- Déclarations
- Identificateurs
- Path Expressions
- Clauses
- Items
- From, Join and Index by
- Select Expressions
- Expressions conditionnelles
- Expressions Collection
- Les valeurs littérales
- Paramètre d'entrée
- Expressions arithmétiques
- Expressions scalaires et le type
- Expressions d'agrégat
- Expressions de cas
- D'autres expressions
- Fonctions

Référence

http://docs.doctrine-project.org/projects/doctrine-orm/en/2.1/reference/dql-doctrine-query-language.html

DQL est synonyme de **Doctrine Query Language** et est un dérivé du langage **Object Query Language** qui est très similaire à la **Hibernate Query Language (HQL)** ou le langage **Java Persistence Query (JPQL)**.

En substance, DQL fournit de puissantes capacités d'interrogation sur votre modèle objet. Imaginez tous vos objets qui traînent dans certains lieux de stockage (comme une base de données objet). Lorsque vous écrivez des requêtes DQL, pensez à interroger sur le stockage et à choisir certain sous-ensemble de vos objets.

Une erreur fréquente des débutants est de tromper DQL pour être simplement une certaine forme de SQL, et donc d'essayer d'utiliser les noms de table et les noms de colonnes ou de joindre des tables arbitraires ensemble dans une requête. Vous devez penser à DQL comme un langage de requête pour votre modèle d'objet, pas pour votre schéma relationnel.

DQL n'est pas sensible à la **case**, à l'exception des espaces de noms, de classe et de champs, qui y sont sensibles.

Types de requêtes DQL

DQL est comme un langage de requête pour SELECT, UPDATE et DELETE, il construit la map correspodants aux types instruction SQL. INSERT n'est pas autorisé dans DQL, car les entités et leurs relations doivent être introduites dans le contexte de persistance dans **EntityManager#persist()** pour assurer la cohérence de votre modèle objet.

Les instructions de DQL SELECT sont un moyen très puissant de récupérer des parties de votre modèle de domaine qui ne sont pas accessibles via les associations. En outre, elles permettent de récupérer des entités et leurs associations dans une seule instruction SQL Select qui peuvent faire une énorme différence dans la performance en contraste avec l'aide de plusieurs requêtes.

DQL UPDATE et DELETE offrent un moyen d'exécuter les changements en masse sur les entités de votre modèle de domaine. Cela est souvent nécessaire lorsque vous ne pouvez pas charger toutes les entités concernées d'une mise à jour majeure dans la mémoire.

Les requêtes SELECT

DQL clause SELECT

La clause select d'une requête DQL spécifie ce qui apparaît dans le résultat de la requête. La composition de toutes les expressions dans la clause select influe également sur la nature du résultat de la requête.

Voici un exemple qui sélectionne tous les utilisateurs avec un âge > 20:

```
<?php
$query = $em->createQuery('SELECT u FROM MyProject\Model\User u WHERE u.age > 20');
$users = $query->getResult();
```

Permet d'examiner la requête:

- u est une variable d'identification que l'on appelle ou alias qui fait référence à la classe **MyProject\Model\User**. En plaçant cet alias dans la clause SELECT, nous spécifions que nous voulons toutes les instances de la classe User qui sont appariés par cette requête à apparaître dans le résultat de la requête.
- Le mot-clef FROM est toujours suivi par un nom de classe entièrement qualifié qui à son tour est suivi par une variable d'identification ou un alias pour ce nom de classe. Cette classe désigne une racine de notre requête à partir de laquelle on peut naviguer à travers d'autres jointures (expliqué plus loin) et les expressions de chemin.
- L'expression **u.age** dans la clause WHERE est une expression de chemin. Les expressions de

chemin de DQL sont facilement identifiables par l'utilisation de la '.' l'opérateur qui est utilisé pour construire des chemins. L'expression de chemin u.age se réfère à l'âge sur le terrain de la classe User.

Le résultat de cette requête serait une liste d'objets utilisateur, où tous les utilisateurs sont âgés de plus de 20.

La clause SELECT permet de spécifier les deux variables d'identification de classe que le signal de l'hydratation d'une classe d'entité complète ou juste les champs de l'entité en utilisant les u.name syntaxe. La combinaison des deux est également autorisée et il est possible d'encapsuler les deux champs d'identification et des valeurs dans les fonctions d'agrégation et DQL. Champs numériques peuvent faire partie des calculs en utilisant des opérations mathématiques. Voir la sous-section sur les fonctions DQL, Granulats et des opérations sur le plus d'informations.

Jointure

Une requête SELECT peut contenir des jointures. Il y a 2 types de jointures: «Regular» et "Fetch".

Regular Joins: Permet de limiter les résultats et/ou calculer les valeurs agrégées.

Fetch Joins: En plus de l'utilisation des **Regular Joins**: Utilisé pour récupérer les entités liées et les inclure dans le résultat d'une requête hydratée.

Il n'y a pas de mot-clé spécial DQL qui distingue une jointure régulière d'un fetch. Une jointure (que ce soit une jointure interne ou externe) devient un "**fetch join**" dès que les champs que l'entité a rejoint apparaissent dans la partie SELECT de la requête DQL en dehors d'une fonction d'agrégation. Sinon c'est un "**regular join**".

Exemple:

Regular Joins de l'adresse:

```
<?php
$query = $em->createQuery("SELECT u FROM User u JOIN u.address a WHERE a.city = 'Berlin'");
$users = $query->getResult();
```

Fetch join à l'adresse:

```
<?php
$query = $em->createQuery("SELECT u, a FROM User u JOIN u.address a WHERE a.city = 'Berlin'");
$users = $query->getResult();
```

Lorsque Doctrine hydrates une requête avec **fetch-join**, elle renvoie la classe dans la clause FROM de la racine du tableau de résultat. Dans l'exemple précédent un tableau d'instances utilisateur est renvoyé et l'adresse de chaque utilisateur est récupérée et hydratée dans la variable **User#address**. Si vous accédez à l'adresse, Doctrine n'a pas besoin de réaliser une évaluation retardée avec une autre requête.

Doctrine fonctionne avec toutes les associations des objets de votre de domaine. Les objets qui ne sont pas déjà chargés dans la base sont remplacés par un chargement retardée de vos instances. Des collections non-chargé sont aussi remplacés par une évaluation retardée des cas qui vont chercher tous les objets contenus sur l'accès en premier.

Cependant s'appuyant sur le mécanisme de chargement retardée conduit à de nombreuses petites requêtes exécutées sur la base de données, ce qui peut affecter sensiblement les performances de votre application. Les **Fetch Joins** sont la solution pour hydrater la plupart ou toutes les entités que vous avez besoin dans une seule requête SELECT query.

Paramètres nommés et positionnels

DQL supporte à la fois des paramètres nommés et positionnels, mais contrairement à de nombreux dialectes les paramètres positionnels SQL sont spécifiés avec des nombres, par exemple "?1", "?2" et ainsi de suite. Les paramètres nommés sont spécifiés avec ":nom1", ":nom2" et ainsi de suite.

Lorsque vous référencez les paramètres dans la requête **Query#setParameter($param, $value)**, les deux paramètres nommés et positionnels sont utilisés **sans leurs préfixes**.

Exemples DQL SELECT

Cette section contient un grand nombre de requêtes DQL et quelques explications sur ce qui se passe. Le résultat réel dépend aussi du mode d'hydratation.

Hydratez toutes les entités utilisatrices:

```
<?php
$query = $em->createQuery('SELECT u FROM MyProject\Model\User u');
$users = $query->getResult(); // array of User objects
```

Récupérer les ID de tous CmsUsers:

```
<?php
$query = $em->createQuery('SELECT u.id FROM CmsUser u');
$ids = $query->getResult(); // array of CmsUser ids
```

Récupérer les identifiants de tous les utilisateurs qui ont écrit un article:

```
<?php
$query = $em->createQuery('SELECT DISTINCT u.id FROM CmsArticle a JOIN a.user u');
$ids = $query->getResult(); // array of CmsUser ids
```

Récupérer tous les articles et les trier par le nom des utilisateurs d'articles de l'instance:

```
<?php
$query = $em->createQuery('SELECT a FROM CmsArticle a JOIN a.user u ORDER BY u.name ASC');
$articles = $query->getResult(); // array of CmsArticle objects
```

Récupérer le nom d'utilisateur et le nom d'un CmsUser:

```
<?php
$query = $em->createQuery('SELECT u.username, u.name FROM CmsUser u');
$users = $query->getResults(); // array of CmsUser username and name values
echo $users[0]['username'];
```

Récupérer un ForumUser et son entité unique associé:

```
<?php
$query = $em->createQuery('SELECT u, a FROM ForumUser u JOIN u.avatar a');
$users = $query->getResult(); // array of ForumUser objects with the avatar association loaded
echo get_class($users[0]->getAvatar());
```

Récupérer un CmsUser chercher joindre tous les phoneNumbers il a:

```
<?php
$query = $em->createQuery('SELECT u, p FROM CmsUser u JOIN u.phonenumbers p');
$users = $query->getResult(); // array of CmsUser objects with the phonenumbers association loaded
$phonenumbers = $users[0]->getPhonenumbers();
```

Hydrater un résultat dans l'ordre croissant:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u ORDER BY u.id ASC');
```

```
$users = $query->getResult(); // array of ForumUser objects
```

Ou par ordre décroissant:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u ORDER BY u.id DESC');
$users = $query->getResult(); // array of ForumUser objects
```

L'utilisation de fonctions d'agrégation:

```
<?php
$query = $em->createQuery('SELECT COUNT(u.id) FROM Entities\User u');
$count = $query->getSingleScalarResult();

$query = $em->createQuery('SELECT u, count(g.id) FROM Entities\User u JOIN u.groups g GROUP BY u.id');
$result = $query->getResult();
```

Avec la clause WHERE et une position de paramètre:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u WHERE u.id = ?1');
$query->setParameter(1, 321);
$users = $query->getResult(); // array of ForumUser objects
```

Avec la clause WERE et un nommage de paramètre:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u WHERE u.username = :name');
$query->setParameter('name', 'Bob');
$users = $query->getResult(); // array of ForumUser objects
```

Avec des conditions dans la clause WHERE:

```
<?php
$query = $em->createQuery('SELECT u from ForumUser u WHERE (u.username = :name OR u.username = :name2) AND u.id = :id');
$query->setParameters(array(
    'name' => 'Bob',
    'name2' => 'Alice',
    'id' => 321,
));
$users = $query->getResult(); // array of ForumUser objects
```

Avec COUNT DISTINCT:

```
<?php
$query = $em->createQuery('SELECT COUNT(DISTINCT u.name) FROM CmsUser');
$users = $query->getResult(); // array of ForumUser objects
```

Avec expression arithmétique dans la clause WHERE:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u WHERE ((u.id + 5000) * u.id + 3) < 10000000');
$users = $query->getResult(); // array of ForumUser objects
```

En utilisant un LEFT JOIN pour hydrater tous les identifiants utilisateurs et éventuellement associée article-ID:

```
<?php
$query = $em->createQuery('SELECT u.id, a.id as article_id FROM CmsUser u LEFT JOIN u.articles a');
$results = $query->getResult(); // array of user ids and every article_id for each user
```

Restreindre une clause JOIN par des conditions supplémentaires:

```
<?php
$query = $em->createQuery("SELECT u FROM CmsUser u LEFT JOIN u.articles a WITH a.topic LIKE '%foo%'");
$users = $query->getResult();
```

Utilisation de plusieurs jointures Fetch:

```
<?php
$query = $em->createQuery('SELECT u, a, p, c FROM CmsUser u JOIN u.articles a JOIN u.phonenumbers p JOIN a.comments c');
$users = $query->getResult();
```

BETWEEN dans la clause WHERE:

```
<?php
$query = $em->createQuery('SELECT u.name FROM CmsUser u WHERE u.id BETWEEN ?1 AND ?2');
$query->setParameter(1, 123);
$query->setParameter(2, 321);
$usernames = $query->getResult();
```

Fonctions DQL dans la clause WHERE:

```
<?php
$query = $em->createQuery("SELECT u.name FROM CmsUser u WHERE TRIM(u.name) = 'someone'");
$usernames = $query->getResult();
```

Expression IN():

```
<?php
$query = $em->createQuery('SELECT u.name FROM CmsUser u WHERE u.id IN(46)');
$usernames = $query->getResult();

$query = $em->createQuery('SELECT u FROM CmsUser u WHERE u.id IN (1, 2)');
$users = $query->getResult();

$query = $em->createQuery('SELECT u FROM CmsUser u WHERE u.id NOT IN (1)');
$users = $query->getResult();
```

Fonction DQL CONCAT():

```
<?php
$query = $em->createQuery("SELECT u.id FROM CmsUser u WHERE CONCAT(u.name, 's') = ?1");
$query->setParameter(1, 'Jess');
$ids = $query->getResult();

$query = $em->createQuery('SELECT CONCAT(u.id, u.name) FROM CmsUser u WHERE u.id = ?1');
$query->setParameter(1, 321);
$idUsernames = $query->getResult();
```

EXISTS dans la clause WHERE avec sous-requête corrélée

```
<?php
$query = $em->createQuery('SELECT u.id FROM CmsUser u WHERE EXISTS (SELECT p.phonenumber FROM CmsPhonenumber p WHERE p.user = u.id)');
$ids = $query->getResult();
```

Obtenez tous les utilisateurs qui sont membres du $group.

```
<?php
$query = $em->createQuery('SELECT u.id FROM CmsUser u WHERE :groupId MEMBER OF u.groups');
$query->setParameter('groupId', $group);
```

```
$ids = $query->getResult();
```

Obtenez tous les utilisateurs qui ont plus de 1 numéro de téléphone

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u WHERE SIZE(u.phonenumbers) > 1');
$users = $query->getResult();
```

Obtenez tous les utilisateurs qui n'ont pas de numéro de téléphone

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u WHERE u.phonenumbers IS EMPTY');
$users = $query->getResult();
```

Obtenez toutes les instances d'un type spécifique, pour une utilisation avec des hiérarchies d'héritage:

```
<?php
$query = $em->createQuery('SELECT u FROM Doctrine\Tests\Models\Company\CompanyPerson u WHERE u INSTANCE OF Doctrine\Tests\Models\Company\CompanyEmployee');
$query = $em->createQuery('SELECT u FROM Doctrine\Tests\Models\Company\CompanyPerson u WHERE u INSTANCE OF ?1');
$query = $em->createQuery('SELECT u FROM Doctrine\Tests\Models\Company\CompanyPerson u WHERE u NOT INSTANCE OF ?1');
```

Syntaxe objet partiel

Par défaut, lorsque vous exécutez une requête DQL dans Doctrine et sélectionnez uniquement un sous-ensemble des champs pour une entité donnée, vous ne recevez pas les objets arrières. Au lieu de cela, vous recevez des tableaux uniquement comme un ensemble de résultats plat rectangulaire, semblable à la façon que vous feriez si vous utilisiez SQL directement par la jointure de données.

Si vous voulez sélectionner des objets partiels, vous pouvez utiliser le mot clé DQL **partial** :

```
<?php
$query = $em->createQuery('SELECT partial u.{id, username} FROM CmsUser u');
$users = $query->getResult(); // array of partially loaded CmsUser objects
```

Vous utilisez la syntaxe **partial** lors d'une jointure ainsi:

```
<?php
$query = $em->createQuery('SELECT partial u.{id, username}, partial a.{id, name} FROM CmsUser u JOIN u.articles a');
$users = $query->getResult(); // array of partially loaded CmsUser objects
```

Utilisation de INDEX BY

L'index par construction **INDEX BY** n'est rien de ce qui se traduit directement en SQL, mais qui affecte l'objet et l'hydratation de tableau. Après chaque clause FROM et JOIN vous précisez par quel domaine cette classe devrait être indexé dans les résultats. Par défaut, un résultat est incrémenté par touches numériques commençant à 0. Cependant, avec **INDEX BY** vous pouvez spécifier une autre colonne pour être la clé de votre résultat, il sera réellement fabriqué avec des champs primaire ou unique si:

```
SELECT u.id, u.status, upper(u.name) nameUpper FROM User u INDEX BY u.id
JOIN u.phonenumbers p INDEX BY p.phonenumber
```

Retourne un tableau du genre suivant, indexé à la fois par l'user-ID, puis phonenumber-id:

```
array
  0 =>
    array
      1 =>
        object(stdClass)[299]
          public '__CLASS__' => string 'Doctrine\Tests\Models\CMS\CmsUser' (length=33)
          public 'id' => int 1
          ..
      'nameUpper' => string 'ROMANB' (length=6)
  1 =>
    array
      2 =>
        object(stdClass)[298]
          public '__CLASS__' => string 'Doctrine\Tests\Models\CMS\CmsUser' (length=33)
          public 'id' => int 2
          ...
      'nameUpper' => string 'JWAGE' (length=5)
```

Requêtes UPDATE

DQL permet non seulement de sélectionner vos entités en utilisant les noms de champs, mais aussi d'exécuter les mises à jour en masse sur un ensemble d'entités en utilisant une requête DQL-UPDATE. La syntaxe d'une requête UPDATE fonctionne comme prévu, comme l'exemple suivant:

```
UPDATE MyProject\Model\User u SET u.password = 'new' WHERE u.id IN (1, 2, 3)
```

Les références à des entités apparentées ne sont possibles que dans la clause WHERE et les sous-sélections.

DQL UPDATE est porté directement dans une instruction UPDATE base de données et donc contourne tout système de verrouillage, les événements et n'incrémente la colonne de version. Les entités qui sont déjà chargés dans le contexte de persistance ne vont pas être synchronisé avec l'état de la base de données actualisée. Il est recommandé de faire appel **EntityManager#clear()** et récupérer de nouvelles instances de toute les entités concernées.

Requêtes DELETE

SUPPRIMER les requêtes peuvent également être spécifié en utilisant DQL et leur syntaxe est aussi simple que la syntaxe UPDATE:

```
DELETE MyProject\Model\User u WHERE u.id = 4
```

Les mêmes restrictions s'appliquent pour la référence des entités liées.

Les traitements DQL DELETE sont portés directement dans une base de données DELETE et donc contourne tous les événements et les vérifications pour la colonne de version si elles ne sont pas explicitement ajouté à la clause WHERE de la requête. De plus ils suppriment des entités précises qui ne seront sont pas en cascade à des entités liées, même si cela est spécifiée dans les métadonnées.

Fonctions, opérateurs, Granulats

Fonctions DQL

Les fonctions suivantes sont supportées dans SELECT, WHERE et HAVING:

- ABS(expression_arithmétique)
- CONCAT(str1, str2)
- CURRENT_DATE() - Retourne la date courante
- CURRENT_TIME() - Retourne l'heure courante
- CURRENT_TIMESTAMP() - Retourne un timestamp de la date et l'heure actuelles.
- LENGTH(str) - Retourne la longueur de la chaîne donnée
- LOCATE(needle, haystack [, offset]) - Localiser la première occurrence de la sous-chaîne dans la chaîne.
- LOWER(str) - retourne la chaîne en minuscules.
- MOD(a, b) - Retourne la modulation de b par a.
- SIZE(collection) - Retourne le nombre d'éléments dans la collection spécifiée
- SQRT(q) - Retour de la racine carrée de q.
- SUBSTRING(str, début [, longueur]) - Retourne la sous-chaîne de la chaîne donnée
- TRIM([LEADING \| TRAILING \| BOTH] ['trchar' FROM] str) - Coupez la str, suivant les espaces par défaut.
- UPPER(str) - Retour de la majuscule de la chaîne donnée.
- DATE_ADD(date, jour, l'unité) - Ajouter le nombre de jours à une date donnée.(Unités supportées

sont jour, mois)
- DATE_SUB(date, jour, l'unité) - Soustraire le nombre de jours d'une date donnée.(Unités supportées sont jour, mois)
- DATE_DIFF(date1, date2) - Calculez la différence en jours entre date1 date2-.

Les opérateurs arithmétiques

Vous pouvez faire des maths en utilisant des valeurs numériques avec DQL.

exemple:

```
SELECT person.salary * 1.5 FROM CompanyPerson person WHERE person.salary < 10000
```

Fonctions d'agrégat

Les fonctions d'agrégation suivantes sont autorisés dans SELECT et les clauses GROUP BY: AVG, COUNT, MIN, MAX, SUM

Autres expressions

DQL offre un large éventail d'expressions supplémentaires qui sont connus à partir de SQL, voici une liste de toutes les constructions supportées:

- **ALL/ANY/SOME** - Utilisé dans une clause WHERE suivie d'une sous-sélection, cela fonctionne comme les constructions équivalentes dans SQL.
- **BETWEEN a AND b**, et **NOT BETWEEN a AND b** peuvent être utilisés pour correspondre à des fourchettes de valeurs arithmétiques.
- **IN(x1, x2, ...)** et **NOT IN (x1, x2, ..)** être utilisé pour correspondre à un ensemble de valeurs données.
- **LIKE ..** et **NOT LIKE ..** match en parties d'une chaîne ou un texte en utilisant **"%"** comme un joker.
- **IS NULL** et **IS NOT NULL** pour vérifier avec les valeurs NULL
- **EXISTS** et **NOT EXISTS** en combinaison avec une sous-sélection

Ajout de vos propres fonctions au language DQL

Par défaut DQL est livré avec des fonctions qui font partie d'une large base de données sous-jacentes. Cependant il vous sera probablement choisi une plate-forme de bases de données au début de votre projet et probablement jamais changer. Pour ce cas, vous pouvez facilement étendre le parseur DQL avec ses propres fonctions plate-forme spécialisée.

Vous pouvez vous inscrire sur mesure des fonctions de configuration dans votre DQL ORM:

```
<?php
$config = new \Doctrine\ORM\Configuration();
$config->addCustomStringFunction($name, $class);
$config->addCustomNumericFunction($name, $class);
$config->addCustomDatetimeFunction($name, $class);

$em = EntityManager::create($dbParams, $config);
```

Les fonctions retournent soit une chaîne, valeur numérique ou datetime selon le type de fonction enregistrée. Comme exemple nous allons ajouter une fonctionnalité spécifique à MySQL: **FLOOR()**. Toutes les classes sont données pour mettre en œuvre la classe de base:

```
<?php
namespace MyProject\Query\AST;

use \Doctrine\ORM\Query\AST\Functions\FunctionNode;
use \Doctrine\ORM\Query\Lexer;

class MysqlFloor extends FunctionNode
{
    public $simpleArithmeticExpression;

    public function getSql(\Doctrine\ORM\Query\SqlWalker $sqlWalker)
    {
        return 'FLOOR(' . $sqlWalker->walkSimpleArithmeticExpression(
            $this->simpleArithmeticExpression
        ) . ')';
    }

    public function parse(\Doctrine\ORM\Query\Parser $parser)
    {
        $lexer = $parser->getLexer();

        $parser->match(Lexer::T_IDENTIFIER);
        $parser->match(Lexer::T_OPEN_PARENTHESIS);

        $this->simpleArithmeticExpression = $parser->SimpleArithmeticExpression();

        $parser->match(Lexer::T_CLOSE_PARENTHESIS);
    }
}
```

Nous allons enregistrer la fonction en appelant et peuvent ensuite l'utiliser:

```
<?php
\Doctrine\ORM\Query\Parser::registerNumericFunction('FLOOR', 'MyProject\Query\MysqlFloor');
$dql = "SELECT FLOOR(person.salary * 1.75) FROM CompanyPerson person";
```

Interrogation héritées classes

Cette section montre comment vous pouvez interroger les classes héritées et quel type de résultats à attendre.

Tableau simple

Héritage d'une seule table est une stratégie de mapping d'héritage où toutes les classes d'une hiérarchie sont mappées à une table de base de données unique. Afin de distinguer que ce qui est rangé représente le type dans la hiérarchie d'une colonne discriminante dites est utilisé.

Nous avons d'abord besoin de configurer en exemple un ensemble d'entités à utiliser. Dans ce scénario, il est une personne générique et l'exemple des employés:

```
<?php
namespace Entities;

/**
 * @Entity
 * @InheritanceType("SINGLE_TABLE")
 * @DiscriminatorColumn(name="discr", type="string")
 * @DiscriminatorMap({"person" = "Person", "employee" = "Employee"})
 */
class Person
{
    /**
     * @Id @Column(type="integer")
     * @GeneratedValue
     */
    protected $id;

    /**
     * @Column(type="string", length=50)
     */
    protected $name;

    // ...
}
```

```
/**
 * @Entity
 */
class Employee extends Person
{
    /**
     * @Column(type="string", length=50)
     */
    private $department;

    // ...
}
```

Le SQL généré pour créer les tables pour ces entités ressemble à ceci:

```
CREATE TABLE Person (
    id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    name VARCHAR(50) NOT NULL,
    discr VARCHAR(255) NOT NULL,
    department VARCHAR(50) NOT NULL
)
```

Maintenant, une instance nouvelle **Employee** fixera la valeur discriminante pour nous automatiquement:

```
<?php
$employee = new \Entities\Employee();
$employee->setName('test');
$employee->setDepartment('testing');
$em->persist($employee);
$em->flush();
```

Maintenant passons une requête simple pour récupérer l'employé que nous venons de créer:

```
SELECT e FROM Entities\Employee e WHERE e.name = 'test'
```

Si nous vérifions le SQL généré, vous remarquerez qu'il a quelques conditions particulières ajoutées pour s'assurer de ne faire revenir que les entités Employé:

```
SELECT p0_.id AS id0, p0_.name AS name1, p0_.department AS department2,
       p0_.discr AS discr3 FROM Person p0_
WHERE (p0_.name = ?) AND p0_.discr IN ('employee')
```

Classe de l'héritage de table

La classe de l'héritage de table est une stratégie de mapping d'héritage où chaque classe dans une hiérarchie est mappé à plusieurs tables: sa propre table et les tables de toutes les classes parentes. Le tableau d'une classe de l'enfant est liée à la table d'une classe parent grâce à une contrainte de clé étrangère. Doctrine2 implémente cette stratégie à travers l'utilisation d'une colonne discriminante dans la table le plus élevé de la hiérarchie parce que c'est la meilleure façon d'atteindre des requêtes polymorphiques de la classe de l'héritage de table.

L'exemple d'héritage de table est le même que pour une seule table, vous avez juste besoin de changer le type d'héritage de SINGLE_TABLE par JOINED

```
<?php
/**
 * @Entity
 * @InheritanceType("JOINED")
 * @DiscriminatorColumn(name="discr", type="string")
 * @DiscriminatorMap({"person" = "Person", "employee" = "Employee"})
 */
class Person
{
    // ...
}
```

Maintenant jetez un oeil à la requête SQL qui est généré pour créer la table, vous remarquerez quelques différences:

```
CREATE TABLE Person (
    id INT AUTO_INCREMENT NOT NULL,
    name VARCHAR(50) NOT NULL,
    discr VARCHAR(255) NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE Employee (
    id INT NOT NULL,
    department VARCHAR(50) NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Employee ADD FOREIGN KEY (id) REFERENCES Person(id) ON DELETE CASCADE
```

- Les données sont réparties entre deux tables
- Une clé étrangère existe entre les deux tableaux

Maintenant, si deviez insérer le même **Employee** comme nous l'avons fait dans l'exemple **SINGLE_TABLE** et exécuter la même requête en exemple il va générer des SQL différents joindra les inforamtions de **Person** automatiquement pour vous:

```
SELECT p0_.id AS id0, p0_.name AS name1, e1_.department AS department2,
       p0_.discr AS discr3
FROM Employee e1_ INNER JOIN Person p0_ ON e1_.id = p0_.id
WHERE p0_.name = ?
```

La classe Query

Une instance de la classe **Doctrine\ORM\Query** représente une requête DQL. Vous créez une instance de requêtes en appelant **EntityManager#createQuery($dql)**, en passant la chaîne de requête DQL. Sinon, vous pouvez créer une instance de requête vide **Query** et invoquer la requête **Query#setDql($dql)** par la suite.

Exemples

```
<?php
// $em instanceof EntityManager

// example1: passing a DQL string
$q = $em->createQuery('select u from MyProject\Model\User u');

// example2: using setDql
$q = $em->createQuery();
$q->setDql('select u from MyProject\Model\User u');
```

Formats des résultats des requêtes

Le format dans lequel le résultat d'une requête DQL SELECT est retournée peut être influencée par un mode hydratation dite. Un mode hydratation spécifie la manière particulière dans laquelle un jeu de résultats SQL est transformée. Chaque mode d'hydratation a sa propre méthode dédiée à la classe de requêtes. Ici, ils sont:

- **Query#getResult()**: Récupère une collection d'objets. Le résultat est soit une collection d'objets simples (pure) ou un tableau où les objets sont emboîtés dans les lignes de résultat (mixte).
- **Query#getSingleResult()**: Récupère un objet unique. Si le résultat contient plus d'un ou aucun objet, une exception est levée. La distinction pure/mixte ne s'applique pas.
- **Query#getOneOrNullResult()**: retourne un objet unique. Si aucun objet n'est trouvé nulle sera retournée.
- **Query#getArrayResult()**: Récupère un graphe de tableau (un tableau imbriqué) qui est largement interchangeables avec le graphe d'objet généré par la **Query#getResult()** pour la lecture seule des fins.
    - Un graphe tableau peut varier à partir du graphe d'objet correspondante dans certains

scénarios à cause de la différence de la sémantique identité entre les tableaux et les objets.
- **Query#getScalarResult()**: Récupère un résultat plat/rectangulaires ensemble de valeurs scalaires qui peuvent contenir des données en double. La distinction pure/mixte ne s'applique pas.
- **Query#getSingleScalarResult()**: Récupère une valeur scalaire unique du résultat retourné par le SGBD. Si le résultat contient plus d'une valeur scalaire unique, une exception est levée. La distinction pure/mixte ne s'applique pas.

Au lieu d'utiliser ces méthodes, vous pouvez également utiliser la méthode de requête d'usage général **Query#execute(array $params = array(), $hydrationMode = Query::HYDRATE_OBJECT)**. En utilisant cette méthode, vous pouvez directement mentionné le mode d'alimentation hydratation comme second paramètre via l'une des constantes de requêtes. En fait, les méthodes mentionnées précédemment ne sont que des raccourcis commodes pour la méthode execute. Par exemple, la méthode **Query#getResult()** n'appelle en interne que le mode d'hydratation pour s'exécuter, en passant par la Query::HYDRATE_OBJECT.

L'utilisation des méthodes mentionnées plus haut sont généralement préférées car elle conduit à un code plus concis.

Résultats pures et mixtes

La nature d'un résultat renvoyé par une requête SELECT DQL récupérées par le biais de **Query#getResult()** ou **Query#getArrayResult()** peut être de 2 formes: pures et mixtes. Dans les exemples précédents simples, vous avez déjà vu un résultat «pure» de requête, avec des objets seulement. Par défaut, le type de résultat est pur, mais dès que des valeurs scalaires, tels que les valeurs agrégées ou des valeurs scalaires d'autres qui n'appartiennent pas à une entité, apparaissent dans la partie SELECT de la requête DQL, le résultat devient mélangé. Un résultat mitigé a une structure différente que le résultat pur afin de tenir compte des valeurs scalaires.

Un résultat pur ressemble généralement à ceci:

```
$dql = "SELECT u FROM User u";

array
    [0] => Object
    [1] => Object
    [2] => Object
    ...
```

Un résultat mitigé a la structure générale suivante:

```
$dql = "SELECT u, 'some scalar string', count(u.groups) AS num FROM User u JOIN u.groups g GROUP BY u.id";

array
    [0]
        [0] = > Object
        [1] = > "some scalar string"
        ['num'] = > 42
        // ... more scalar values, either indexed numerically or with a name
    [1]
        [0] = > Object
        [1] = > "some scalar string"
        ['num'] = > 42
        // ... more scalar values, either indexed numerically or with a name
```

Afin de mieux comprendre des résultats mitigés, considérer la requête suivante DQL:

```
SELECT u, UPPER(u.name) nameUpper FROM MyProject\Model\User u
```

Cette requête fait usage de la fonction UPPER DQL qui retourne une valeur scalaire et parce qu'il y a maintenant une valeur scalaire dans la clause SELECT, on obtient un résultat mitigé.

Conventions pour des résultats mitigés sont comme suit:

- L'objet récupéré dans la clause FROM est toujours positionné à la clé '0'.
- Chaque scalaire sans nom est numérotée dans l'ordre donné dans la requête, en commençant par 1.
- Chaque scalaire alias est donné avec son nom-alias comme la clé. La **case** de la dénomination est maintenue.
- Si plusieurs objets sont extraites de la clause FROM ils alternent chaque ligne.

Voici comment le résultat peut ressembler :

```
array
    array
        [0] = > User (Object)
        ['nameUpper'] = > "ROMAN"
    array
        [0] = > User (Object)
        ['nameUpper'] = > "JONATHAN"
    ...
```

Et voici comment vous y accéder dans le code PHP:

```
<?php
foreach ($results as $row) {
    echo "Name: " . $row[0]->getName();
    echo "Name UPPER: " . $row['nameUpper'];
}
```

Comparaison multiple des Entités

Si vous allez chercher de multiples entités qui sont énumérés dans la clause FROM, l'hydratation sera de retour dans les lignes itérations des différentes entitées de haut niveau.

```
$dql = "SELECT u, g FROM User u, Group g";

array
    [0] = > Object (User)
    [1] = > Object (Group)
    [2] = > Object (User)
    [3] = > Object (Group)
```

Modes d'hydratation

Chacun des modes d'hydratation fait des hypothèses sur la façon dont le résultat est retourné à l'utilisateur. Vous devez connaître tous les détails pour faire le meilleur usage des formats de résultats différents:

Les constantes pour les modes d'hydratation différents sont les suivants:

```
- Query::HYDRATE_OBJECT
- Query::HYDRATE_ARRAY
- Query::HYDRATE_SCALAR
- Query::HYDRATE_SINGLE_SCALAR
```

Hydratation objet

Objets hydratés, le résultats dans le graphe d'objets. d'objet:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u');
$users = $query->getResult(Query::HYDRATE_OBJECT);
```

Hydratation tableau

Vous pouvez exécuter la même requête à l'hydratation de tableau et le jeu de résultats est hydratée dans un tableau qui représente le graphe d'objets:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u');
$users = $query->getResult(Query::HYDRATE_ARRAY);
```

Vous pouvez utiliser le raccourci **getArrayResult()** ainsi:

```
<?php
$users = $query->getArrayResult();
```

Hydratation Scalaire

Si vous voulez retourner un résultat plat rectangulaire au lieu de définir un objet graphique, vous pouvez utiliser l'hydratation scalaire:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u');
$users = $query->getResult(Query::HYDRATE_SCALAR);
echo $users[0]['u_id'];
```

Les hypothèses suivantes ont été faites sur les champs sélectionnés à l'aide d'hydratation scalaire:

Les champs de classes sont préfixés par l'alias DQL dans le résultat. Une requête de 'SELECT u.name ..' une clef 'u_name' dans les lignes de résultat.

Hydratation scalaire unique

Si vous une requête qui renvoie juste une valeur scalaire unique, vous pouvez utiliser l'hydratation scalaire unique:

```
<?php
$query = $em->createQuery('SELECT COUNT(a.id) FROM CmsUser u LEFT JOIN u.articles a WHERE u.username = ?1 GROUP BY u.id');
$query->setParameter(1, 'jwage');
$numArticles = $query->getResult(Query::HYDRATE_SINGLE_SCALAR);
```

Vous pouvez utiliser le raccourci getSingleScalarResult() ainsi:

```
<?php
$numArticles = $query->getSingleScalarResult();
```

Modes d'hydratation sur mesure

Vous pouvez facilement ajouter vos propres modes d'hydratation sur mesure en créant d'abord une classe avec l'extension **AbstractHydrator**:

```
<?php
namespace MyProject\Hydrators;

use Doctrine\ORM\Internal\Hydration\AbstractHydrator;

class CustomHydrator extends AbstractHydrator
{
    protected function _hydrateAll()
    {
        return $this->_stmt->fetchAll(PDO::FETCH_ASSOC);
    }
}
```

Ensuite, vous avez juste besoin d'ajouter la classe à la configuration ORM:

```
<?php
$em->getConfiguration()->addCustomHydrationMode('CustomHydrator', 'MyProject\Hydrators\CustomHydrator');
```

Maintenant l'hydrateur est prêt à être utilisé dans vos requêtes:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u');
$results = $query->getResult('CustomHydrator');
```

Itération Résultat de grands ensembles

Il y a des situations où une requête que vous voulez exécuter renvoie un très grand ensemble de résultats qui doivent être traitée. Tous les modes d'hydratation précédemment décrites chargent complètement un jeu de résultats dans la mémoire, qui pourrait ne pas être faisable avec de grands résultats. Voir la section de traitement batch sur les détails comment parcourir de grands résultats.

Fonctions

Les méthodes suivantes sont réunies sur **AbstractQuery** qui utilise à la fois l'extensiion de **Query** et **NativeQuery**.

Paramètres

Déclarations préparées qui utilisent des caractères génériques numériques ou nommé nécessitent des paramètres supplémentaires pour les rendre exécutable pour la base de données. Pour passer des paramètres à la requête des méthodes suivantes peuvent être utilisées:

- **AbstractQuery::setParameter($param, $valeur)** - Setter le numérique ou la wildcard à une valeur donnée.
- **AbstractQuery::setParameters(array $params)** - Définir un tableau de paramètres paires clé-valeur.
- **AbstractQuery::getParameter($param)**
- **AbstractQuery::getParameters()**

Les deux paramètres nommés et positionnels sont transmises à ces méthodes sans leur? ou: le préfixe.

Cache liée API

Vous pouvez mettre en cache les résultats des requêtes basées soit sur toutes les variables qui définissent le résultat (SQL, hydratation Mode, Paramètres et astuces) ou sur les touches de cache définis par l'utilisateur. Cependant tous les résultats de requête par défaut ne sont pas mis en cache. Vous devez activer le cache de résultat sur une base par requête. L'exemple suivant montre un flux de travail complet en utilisant le cache Résultat API:

```
<?php
$query = $em->createQuery('SELECT u FROM MyProject\Model\User u WHERE u.id = ?1');
$query->setParameter(1, 12);

$query->setResultCacheDriver(new ApcCache());

$query->useResultCache(true)
      ->setResultCacheLifeTime($seconds = 3600);

$result = $query->getResult(); // cache miss

$query->expireResultCache(true);
$result = $query->getResult(); // forced expire, cache miss

$query->setResultCacheId('my_query_result');
$result = $query->getResult(); // saved in given result cache id.

// or call useResultCache() with all parameters:
$query->useResultCache(true, $seconds = 3600, 'my_query_result');
$result = $query->getResult(); // cache hit!

// Introspection
$queryCacheProfile = $query->getQueryCacheProfile();
$cacheDriver = $query->getResultCacheDriver();
$lifetime = $query->getLifetime();
$key = $query->getCacheKey();
```

Vous pouvez setter le gestionnaire de cache globalement sur la Doctrine Doctrine\ORM\Configuration de sorte qu'il soit passé à chaque requête et de l'instance NativeQuery.

Indicateurs de requête

Vous pouvez passer des notes à l'analyseur de requête et hydratants en utilisant la méthode **AbstractQuery::setHint($name, $value)**. Actuellement, il existe des indicateurs de requête pour la plupart internes qui ne sont pas être consommés en userland. Toutefois, les quelques conseils suivants doivent être utilisés en userland:

- **Query::HINT_FORCE_PARTIAL_LOAD** - Permet aux objets d'hydrate mais pas toutes leurs colonnes sont récupérées. Cet indicateur de requête peut être utilisé pour traiter les problèmes de consommation de mémoire avec un grand résultat des jeux qui contiennent des données de type char ou binaire. Doctrine n'a aucun moyen de rechargement implicitement ces données. Objets partiellement chargés doivent être transmis à **EntityManager::refresh()** si elles doivent être rechargées complètement à partir de la base de données.
- **Query::HINT_REFRESH** - Cette requête est utilisée en interne par **EntityManager::refresh()** et peut être utilisé en userland ainsi. Si vous spécifiez cette indication, et qu'une requête renvoie les données pour une entité qui est déjà géré par le UnitOfWork, les champs de l'entité existante seront rafraîchis. En fonctionnement normal un jeu de résultats qui charge des données d'une entité déjà existante est abandonné au profit de l'entité déjà existante
- **Query::HINT_CUSTOM_TREE_WALKERS** - Un tableau de la doctrine supplémentaires **Doctrine\ORM\Query\TreeWalker** qui sont attachés au processus de l'analyse des requêtes DQL.

Cache de requêtes (Query DQL seulement)

L'analyse d'une requête DQL et la convertir en une requête SQL sur la plate-forme sous-jacente de base de données a évidemment un coût, contrairement à exécuter directement des requêtes SQL natives. C'est pourquoi il ya un cache de requêtes dédié à la mise en cache des résultats analyseur DQL. En combinaison avec l'utilisation de jokers, vous pouvez réduire le nombre de requêtes analysées dans la production à zéro.

Le pilote du cache de requêtes est passé de **Doctrine\ORM\Configuration** à chaque requête pour une instance de **Doctrine\ORM\Configuration** par défaut et est également activée par défaut. Cela signifie également que vous n'avez pas besoin de jouer régulièrement avec les paramètres du cache de requêtes, mais si vous le faites il y a plusieurs méthodes pour interagir avec lui:

- Query::setQueryCacheDriver($driver) - Permet de définir une instance de cache
- Query::setQueryCacheLifeTime($secondes = 3600) - Durée de vie Jeu de la mise en cache de requête.
- Query::expireQueryCache($bool) - Appliquer l'expiration de la cache de requêtes si la valeur true.
- Query::getExpireQueryCache()
- Query::getQueryCacheDriver()
- Query::getQueryCacheLifeTime()

Articles premier résultat et Max (requête DQL seulement)

Vous pouvez limiter le nombre de résultats retournés par une requête DQL ainsi que de spécifier l'offset de départ, la doctrine utilise ensuite une stratégie de manipulation de la requête de sélection pour retourner uniquement le nombre requis de résultats:

- **Query::setMaxResults($maxResults)**
- **Query::setFirstResult($offset)**

Si votre requête contient une collection de **jointures fetch** en précisant les méthodes limites, les résultat ne fonctionnent pas comme on peut s'y attendre. Setter **Max Results** restreint le nombre de lignes de la résultat de base de données, mais si c'est le cas d'aller chercher les collections, une entité racine pourraient apparaître dans de nombreuses lignes, hydratant de manière efficace un nombre inférieur au nombre spécifié de résultats.

Changer temporairement le mode de récupération en DQL

Alors que normalement toutes vos associations sont marquées comme évaluation retardée ou extra retardée, vous aurez le cas où vous utilisez DQL et ne voudrait pas chercher à joindre un deuxième, troisième ou quatrième niveau d'entités dans votre résultat, en raison de l'augmentation du coût de l'instruction SQL JOIN. Vous pouvez marquer une association many-to-one ou one-to-one comme tiré par les cheveux temporairement pour le chargement temporaire par batch de ces entités en utilisant une requête WHERE .. IN.

```
<?php
$query = $em->createQuery("SELECT u FROM MyProject\User u");
$query->setFetchMode("MyProject\User", "address", "EAGER");
$query->execute();
```

Étant donné qu'il ya 10 utilisateurs et des adresses correspondantes dans la base de données des requêtes exécutées ressemblera à quelque chose comme:

```
SELECT * FROM users;
SELECT * FROM address WHERE id IN (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

EBNF

Ce qui suit est hors-contexte de la syntaxe habituelle, écrit dans une variante EBNF, décrit la Doctrine Query Language. Vous pouvez consulter cette syntaxe quand vous n'êtes pas sûr de ce qui est possible avec ce DQL ou la syntaxe correcte d'une requête particulière.

La syntaxe du document:

- non-terminaux commencer par une lettre majuscule
- terminals commencer par un caractère en minuscule
- entre parenthèses(...) sont utilisés pour le regroupement
- entre crochets [...] sont utilisés pour définir une partie optionnelle, par exemple, zéro ou une fois
- accolades {...} sont utilisés pour la répétition, par exemple, zéro ou plusieurs fois
- guillemets "..." définir une chaîne terminale d'une barre verticale \| représente une alternative

Terminals

- identifier (name, email, ...)
- string ('foo', 'bar''s house', '%ninja%', ...)
- char ('/', '\', ' ', ...)
- integer (-1, 0, 1, 34, ...)
- float (-0.23, 0.007, 1.245342E+8, ...)
- boolean (false, true)

Query Language

```
QueryLanguage ::= SelectStatement | UpdateStatement | DeleteStatement
```

Déclarations

```
SelectStatement ::= SelectClause FromClause [WhereClause] [GroupByClause] [HavingClause] [OrderByClause]
UpdateStatement ::= UpdateClause [WhereClause]
DeleteStatement ::= DeleteClause [WhereClause]
```

Identificateurs

```
/* Alias Identification usage (the "u" of "u.name") */
IdentificationVariable ::= identifier

/* Alias Identification declaration (the "u" of "FROM User u") */
AliasIdentificationVariable :: = identifier

/* identifier that must be a class name (the "User" of "FROM User u") */
AbstractSchemaName ::= identifier
```

```
/* identifier that must be a field (the "name" of "u.name") */
/* This is responsible to know if the field exists in Object, no matter if it's a relation or a simple field */
FieldIdentificationVariable ::= identifier

/* identifier that must be a collection-valued association field (to-many) (the "Phonenumbers" of "u.Phonenumbers") */
CollectionValuedAssociationField ::= FieldIdentificationVariable

/* identifier that must be a single-valued association field (to-one) (the "Group" of "u.Group") */
SingleValuedAssociationField ::= FieldIdentificationVariable

/* identifier that must be an embedded class state field (for the future) */
EmbeddedClassStateField ::= FieldIdentificationVariable

/* identifier that must be a simple state field (name, email, ...) (the "name" of "u.name") */
/* The difference between this and FieldIdentificationVariable is only semantical, because it points to a single field (not mapping to a relation) */
SimpleStateField ::= FieldIdentificationVariable

/* Alias ResultVariable declaration (the "total" of "COUNT(*) AS total") */
AliasResultVariable = identifier

/* ResultVariable identifier usage of mapped field aliases (the "total" of "COUNT(*) AS total") */
ResultVariable = identifier
```

Path Expressions

```
/* "u.Group" or "u.Phonenumbers" declarations */
JoinAssociationPathExpression             ::= IdentificationVariable "." (CollectionValuedAssociationField | SingleValuedAssociationField)

/* "u.Group" or "u.Phonenumbers" usages */
AssociationPathExpression                 ::= CollectionValuedPathExpression | SingleValuedAssociationPathExpression

/* "u.name" or "u.Group" */
SingleValuedPathExpression                ::= StateFieldPathExpression | SingleValuedAssociationPathExpression

/* "u.name" or "u.Group.name" */
StateFieldPathExpression                  ::= IdentificationVariable "." StateField | SingleValuedAssociationPathExpression "." StateField

/* "u.Group" */
SingleValuedAssociationPathExpression     ::= IdentificationVariable "." SingleValuedAssociationField

/* "u.Group.Permissions" */
CollectionValuedPathExpression            ::= IdentificationVariable "." {SingleValuedAssociationField "."}* CollectionValuedAssociationField

/* "name" */
StateField                                ::= {EmbeddedClassStateField "."}* SimpleStateField

/* "u.name" or "u.address.zip" (address = EmbeddedClassStateField) */
SimpleStateFieldPathExpression            ::= IdentificationVariable "." StateField
```

Clauses

```
SelectClause        ::= "SELECT" ["DISTINCT"] SelectExpression {"," SelectExpression}*
SimpleSelectClause  ::= "SELECT" ["DISTINCT"] SimpleSelectExpression
UpdateClause        ::= "UPDATE" AbstractSchemaName ["AS"] AliasIdentificationVariable "SET" UpdateItem {"," UpdateItem}*
DeleteClause        ::= "DELETE" ["FROM"] AbstractSchemaName ["AS"] AliasIdentificationVariable
FromClause          ::= "FROM" IdentificationVariableDeclaration {"," IdentificationVariableDeclaration}*
SubselectFromClause ::= "FROM" SubselectIdentificationVariableDeclaration {"," SubselectIdentificationVariableDeclaration}*
WhereClause         ::= "WHERE" ConditionalExpression
HavingClause        ::= "HAVING" ConditionalExpression
GroupByClause       ::= "GROUP" "BY" GroupByItem {"," GroupByItem}*
OrderByClause       ::= "ORDER" "BY" OrderByItem {"," OrderByItem}*
Subselect           ::= SimpleSelectClause SubselectFromClause [WhereClause] [GroupByClause] [HavingClause] [OrderByClause]
```

Items

```
UpdateItem  ::= IdentificationVariable "." (StateField | SingleValuedAssociationField) "=" NewValue
OrderByItem ::= (ResultVariable | SingleValuedPathExpression) ["ASC" | "DESC"]
GroupByItem ::= IdentificationVariable | SingleValuedPathExpression
NewValue    ::= ScalarExpression | SimpleEntityExpression | "NULL"
```

From, Join and Index by

```
IdentificationVariableDeclaration          ::= RangeVariableDeclaration [IndexBy] {JoinVariableDeclaration}*
SubselectIdentificationVariableDeclaration ::= IdentificationVariableDeclaration | (AssociationPathExpression ["AS"] AliasIdentificationVariable)
JoinVariableDeclaration                    ::= Join [IndexBy]
RangeVariableDeclaration                   ::= AbstractSchemaName ["AS"] AliasIdentificationVariable
Join                                       ::= ["LEFT" ["OUTER"] | "INNER"] "JOIN" JoinAssociationPathExpression
                                               ["AS"] AliasIdentificationVariable ["WITH" ConditionalExpression]
IndexBy                                    ::= "INDEX" "BY" SimpleStateFieldPathExpression
```

Select Expressions

```
SelectExpression        ::= IdentificationVariable | PartialObjectExpression | (AggregateExpression | "(" Subselect ")"  | FunctionDeclaration | ScalarExpression) [["AS"] AliasResultVariable]
SimpleSelectExpression  ::= ScalarExpression | IdentificationVariable |
                            (AggregateExpression [["AS"] AliasResultVariable])
PartialObjectExpression ::= "PARTIAL" IdentificationVariable "." PartialFieldSet
PartialFieldSet         ::= "{" SimpleStateField {"," SimpleStateField}* "}"
```

Expressions conditionnelles

```
ConditionalExpression       ::= ConditionalTerm {"OR" ConditionalTerm}*
ConditionalTerm             ::= ConditionalFactor {"AND" ConditionalFactor}*
ConditionalFactor           ::= ["NOT"] ConditionalPrimary
ConditionalPrimary          ::= SimpleConditionalExpression | "(" ConditionalExpression ")"
SimpleConditionalExpression ::= ComparisonExpression | BetweenExpression | LikeExpression |
                                InExpression | NullComparisonExpression | ExistsExpression |
                                EmptyCollectionComparisonExpression | CollectionMemberExpression |
                                InstanceOfExpression
```

Expressions Collection

```
EmptyCollectionComparisonExpression ::= CollectionValuedPathExpression "IS" ["NOT"] "EMPTY"
CollectionMemberExpression          ::= EntityExpression ["NOT"] "MEMBER" ["OF"] CollectionValuedPathExpression
```

Les valeurs littérales

```
Literal     ::= string | char | integer | float | boolean
InParameter ::= Literal | InputParameter
```

Paramètre d'entrée

```
InputParameter      ::= PositionalParameter | NamedParameter
PositionalParameter ::= "?" integer
NamedParameter      ::= ":" string
```

Expressions arithmétiques

```
ArithmeticExpression       ::= SimpleArithmeticExpression | "(" Subselect ")"
SimpleArithmeticExpression ::= ArithmeticTerm {("+" | "-") ArithmeticTerm}*
ArithmeticTerm             ::= ArithmeticFactor {("*" | "/") ArithmeticFactor}*
ArithmeticFactor           ::= [("+" | "-")] ArithmeticPrimary
ArithmeticPrimary          ::= SingleValuedPathExpression | Literal | "(" SimpleArithmeticExpression ")"
                               | FunctionsReturningNumerics | AggregateExpression | FunctionsReturningStrings
                               | FunctionsReturningDatetime | IdentificationVariable | InputParameter | CaseExpression
```

Expressions scalaires et le type

```
ScalarExpression       ::= SimpleArithmeticExpression | StringPrimary | DateTimePrimary | StateFieldPathExpression
                           BooleanPrimary | EntityTypeExpression | CaseExpression
StringExpression       ::= StringPrimary | "(" Subselect ")"
StringPrimary          ::= StateFieldPathExpression | string | InputParameter | FunctionsReturningStrings | AggregateExpression | CaseExpression
BooleanExpression      ::= BooleanPrimary | "(" Subselect ")"
```

```
BooleanPrimary          ::= StateFieldPathExpression | boolean | InputParameter
EntityExpression        ::= SingleValuedAssociationPathExpression | SimpleEntityExpression
SimpleEntityExpression  ::= IdentificationVariable | InputParameter
DatetimeExpression      ::= DatetimePrimary | "(" Subselect ")"
DatetimePrimary         ::= StateFieldPathExpression | InputParameter | FunctionsReturningDatetime | AggregateExpression
```

Expressions d'agrégat

```
AggregateExpression ::= ("AVG" | "MAX" | "MIN" | "SUM") "(" ["DISTINCT"] StateFieldPathExpression ")" |
                        "COUNT" "(" ["DISTINCT"] (IdentificationVariable | SingleValuedPathExpression) ")"
```

Expressions de cas

```
CaseExpression        ::= GeneralCaseExpression | SimpleCaseExpression | CoalesceExpression | NullifExpression
GeneralCaseExpression ::= "CASE" WhenClause {WhenClause}* "ELSE" ScalarExpression "END"
WhenClause            ::= "WHEN" ConditionalExpression "THEN" ScalarExpression
SimpleCaseExpression  ::= "CASE" CaseOperand SimpleWhenClause {SimpleWhenClause}* "ELSE" ScalarExpression "END"
CaseOperand           ::= StateFieldPathExpression | TypeDiscriminator
SimpleWhenClause      ::= "WHEN" ScalarExpression "THEN" ScalarExpression
CoalesceExpression    ::= "COALESCE" "(" ScalarExpression {"," ScalarExpression}* ")"
NullifExpression      ::= "NULLIF" "(" ScalarExpression "," ScalarExpression ")"
```

D'autres expressions

QUANTIFIED/BETWEEN/COMPARISON/LIKE/NULL/EXISTS

```
QuantifiedExpression     ::= ("ALL" | "ANY" | "SOME") "(" Subselect ")"
BetweenExpression        ::= ArithmeticExpression ["NOT"] "BETWEEN" ArithmeticExpression "AND" ArithmeticExpression
ComparisonExpression     ::= ArithmeticExpression ComparisonOperator ( QuantifiedExpression | ArithmeticExpression )
InExpression             ::= StateFieldPathExpression ["NOT"] "IN" "(" (InParameter {"," InParameter}* | Subselect) ")"
InstanceOfExpression     ::= IdentificationVariable ["NOT"] "INSTANCE" ["OF"] (InstanceOfParameter | "(" InstanceOfParameter {"," InstanceOfParameter}* ")")
InstanceOfParameter      ::= AbstractSchemaName | InputParameter
LikeExpression           ::= StringExpression ["NOT"] "LIKE" string ["ESCAPE" char]
NullComparisonExpression ::= (SingleValuedPathExpression | InputParameter) "IS" ["NOT"] "NULL"
ExistsExpression         ::= ["NOT"] "EXISTS" "(" Subselect ")"
ComparisonOperator       ::= "=" | "<" | "<=" | "<>" | ">" | ">=" | "!="
```

Fonctions

```
FunctionDeclaration ::= FunctionsReturningStrings | FunctionsReturningNumerics | FunctionsReturningDateTime

FunctionsReturningNumerics ::=
        "LENGTH" "(" StringPrimary ")" |
        "LOCATE" "(" StringPrimary "," StringPrimary ["," SimpleArithmeticExpression]")" |
        "ABS" "(" SimpleArithmeticExpression ")" | "SQRT" "(" SimpleArithmeticExpression ")" |
        "MOD" "(" SimpleArithmeticExpression "," SimpleArithmeticExpression ")" |
        "SIZE" "(" CollectionValuedPathExpression ")"

FunctionsReturningDateTime ::= "CURRENT_DATE" | "CURRENT_TIME" | "CURRENT_TIMESTAMP"

FunctionsReturningStrings ::=
        "CONCAT" "(" StringPrimary "," StringPrimary ")" |
        "SUBSTRING" "(" StringPrimary "," SimpleArithmeticExpression "," SimpleArithmeticExpression ")" |
        "TRIM" "(" [["LEADING" | "TRAILING" | "BOTH"] [char] "FROM"] StringPrimary ")" |
        "LOWER" "(" StringPrimary ")" |
        "UPPER" "(" StringPrimary ")"
```