14. Doctrine Query Language

DQL stands for Doctrine Query Language and is an Object Query Language derivate that is very similar to the Hibernate Query Language (HQL) or the Java Persistence Query Language (JPQL).

In essence, DQL provides powerful querying capabilities over your object model. Imagine all your objects lying around in some storage (like an object database). When writing DQL queries, think about querying that storage to pick a certain subset of your objects.

> A common mistake for beginners is to mistake DQL for being just some form of SQL and therefore trying to use table names and column names or join arbitrary tables together in a query. You need to think about DQL as a query language for your object model, not for your relational schema.

DQL is case in-sensitive, except for namespace, class and field names, which are case sensitive.

14.1. Types of DQL queries

DQL as a query language has SELECT, UPDATE and DELETE constructs that map to their corresponding SQL statement types. INSERT statements are not allowed in DQL, because entities and their relations have to be introduced into the persistence context through `EntityManager#persist()` to ensure consistency of your object model.

DQL SELECT statements are a very powerful way of retrieving parts of your domain model that are not accessible via associations. Additionally they allow to retrieve entities and their associations in one single SQL select statement which can make a huge difference in performance in contrast to using several queries.

DQL UPDATE and DELETE statements offer a way to execute bulk changes on the entities of your domain model. This is often necessary when you cannot load all the affected entities of a bulk update into memory.

14.2. SELECT queries

14.2.1. DQL SELECT clause

The select clause of a DQL query specifies what appears in the query result. The composition of all the expressions in the select clause also influences the nature of the query result.

Here is an example that selects all users with an age > 20:

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

Lets examine the query:

- `u` is a so called identification variable or alias that refers to the `MyProject\Model\User` class. By placing this alias in the SELECT clause we specify that we want all instances of the User class that are matched by this query to appear in the query result.
- The FROM keyword is always followed by a fully-qualified class name which in turn is followed by an identification variable or alias for that class name. This class designates a root of our query from which we can navigate further via joins (explained later) and path expressions.
- The expression `u.age` in the WHERE clause is a path expression. Path expressions in DQL are easily identified by the use of the '.' operator that is used for constructing paths. The path expression `u.age` refers to the `age` field on the User class.

The result of this query would be a list of User objects where all users are older than 20.

The SELECT clause allows to specify both class identification variables that signal the hydration of a complete entity class or just fields of the entity using the syntax `u.name`. Combinations of both are also allowed and it is possible to wrap both fields and identification values into aggregation and DQL functions. Numerical fields can be part of computations using mathematical operations. See the sub-section on Functions, Operators, Aggregates for more information.

14.2.2. Joins

A SELECT query can contain joins. There are 2 types of JOINs: "Regular" Joins and "Fetch" Joins.

Regular Joins: Used to limit the results and/or compute aggregate values.

Fetch Joins: In addition to the uses of regular joins: Used to fetch related entities and include them in the hydrated result of a query.

There is no special DQL keyword that distinguishes a regular join from a fetch join. A join (be it an inner or outer join) becomes a “fetch join” as soon as fields of the joined entity appear in the SELECT part of the DQL query outside of an aggregate function. Otherwise its a “regular join”.

Example:

Regular join of the address:

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

Fetch join of the address:

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

When Doctrine hydrates a query with fetch-join it returns the class in the FROM clause on the root level of the result array. In the previous example an array of User instances is returned and the address of each user is fetched and hydrated into the `User#address` variable. If you access the address Doctrine does not need to lazy load the association with another query.

> Doctrine allows you to walk all the associations between all the objects in your domain model. Objects that were not already loaded from the database are replaced with lazy load proxy instances. Non-loaded Collections are also replaced by lazy-load instances that fetch all the contained objects upon first access. However relying on the lazy-load mechanism leads to many small queries executed against the database, which can significantly affect the performance of your application. **Fetch Joins** are the solution to hydrate most or all of the entities that you need in a single SELECT query.

14.2.3. Named and Positional Parameters

DQL supports both named and positional parameters, however in contrast to many SQL dialects positional parameters are specified with numbers, for example ”?1”, ”?2” and so on. Named parameters are specified with ”:name1”, ”:name2” and so on.

When referencing the parameters in `Query#setParameter($param, $value)` both named and positional parameters are used **without** their prefixes.

14.2.4. DQL SELECT Examples

This section contains a large set of DQL queries and some explanations of what is happening. The actual result also depends on the hydration mode.

Hydrate all User entities:

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

Retrieve the IDs of all CmsUsers:

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

Retrieve the IDs of all users that have written an article:

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

Retrieve all articles and sort them by the name of the articles users 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
```

Retrieve the Username and Name of a CmsUser:

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

Retrieve a ForumUser and his single associated entity:

```
<?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());
```

Retrieve a CmsUser and fetch join all the phonenumbers he has:

```
<?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();
```

Hydrate a result in Ascending:

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

Or in Descending Order:

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

Using Aggregate Functions:

```
<?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();
```

With WHERE Clause and Positional Parameter:

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

With WHERE Clause and Named Parameter:

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

With Nested Conditions in WHERE Clause:

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

With COUNT DISTINCT:

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

With Arithmetic Expression in WHERE clause:

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

Retrieve user entities with Arithmetic Expression in ORDER clause, using the `HIDDEN` keyword:

```
<?php
$query = $em->createQuery('SELECT u, u.posts_count + u.likes_count AS HIDDEN score FROM CmsUser u ORDER BY score');
$users = $query->getResult(); // array of User objects
```

Using a LEFT JOIN to hydrate all user-ids and optionally associated article-ids:

```
<?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
```

Restricting a JOIN clause by additional conditions:

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

Using several Fetch JOINs:

```
<?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 in WHERE clause:

```
<?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();
```

DQL Functions in WHERE clause:

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

IN() Expression:

```
<?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();
```

CONCAT() DQL Function:

```
<?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 in WHERE clause with correlated Subquery

```
<?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();
```

Get all users who are members of $group.

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

Get all users that have more than 1 phonenumber

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

Get all users that have no phonenumber

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

Get all instances of a specific type, for use with inheritance hierarchies:

New in version 2.1.

```
<?php
$query = $em->createQuery('SELECT u FROM Doctrine\Tests\Models\Company\CompanyPerson u WHERE u INSTANCE OF Doctrine\Te
sts\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');
```

Get all users visible on a given website that have chosen certain gender:

New in version 2.2.

```
<?php
$query = $em->createQuery('SELECT u FROM User u WHERE u.gender IN (SELECT IDENTITY(agl.gender) FROM Site s JOIN s.acti
veGenderList agl WHERE s.id = ?1)');
```

New in version 2.4.

Starting with 2.4, the IDENTITY() DQL function also works for composite primary keys:

```
<?php
$query = $em->createQuery("SELECT IDENTITY(c.location, 'latitude') AS latitude, IDENTITY(c.location, 'longitude') AS l
ongitude FROM Checkpoint c WHERE c.user = ?1");
```

Joins between entities without associations were not possible until version 2.4, where you can generate an arbitrary join with the following syntax:

```
<?php
$query = $em->createQuery('SELECT u FROM User u JOIN Blacklist b WITH u.email = b.email');
```

14.2.4.1. Partial Object Syntax

By default when you run a DQL query in Doctrine and select only a subset of the fields for a given entity, you do not receive objects back. Instead, you receive only arrays as a flat rectangular result set, similar to how you would if you were just using SQL directly and joining some data.

If you want to select partial objects you can use the `partial` DQL keyword:

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

You use the partial syntax when joining as well:

```
<?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
```

14.2.4.2. “NEW” Operator Syntax

New in version 2.4.

Using the `NEW` operator you can construct Data Transfer Objects (DTOs) directly from DQL queries.

- When using `SELECT NEW` you don't need to specify a mapped entity.
- You can specify any PHP class, it only requires that the constructor of this class matches the `NEW` statement.
- This approach involves determining exactly which columns you really need, and instantiating a data-transfer object that contains a constructor with those arguments.

If you want to select data-transfer objects you should create a class:

```
<?php
class CustomerDTO
{
    public function __construct($name, $email, $city, $value = null)
    {
        // Bind values to the object properties.
    }
}
```

And then use the `NEW` DQL keyword :

```
<?php
$query = $em->createQuery('SELECT NEW CustomerDTO(c.name, e.email, a.city) FROM Customer c JOIN c.email e JOIN c.addre
ss a');
$users = $query->getResult(); // array of CustomerDTO
```

```
<?php
$query = $em->createQuery('SELECT NEW CustomerDTO(c.name, e.email, a.city, SUM(o.value)) FROM Customer c JOIN c.email
e JOIN c.address a JOIN c.orders o GROUP BY c');
$users = $query->getResult(); // array of CustomerDTO
```

Note that you can only pass scalar expressions to the constructor.

14.2.5. Using INDEX BY

The INDEX BY construct is nothing that directly translates into SQL but that affects object and array hydration. After each FROM and JOIN clause you specify by which field this class should be indexed in the result. By default a result is incremented by numerical keys starting with 0. However with INDEX BY you can specify any other column to be the key of your result, it really only makes sense with primary or unique fields though:

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

Returns an array of the following kind, indexed by both user-id then 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)
```

14.3. UPDATE queries

DQL not only allows to select your Entities using field names, you can also execute bulk updates on a set of entities using an DQL-UPDATE query. The Syntax of an UPDATE query works as expected, as the following example shows:

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

References to related entities are only possible in the WHERE clause and using sub-selects.

> DQL UPDATE statements are ported directly into a Database UPDATE statement and therefore bypass any locking scheme, events and do not increment the version column. Entities that are already loaded into the persistence context will **NOT** be synced with the updated database state. It is recommended to call `EntityManager#clear()` and retrieve new instances of any affected entity.

14.4. DELETE queries

DELETE queries can also be specified using DQL and their syntax is as simple as the UPDATE syntax:

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

The same restrictions apply for the reference of related entities.

> DQL DELETE statements are ported directly into a Database DELETE statement and therefore bypass any events and checks for the version column if they are not explicitly added to the WHERE clause of the query. Additionally Deletes of specifies entities are **NOT** cascaded to related entities even if specified in the metadata.

14.5. Functions, Operators, Aggregates

14.5.1. DQL Functions

The following functions are supported in SELECT, WHERE and HAVING clauses:

- IDENTITY(single_association_path_expression [, fieldMapping]) - Retrieve the foreign key column of association of the owning side
- ABS(arithmetic_expression)
- CONCAT(str1, str2)
- CURRENT_DATE() - Return the current date
- CURRENT_TIME() - Returns the current time
- CURRENT_TIMESTAMP() - Returns a timestamp of the current date and time.
- LENGTH(str) - Returns the length of the given string
- LOCATE(needle, haystack [, offset]) - Locate the first occurrence of the substring in the string.
- LOWER(str) - returns the string lowercased.
- MOD(a, b) - Return a MOD b.
- SIZE(collection) - Return the number of elements in the specified collection
- SQRT(q) - Return the square-root of q.
- SUBSTRING(str, start [, length]) - Return substring of given string.
- TRIM([LEADING | TRAILING | BOTH] ['trchar' FROM] str) - Trim the string by the given trim char, defaults to whitespaces.
- UPPER(str) - Return the upper-case of the given string.
- DATE_ADD(date, days, unit) - Add the number of days to a given date. (Supported units are DAY, MONTH)
- DATE_SUB(date, days, unit) - Substract the number of days from a given date. (Supported units are DAY, MONTH)
- DATE_DIFF(date1, date2) - Calculate the difference in days between date1-date2.

14.5.2. Arithmetic operators

You can do math in DQL using numeric values, for example:

```
SELECT person.salary * 1.5 FROM CompanyPerson person WHERE person.salary < 100000
```

14.5.3. Aggregate Functions

The following aggregate functions are allowed in SELECT and GROUP BY clauses: AVG, COUNT, MIN, MAX, SUM

14.5.4. Other Expressions

DQL offers a wide-range of additional expressions that are known from SQL, here is a list of all the supported constructs:

- `ALL/ANY/SOME` - Used in a WHERE clause followed by a sub-select this works like the equivalent constructs in SQL.
- `BETWEEN a AND b` and `NOT BETWEEN a AND b` can be used to match ranges of arithmetic values.
- `IN (x1, x2, ...)` and `NOT IN (x1, x2, ..)` can be used to match a set of given values.
- `LIKE ..` and `NOT LIKE ..` match parts of a string or text using % as a wildcard.
- `IS NULL` and `IS NOT NULL` to check for null values
- `EXISTS` and `NOT EXISTS` in combination with a sub-select

14.5.5. Adding your own functions to the DQL language

By default DQL comes with functions that are part of a large basis of underlying databases. However you will most likely choose a database platform at the beginning of your project and most likely never change it. For this cases you can easily extend the DQL parser with own specialized platform functions.

You can register custom DQL functions in your ORM Configuration:

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

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

The functions have to return either a string, numeric or datetime value depending on the registered function type. As an example we will add a MySQL specific FLOOR() functionality. All the given classes have to implement the base class :

```
<?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)
    {
        $parser->match(Lexer::T_IDENTIFIER);
        $parser->match(Lexer::T_OPEN_PARENTHESIS);

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

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

We will register the function by calling and can then use it:

```
<?php
$config = $em->getConfiguration();
$config->registerNumericFunction('FLOOR', 'MyProject\Query\MysqlFloor');

$dql = "SELECT FLOOR(person.salary * 1.75) FROM CompanyPerson person";
```

14.6. Querying Inherited Classes

This section demonstrates how you can query inherited classes and what type of results to expect.

14.6.1. Single Table

Single Table Inheritance (http://martinfowler.com/eaaCatalog/singleTableInheritance.html) is an inheritance mapping strategy where all classes of a hierarchy are mapped to a single database table. In order to distinguish which row represents which type in the hierarchy a so-called discriminator column is used.

First we need to setup an example set of entities to use. In this scenario it is a generic Person and Employee example:

```
<?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;

    // ...
}
```

First notice that the generated SQL to create the tables for these entities looks like the following:

```
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
)
```

Now when persist a new `Employee` instance it will set the discriminator value for us automatically:

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

Now lets run a simple query to retrieve the `Employee` we just created:

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

If we check the generated SQL you will notice it has some special conditions added to ensure that we will only get back `Employee` entities:

```
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')
```

14.6.2. Class Table Inheritance

Class Table Inheritance (http://martinfowler.com/eaaCatalog/classTableInheritance.html) is an inheritance mapping strategy where each class in a hierarchy is mapped to several tables: its own table and the tables of all parent classes. The table of a child class is linked to the table of a parent class through a foreign key constraint. Doctrine 2 implements this strategy through the use of a discriminator column in the topmost table of the hierarchy because this is the easiest way to achieve polymorphic queries with Class Table Inheritance.

The example for class table inheritance is the same as single table, you just need to change the inheritance type from `SINGLE_TABLE` to `JOINED`:

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

Now take a look at the SQL which is generated to create the table, you'll notice some differences:

```
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
```

- The data is split between two tables
- A foreign key exists between the two tables

Now if were to insert the same `Employee` as we did in the `SINGLE_TABLE` example and run the same example query it will generate different SQL joining the `Person` information automatically for you:

```
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 = ?
```

14.7. The Query class

An instance of the `Doctrine\ORM\Query` class represents a DQL query. You create a Query instance be calling `EntityManager#createQuery($dql)`, passing the DQL query string. Alternatively you can create an empty `Query` instance and invoke `Query#setDql($dql)` afterwards. Here are some examples:

```
<?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');
```

14.7.1. Query Result Formats

The format in which the result of a DQL SELECT query is returned can be influenced by a so-called `hydration mode`. A hydration mode specifies a particular way in which a SQL result set is transformed. Each hydration mode has its own dedicated method on the Query class. Here they are:

- `Query#getResult()`: Retrieves a collection of objects. The result is either a plain collection of objects (pure) or an array where the objects are nested in the result rows (mixed).
- `Query#getSingleResult()`: Retrieves a single object. If the result contains more than one object, an `NonUniqueResultException` is thrown. If the result contains no objects, an `NoResultException` is thrown. The pure/mixed distinction does not apply.
- `Query#getOneOrNullResult()`: Retrieve a single object. If no object is found null will be returned.
- `Query#getArrayResult()`: Retrieves an array graph (a nested array) that is largely interchangeable with the object graph generated by `Query#getResult()` for read-only purposes.

> An array graph can differ from the corresponding object graph in certain scenarios due to the difference of the identity semantics between arrays and objects.

- `Query#getScalarResult()`: Retrieves a flat/rectangular result set of scalar values that can contain duplicate data. The pure/mixed distinction does not apply.
- `Query#getSingleScalarResult()`: Retrieves a single scalar value from the result returned by the dbms. If the result contains more than a single scalar value, an exception is thrown. The pure/mixed distinction does not apply.

Instead of using these methods, you can alternatively use the general-purpose method `Query#execute(array $params = array(), $hydrationMode = Query::HYDRATE_OBJECT)`. Using this method you can directly supply the hydration mode as the second parameter via one of the Query constants. In fact, the methods mentioned earlier are just convenient shortcuts for the execute method. For example, the method `Query#getResult()` internally invokes execute, passing in `Query::HYDRATE_OBJECT` as the hydration mode.

The use of the methods mentioned earlier is generally preferred as it leads to more concise code.

14.7.2. Pure and Mixed Results

The nature of a result returned by a DQL SELECT query retrieved through `Query#getResult()` or `Query#getArrayResult()` can be of 2 forms: **pure** and **mixed**. In the previous simple examples, you already saw a “pure” query result, with only objects. By default, the result type is **pure** but **as soon as scalar values, such as aggregate values or other scalar values that do not belong to an entity, appear in the SELECT part of the DQL query, the result becomes mixed**. A mixed result has a different structure than a pure result in order to accommodate for the scalar values.

A pure result usually looks like this:

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

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

A mixed result on the other hand has the following general structure:

```
$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
```

To better understand mixed results, consider the following DQL query:

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

This query makes use of the `UPPER` DQL function that returns a scalar value and because there is now a scalar value in the SELECT clause, we get a mixed result.

Conventions for mixed results are as follows:

- The object fetched in the FROM clause is always positioned with the key ‘0’.
- Every scalar without a name is numbered in the order given in the query, starting with 1.
- Every aliased scalar is given with its alias-name as the key. The case of the name is kept.
- If several objects are fetched from the FROM clause they alternate every row.

Here is how the result could look like:

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

And here is how you would access it in PHP code:

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

14.7.3. Fetching Multiple FROM Entities

If you fetch multiple entities that are listed in the FROM clause then the hydration will return the rows iterating the different top-level entities.

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

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

14.7.4. Hydration Modes

Each of the Hydration Modes makes assumptions about how the result is returned to user land. You should know about all the details to make best use of the different result formats:

The constants for the different hydration modes are:

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

14.7.4.1. Object Hydration

Object hydration hydrates the result set into the object graph:

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

14.7.4.2. Array Hydration

You can run the same query with array hydration and the result set is hydrated into an array that represents the object graph:

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

You can use the `getArrayResult()` shortcut as well:

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

14.7.4.3. Scalar Hydration

If you want to return a flat rectangular result set instead of an object graph you can use scalar hydration:

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

The following assumptions are made about selected fields using Scalar Hydration:

Fields from classes are prefixed by the DQL alias in the result. A query of the kind 'SELECT u.name ..' returns a key 'u_name' in the result rows.

14.7.4.4. Single Scalar Hydration

If you have a query which returns just a single scalar value you can use single scalar hydration:

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

You can use the `getSingleScalarResult()` shortcut as well:

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

14.7.4.5. Custom Hydration Modes

You can easily add your own custom hydration modes by first creating a class which extends `AbstractHydrator`:

```php
<?php
namespace MyProject\Hydrators;

use Doctrine\ORM\Internal\Hydration\AbstractHydrator;

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

Next you just need to add the class to the ORM configuration:

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

Now the hydrator is ready to be used in your queries:

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

14.7.5. Iterating Large Result Sets

There are situations when a query you want to execute returns a very large result-set that needs to be processed. All the previously described hydration modes completely load a result-set into memory which might not be feasible with large result sets. See the Batch Processing (batch-processing.html) section on details how to iterate large result sets.

14.7.6. Functions

The following methods exist on the `AbstractQuery` which both `Query` and `NativeQuery` extend from.

14.7.6.1. Parameters

Prepared Statements that use numerical or named wildcards require additional parameters to be executable against the database. To pass parameters to the query the following methods can be used:

- `AbstractQuery::setParameter($param, $value)` - Set the numerical or named wildcard to the given value.
- `AbstractQuery::setParameters(array $params)` - Set an array of parameter key-value pairs.

- `AbstractQuery::getParameter($param)`
- `AbstractQuery::getParameters()`

Both named and positional parameters are passed to these methods without their ? or : prefix.

14.7.6.2. Cache related API

You can cache query results based either on all variables that define the result (SQL, Hydration Mode, Parameters and Hints) or on user-defined cache keys. However by default query results are not cached at all. You have to enable the result cache on a per query basis. The following example shows a complete workflow using the Result Cache 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();
```

> You can set the Result Cache Driver globally on the `Doctrine\ORM\Configuration` instance so that it is passed to every `Query` and `NativeQuery` instance.

14.7.6.3. Query Hints

You can pass hints to the query parser and hydrators by using the `AbstractQuery::setHint($name, $value)` method. Currently there exist mostly internal query hints that are not be consumed in userland. However the following few hints are to be used in userland:

- Query::HINT_FORCE_PARTIAL_LOAD - Allows to hydrate objects although not all their columns are fetched. This query hint can be used to handle memory consumption problems with large result-sets that contain char or binary data. Doctrine has no way of implicitly reloading this data. Partially loaded objects have to be passed to `EntityManager::refresh()` if they are to be reloaded fully from the database.
- Query::HINT_REFRESH - This query is used internally by `EntityManager::refresh()` and can be used in userland as well. If you specify this hint and a query returns the data for an entity that is already managed by the UnitOfWork, the fields of the existing entity will be refreshed. In normal operation a result-set that loads data of an already existing entity is discarded in favor of the already existing entity.
- Query::HINT_CUSTOM_TREE_WALKERS - An array of additional `Doctrine\ORM\Query\TreeWalker` instances that are attached to the DQL query parsing process.

14.7.6.4. Query Cache (DQL Query Only)

Parsing a DQL query and converting it into a SQL query against the underlying database platform obviously has some overhead in contrast to directly executing Native SQL queries. That is why there is a dedicated Query Cache for caching the DQL parser results. In combination with the use of wildcards you can reduce the number of parsed queries in production to zero.

The Query Cache Driver is passed from the `Doctrine\ORM\Configuration` instance to each `Doctrine\ORM\Query` instance by default and is also enabled by default. This also means you don't regularly need to fiddle with the parameters of the Query Cache, however if you do there are several methods to interact with it:

- `Query::setQueryCacheDriver($driver)` - Allows to set a Cache instance
- `Query::setQueryCacheLifeTime($seconds = 3600)` - Set lifetime of the query caching.
- `Query::expireQueryCache($bool)` - Enforce the expiring of the query cache if set to true.
- `Query::getExpireQueryCache()`
- `Query::getQueryCacheDriver()`
- `Query::getQueryCacheLifeTime()`

14.7.6.5. First and Max Result Items (DQL Query Only)

You can limit the number of results returned from a DQL query as well as specify the starting offset, Doctrine then uses a strategy of manipulating the select query to return only the requested number of results:

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

> If your query contains a fetch-joined collection specifying the result limit methods are not working as you would expect. Set Max Results restricts the number of database result rows, however in the case of fetch-joined collections one root entity might appear in many rows, effectively hydrating less than the specified number of results.

14.7.6.6. Temporarily change fetch mode in DQL

While normally all your associations are marked as lazy or extra lazy you will have cases where you are using DQL and don't want to fetch join a second, third or fourth level of entities into your result, because of the increased cost of the SQL JOIN. You can mark a many-to-one or one-to-one association as fetched temporarily to batch fetch these entities using a WHERE .. IN query.

```
<?php
$query = $em->createQuery("SELECT u FROM MyProject\User u");
$query->setFetchMode("MyProject\User", "address", \Doctrine\ORM\Mapping\ClassMetadata::FETCH_EAGER);
$query->execute();
```

Given that there are 10 users and corresponding addresses in the database the executed queries will look something like:

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

> Changing the fetch mode during a query is only possible for one-to-one and many-to-one relations.

14.8. EBNF

The following context-free grammar, written in an EBNF variant, describes the Doctrine Query Language. You can consult this grammar whenever you are unsure about what is possible with DQL or what the correct syntax for a particular query should be.

14.8.1. Document syntax:

- non-terminals begin with an upper case character
- terminals begin with a lower case character
- parentheses (...) are used for grouping
- square brackets [...] are used for defining an optional part, e.g. zero or one time
- curly brackets {...} are used for repetition, e.g. zero or more times
- double quotation marks "..." define a terminal string
- a vertical bar | represents an alternative

14.8.2. Terminals

- identifier (name, email, ...) must match `[a-z_][a-z0-9_]*`
- fully_qualified_name (DoctrineTestsModelsCMSCmsUser) matches PHP's fully qualified class names
- aliased_name (CMS:CmsUser) uses two identifiers, one for the namespace alias and one for the class inside it
- string ('foo', 'bar''s house', '%ninja%', ...)
- char ('/', '\', ' ', ...)
- integer (-1, 0, 1, 34, ...)
- float (-0.23, 0.007, 1.245342E+8, ...)
- boolean (false, true)

14.8.3. Query Language

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

14.8.4. Statements

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

14.8.5. Identifiers

```
/* 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"), possibly as a fully qualified class name or nam
espace-aliased */
AbstractSchemaName ::= fully_qualified_name | aliased_name | identifier

/* 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

/* 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 */
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
```

14.8.6. Path Expressions

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

/* "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

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

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

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

14.8.7. Clauses

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

14.8.8. Items

```
UpdateItem  ::= SingleValuedPathExpression "=" NewValue
OrderByItem ::= (SimpleArithmeticExpression | SingleValuedPathExpression | ScalarExpression | ResultVariable | Functio
nDeclaration) ["ASC" | "DESC"]
GroupByItem ::= IdentificationVariable | ResultVariable | SingleValuedPathExpression
NewValue    ::= SimpleArithmeticExpression | "NULL"
```

14.8.9. From, Join and Index by

```
IdentificationVariableDeclaration          ::= RangeVariableDeclaration [IndexBy] {Join}*
SubselectIdentificationVariableDeclaration ::= IdentificationVariableDeclaration
RangeVariableDeclaration                   ::= AbstractSchemaName ["AS"] AliasIdentificationVariable
JoinAssociationDeclaration                 ::= JoinAssociationPathExpression ["AS"] AliasIdentificationVariable [Index
By]
Join                                       ::= ["LEFT" ["OUTER"] | "INNER"] "JOIN" (JoinAssociationDeclaration | Range
VariableDeclaration) ["WITH" ConditionalExpression]
IndexBy                                    ::= "INDEX" "BY" StateFieldPathExpression
```

14.8.10. Select Expressions

```
SelectExpression        ::= (IdentificationVariable | ScalarExpression | AggregateExpression | FunctionDeclaration | P
artialObjectExpression | "(" Subselect ")" | CaseExpression | NewObjectExpression) [["AS"] ["HIDDEN"] AliasResultVaria
ble]
SimpleSelectExpression  ::= (StateFieldPathExpression | IdentificationVariable | FunctionDeclaration | AggregateExpres
sion | "(" Subselect ")" | ScalarExpression) [["AS"] AliasResultVariable]
PartialObjectExpression ::= "PARTIAL" IdentificationVariable "." PartialFieldSet
PartialFieldSet         ::= "{" SimpleStateField {"," SimpleStateField}* "}"
NewObjectExpression     ::= "NEW" AbstractSchemaName "(" NewObjectArg {"," NewObjectArg}* ")"
NewObjectArg            ::= ScalarExpression | "(" Subselect ")"
```

14.8.11. Conditional Expressions

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

14.8.12. Collection Expressions

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

14.8.13. Literal Values

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

14.8.14. Input Parameter

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

14.8.15. Arithmetic Expressions

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

14.8.16. Scalar and Type Expressions

```
ScalarExpression       ::= SimpleArithmeticExpression | StringPrimary | DateTimePrimary | StateFieldPathExpression | B
ooleanPrimary | CaseExpression | InstanceOfExpression
StringExpression       ::= StringPrimary | ResultVariable | "(" Subselect ")"
StringPrimary          ::= StateFieldPathExpression | string | InputParameter | FunctionsReturningStrings | AggregateE
xpression | CaseExpression
BooleanExpression      ::= BooleanPrimary | "(" Subselect ")"
BooleanPrimary         ::= StateFieldPathExpression | boolean | InputParameter
EntityExpression       ::= SingleValuedAssociationPathExpression | SimpleEntityExpression
SimpleEntityExpression ::= IdentificationVariable | InputParameter
DatetimeExpression     ::= DatetimePrimary | "(" Subselect ")"
DatetimePrimary        ::= StateFieldPathExpression | InputParameter | FunctionsReturningDatetime | AggregateExpressio
n
```

Parts of CASE expressions are not yet implemented.

14.8.17. Aggregate Expressions

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

14.8.18. Case Expressions

```
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 ")"
```

14.8.19. Other 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             ::= SingleValuedPathExpression ["NOT"] "IN" "(" (InParameter {"," InParameter}* | Subselect)
")"
InstanceOfExpression     ::= IdentificationVariable ["NOT"] "INSTANCE" ["OF"] (InstanceOfParameter | "(" InstanceOfPar
ameter {"," InstanceOfParameter}* ")")
InstanceOfParameter      ::= AbstractSchemaName | InputParameter
LikeExpression           ::= StringExpression ["NOT"] "LIKE" StringPrimary ["ESCAPE" char]
NullComparisonExpression ::= (InputParameter | NullIfExpression | CoalesceExpression | AggregateExpression | FunctionD
eclaration | IdentificationVariable | SingleValuedPathExpression | ResultVariable) "IS" ["NOT"] "NULL"
ExistsExpression         ::= ["NOT"] "EXISTS" "(" Subselect ")"
ComparisonOperator       ::= "=" | "<" | "<=" | "<>" | ">" | ">=" | "!="
```

14.8.20. Functions

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

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

FunctionsReturningDateTime ::=
        "CURRENT_DATE" |
        "CURRENT_TIME" |
        "CURRENT_TIMESTAMP" |
        "DATE_ADD" "(" ArithmeticPrimary "," ArithmeticPrimary "," StringPrimary ")" |
        "DATE_SUB" "(" ArithmeticPrimary "," ArithmeticPrimary "," StringPrimary ")"

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