Die Java Persistence Query Language

Objektorientiertes SQL

Der Standard zur Speicherung und Verwaltung von Daten in Java-Anwendungen wird als Java Persistence API bezeichnet. Dieser beinhaltet die leistungsfähige, objektorientierte Abfragesprache JPQL. **Frank Schwarz**

Auf einen Blick

- Die Java Persistence Query Language – kurz JPQL – ist eine standardisierte Sprache, um in Java-Anwendungen Datenabfragen durchzuführen, wobei das Ergebnis in Java-Objekte übernommen wird.
- Dieser Artikel beschreibt anhand von Beispielen die grundlegende Datenabfrage-Syntax von JPQL.
- **Plattform**
  Java EE 5
- **Technik/Anwendung**
  O/R-Mapping, JPQL-Abfragesprache
- **Voraussetzungen**
  DBMS (zum Beispiel Oracle, SQL Server, DB2, MySQL)

Die Spezifikation der Java Persistence API zerfällt in drei große Bereiche: die Mapping-Spezifikation für die Abbildung von Klassen auf Tabellen, das API für den Umgang mit persistenten Objekten und eine Abfragesprache zum zielgerichteten Laden größerer Objektmengen. Der Spezifikationsgruppe hinter JPA ist mit der Abfragesprache ein kleines Meisterstück gelungen. Mit den Sub-Selektionen und Aggregat-Funktionen sieht die Sprache zwar aus wie SQL, funktioniert aber in sich streng objektorientiert.

Ein transparentes objektrelationales Mapping (O/R-Mapping) hat gegenüber reinem SQL zahlreiche Vorzüge, etwa ein automatisches Speichern von Änderungen, feingranulares Caching zur Performance-Steigerung und transparente Sperrprotokolle zum Schutz gegen Lost Updates. Dennoch schrecken viele vor dem Einsatz dieser Technologie zurück. Zwei Argumente halten sich dabei hartnäckig: Erstens sei O/R-Mapping zu langsam und zweitens sei die Abfragesprache für den praktischen Einsatz zu unflexibel. Das erste Argument erscheint auf den ersten Blick plausibel, da ein O/R-Mapper zusätzliche Metadaten verarbeitet, was bei einem direkten Zugriff auf die Datenbank nicht notwendig wäre. Eine isolierte Betrachtungsweise wird realen Projekten jedoch kaum gerecht. Auch wenn ein O/R-Mapper vermieden wird, existiert doch in jedem Projekt eine gewisse Abstraktionsschicht für den Datenzugriff. Dementsprechend findet man kaum eine selbst geschriebene Persistenz-Schicht, die mit der Performance der bekannten O/R-Mapper für Java mithalten kann. Auch das zweite Argument ist haltlos, da man in JPA direkt auch native SQL-Abfragen formulieren kann. Doch so weit muss es nicht kommen.

Select your objects, please

JPQL, die Abfragesprache für JPA, hält sicher nur wenige Überraschungen für einen SQL-Experten bereit. Die kürzeste Abfrage in JPQL lässt sich formulieren als:

```
SELECT b FROM Buch b
```

Wenn die Klasse *Buch* in keiner Vererbungsbeziehung steht, resultiert daraus ein fast identisches SQL-Statement:

```
SELECT b.* FROM BUCH b
```

Zur Einschränkung der Auswahl kann man im *WHERE*-Teil der Abfrage weitere Kriterien der Selektion angeben:

```
SELECT b FROM Buch b WHERE b.seiten
< 500 AND b.titel LIKE '%Java%'
```

Der objektorientierte Aspekt kommt zum Tragen, wenn man auf Eigenschaften eines assoziierten Objekts zugreifen möchte:

```
SELECT b FROM Buch b
WHERE b.verlag.name = 'Rowohlt'
```

selektiert alle Bücher, die mit einem Verlag-Objekt verbunden sind, dessen Eigenschaft *name* gleich „Rowohlt“ ist. Auch für die Sortierung der Ergebnismenge funktioniert die SQL-Syntax:

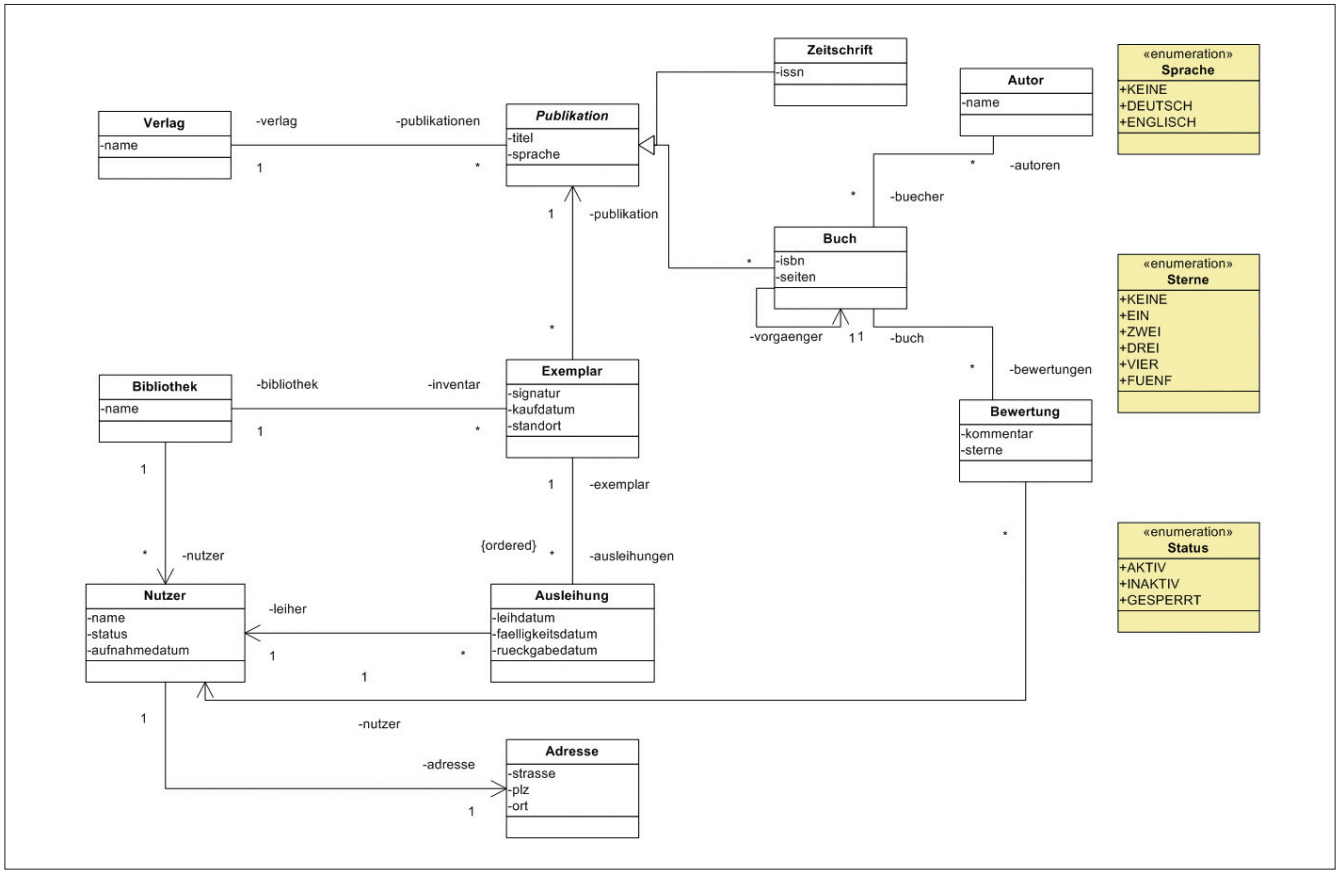
```
SELECT b FROM Buch b ORDER BY
b.titel ASC
```

liefert alle Bücher aufsteigend sortiert entsprechend ihres Titels zurück. Kaum verwundern sollte, dass auch die Schlüsselwörter *GROUP BY* und *HAVING* ihren Platz in JPQL haben. Eine ausführliche Sprachreferenz zu JPQL finden Sie im Internet unter der Adresse [1].

Szenario Buchverwaltung

Um die Ausdrucksstärke von JPQL zu zeigen, wird nachfolgend ein kleines Objektmodell verwendet (Bild 1): Ausgangspunkt ist eine Bibliothek mit einem Inventar an Buch- und Zeitschriftenexemplaren. Da die Bibliothek mehrere Exemplare eines Buches oder einer Zeitschrift beinhalten kann, wird streng zwischen Typ- und Exemplar-Ebene unterschieden. Bücher und Zeitschriften werden zur Superklasse *Publikation* zusammengeführt. Bestandteil der Bibliothek sind Nutzer, die Ausleihungen tätigen und Publikationen bewerten. Streng objektorientiert kann jedes Buch-Objekt mit mehreren Autor-Objekten assoziiert werden.

Jede Publikation referenziert genau ein Verlag-Objekt. Ein Buch referenziert optional ein weiteres Buch, falls dieses als sein Vorgänger angesehen werden kann. So kann etwa das Buch



Szenario einer Buchverwaltung als UML-Klassendiagramm mit Pfeilen, die gerichtete Beziehungen veranschaulichen (Bild 1)

„Hibernate in Action“ als Vorgänger zu „Java Persistence with Hibernate“ betrachtet werden.

Einfache Abfragen

JPQL bietet Operatoren, vordefinierte Literale und Funktionen, um komplexe Filterbedingungen aufzustellen (vergleiche Textkasten „Elemente für Abfragen“, Seite 38). Um alle Ausleihungen abzufragen, deren Fälligkeitsdatum in der Vergangenheit liegt, kann man formulieren:

```
SELECT a FROM Ausleihung WHERE
a.faelligkeitsdatum < CURRENT_DATE
```

CURRENT_DATE ist ein Literal, welches automatisch zum heutigen Datum ohne Uhrzeit expandiert wird. Möchte man alle überfälligen Ausleihungen abfragen, also Ausleihungen mit abgelaufenem Fälligkeitsdatum und ohne gesetztes Rückgabedatum, dann werden zwei Filterkriterien durch ein logisches *AND* verknüpft:

```
SELECT a FROM Ausleihung WHERE
a.faelligkeitsdatum < CURRENT_DATE
AND a.rueckgabedatum IS NULL
```

Die Prüfung auf einen *NULL*-Wert erfolgt in gleicher Weise wie in SQL. Ein Filter der Art *a.rueckgabedatum = NULL* wäre unzulässig.

Wie bereits gezeigt, lassen sich String-Literale durch einfache Hochkommas in den Filter integrieren. Das Escaping erfolgt durch Verdopplung des Hochkommas. Alle Autoren, die James O'Brian heißen, findet man folglich so:

```
SELECT a FROM Autor a WHERE a.name
= 'James O''Brian'
```

Auch Literale der Enumerationstypen lassen sich nutzen. Alle englischen Bücher erfragt man mit dieser Query:

```
SELECT b FROM Book b WHERE b.sprache
= buchverwaltung.Sprache.ENGLISCH
```

Literale lassen sich durch Input-Parameter ersetzen. Dies ist die empfohlene Verwendungsweise, da man sich dann über ein korrektes Escaping keine Gedanken machen muss und auch eine JPQL-Injection unmöglich wird. JPA sieht zwei Möglichkeiten für Input-Parameter vor: Positionsparameter und benannte Parameter. Eine Abfrage nach englischen Büchern sieht für Positionsparameter so aus:

```
SELECT b FROM Book b WHERE b.sprache
= ?1
```

Für benannte Parameter sieht die Abfrage folgendermaßen aus:

```
SELECT b FROM Book b WHERE b.sprache
= :sprache
```

Numerische Literale

Numerische Literale können in der bekannten Java- und SQL-Syntax angegeben werden. Literale für beliebige Datums- und Uhrzeitwerte sind in JPA 1.0 noch nicht vorgesehen.

Elemente für Abfragen

TRUE, FALSE
CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP
OR, AND, NOT
+, –, *, /
=, >, >=, <, <=, <>
[NOT] BETWEEN,
[NOT] LIKE
[NOT] IN
IS [NOT] NULL
IS [NOT] EMPTY
[NOT] MEMBER [OF]
CONCAT, SUBSTRING, TRIM
LOWER, UPPER
LENGTH, LOCATE
ABS, SQRT, MOD, SIZE

Im ersten Fall wird der Parameter über die Zahl identifiziert, hier *1*, im zweiten Fall über seinen Namen, hier *sprache*. Mit Hilfe von Input-Parametern werden auch gezielte Datumsabfragen möglich. Möchte man alle Nutzer in Erfahrung bringen, die vor dem 1.1.2008 aufgenommen wurden, lässt sich fragen:

```
SELECT n FROM Nutzer n WHERE
n.aufnahmedatum < :datum
```

Für den Parameter *datum* muss man später ein Objekt vom Typ *java.util.Date* an die Query übergeben.

Spezialoperatoren

Für den Umgang mit Strings und numerischen Werten definiert die JPA-Spezifikation weitere Ausdrücke und Operatoren. Für Bereichsprüfungen existiert ein *BETWEEN*-Ausdruck:

```
SELECT b FROM Buch b WHERE
b.seiten BETWEEN 400 AND 700
```

Für die Musterprüfung in Strings kann ein *LIKE*-Ausdruck herangezogen werden. Alle Publikationen, die das Wort „SQL“ im Titel tragen, erfragt man so:

```
SELECT p FROM Publikation p WHERE
p.titel LIKE '%SQL%'
```

Während das Zeichen % für einen beliebigen String steht, kann man das Zeichen _ (Unterstrich) für den Ersatz eines einzelnen Buchstabens nutzen. Kompliziertere reguläre Ausdrücke sind leider nicht möglich. Eine weitere Filtermöglichkeit stellt der *IN*-Ausdruck dar. Hiermit kann man als Kriterium eine Liste von Möglichkeiten angeben. Möchte man Bücher mit einer bestimmten ISBN selektieren, kann man dies folgendermaßen formulieren:

```
SELECT b FROM Buch b WHERE b.isbn
IN('12-23-4', '23-27-4', '72-88-3')
```

Um alle Bücher zu finden, die im Titel „SQL“ tragen, kann man auch auf die String-Funktion *LOCATE* zurückgreifen:

```
SELECT b FROM Buch b WHERE
LOCATE('SQL', b.titel) <> 0
```

Des Weiteren existieren Funktionen zum Verknüpfen von Strings, zum Bilden eines Substrings, zum Trimmen eines Strings und zur Änderung der Groß- und Kleinschreibung:

```
SELECT b FROM Buch b WHERE
UPPER(b.titel) = 'SQL'
```

findet beispielsweise auch Bücher mit der Schreibweise *Sql*.

Zu beachten ist, dass all diese Funktionen bei JPA 1.0 nur im *WHERE*- oder *HAVING*-Teil der Abfrage vorkommen dürfen. Auch für numerische Werte existieren drei Funktionen, die neben den üblichen arithmetischen Operatoren verwendet werden können: der Absolutwert (*ABS*), die Quadratwurzel (*SQRT*) und der Modulus-Operator (*MOD*).

Navigationen

Alle Filterbedingungen bezogen sich bisher auf die Kandidatenklasse der Abfrage. Dies muss jedoch nicht immer der Fall sein. Die objektorientierte Seite von JPQL ermöglicht eine Navigation über den Objektgrafen in einfacher Punkt-Notation. Einzige Bedingung hierfür ist, dass eine einwertige Beziehung (1-1 oder N-1) vorliegt und dass diese Beziehung auf Instanz-Ebene erfüllt ist.

Alle Bücher aus dem Verlag „Reclam“ erhält man beispielsweise über die Query:

```
SELECT b FROM Buch b WHERE
b.verlag.name ='Reclam'
```

Eine Falle lauert hier, wenn die Beziehung nicht erfüllt ist. Fehlt beispielsweise manchem Nutzer die Adressangabe, so findet die Abfrage

```
SELECT n FROM Nutzer n WHERE
n.adresse.stadt ='Lüdenscheid'
OR n.name = 'Lüdenscheid'
```

nur dann einen Nutzer mit dem Namen „Lüdenscheid“, wenn für diesen auch eine Adresse hinterlegt ist. Hier wäre seitens der Spezifikation

Java Persistence API

Die Java Persistence API (JPA) ist ein Java-Standard, auf den sich die Softwareindustrie 2006 geeinigt hat. Er wird unter anderem von Sun, Oracle, SAP, IBM, JBoss, BEA, Borland und der Apache Software Foundation unterstützt. Mittlerweile gibt es eine Vielzahl an Softwarebibliotheken, die kompatibel zu JPA sind, wie Hibernate oder Oracle Toplink. Der Begriff Persistence bezeichnet dabei die Fähigkeit, Laufzeitobjekte über eine einzelne Sitzung hinaus auch dauerhaft zu speichern. Als Datenspeicher fungieren dabei relationale Datenbanken. JPA ist Teil der Enterprise-Javabeans-3.0-Spezifikation und damit Bestandteil der Java-EE-5-Plattform. Sie bietet objektrelationales Mapping, ein Entity-Manager-API sowie die Abfragesprache JPQL. Beim O/R-Mapping werden Beziehungen zwischen den objektorientierten Java-Klassen und einzelnen Tabellen in einer relationalen Datenbank über objektrelationale Metadaten hergestellt. Die Metadaten werden als Java-Annotationen angelegt oder in gesonderten XML-Dateien abgelegt. Das Entity-Manager-API ist eine gesonderte Programmierschnittstelle, über die CRUD-Operationen durchgeführt werden können (Create, Read, Update und Delete). JPQL ist schließlich die objektorientierte Abfragesprache. Weiterführende Informationen zum O/R-Mapping und JPQL finden Sie im Internet unter den Adressen [1], [2] und [3].

Besseres möglich gewesen, da das JPA-Mapping für einwertige Beziehungen die Angabe beinhaltet, ob eine Beziehung optional oder verpflichtend ist. Damit kein Ungemach aufgrund falscher Join-Bedingungen im generierten SQL-Statement droht, erlaubt die JPA-Spezifikation die Ausformulierung der Objekt-Verknüpfungen. Kann es also vorkommen, dass ein Nutzer keine Adresse besitzt, so müsste man die Suche nach dem Namen *Lüdenscheid* über einen Left-Outer-Join formulieren:

```
SELECT n FROM Nutzer n LEFT JOIN
n.adresse a WHERE a.stadt =
'Lüdenscheid' OR n.name =
'Lüdenscheid'
```

Die exakte Join-Bedingung kann der O/R-Mapper aus den Mapping-Daten ableiten, sie wird deshalb nicht mit angegeben. Eine Navigation mit Inner-Join wird dementsprechend so formuliert:

```
SELECT n FROM Nutzer n JOIN n.adresse
a WHERE a.stadt ='Lüdenscheid'
```

Für die Navigation entlang mehrwertiger Objektbeziehungen existieren ebenfalls zwei syntaktische Möglichkeiten. Die erste Variante entspricht dem Join-Verhalten der Punkt-Navigation, lässt sich aber leider nicht so elegant ausdrücken: Eine Abfrage nach allen Büchern, bei denen mindestens ein Autor auf „Waal" lautet, sähe so aus:

```
SELECT b FROM Buch b, IN(b.autoren) a
WHERE a.name LIKE '%Waal'
```

Möchte man über mehrere mehrwertige Assoziationen navigieren, so kann der *IN*-Ausdruck mehrfach angewendet werden. Von der Bibliothek zu Ausleihung kommt man über

```
SELECT ... FROM Bibliothek b,
IN(b.inventar) expl,
IN(expl.ausleihungen) a WHERE
a.leihdatum ...
```

Der *IN*-Ausdruck führt im generierten SQL-Statement zu einem Inner-Join zwischen den beteiligten Relationen. Damit kommen keine Objekte in der Ergebnismenge vor, bei denen die Assoziation auf Instanzebene nicht erfüllt ist. Möchte man dennoch dieses Verhalten erreichen, so empfiehlt sich die zweite Variante der Navigation:

```
SELECT b FROM Buch b LEFT JOIN
b.autoren a WHERE b.name LIKE
'%Nachhut%' OR a.name LIKE '%Waal'
```

Navigationen werden auch benutzt, um Objekte zu vergleichen. Auch hier unterscheidet sich die Syntax wieder bei ein- und mehrwertigen Assoziationen. Hat man beispielsweise bereits ein Buch-Objekt geladen und möchte wissen, für welche anderen Bücher dies den Vorgänger darstellt, so kommt man mit dem Objekt-Modell allein nicht weiter. Modelliert ist nur die Vorgänger-, nicht aber die Nachfolger-Beziehung. Eine Query schafft Abhilfe:

Listing 1: Paging

```
Query query = entity Manager
  .createQuery("SELECT b FROM Buch b")
  .setFirstResult(100)
  .setMaxResults(50);
@SuppressWarnings("unchecked")
List<Buch> buecher =
  query.getResultList();
for (Buch buch: buecher) {/*...*/}
```

```
SELECT b FROM Buch b WHERE
b.vorgaenger = :buch
```

Beim Ausführen der Abfrage muss das Ausgangsbuch als Parameter übergeben werden. Wie man sieht, werden im JPQL-Statement keine Objekt-IDs miteinander verglichen, sondern die Objekte selbst. Für mehrwertige Beziehungen sieht die Abfrage deutlich anders aus. Hat man ein Nutzer-Objekt und möchte wissen, zu welcher Bibliothek es gehört, fragt man in JPQL:

```
SELECT b FROM Bibliothek b WHERE
:nutzer MEMBER OF b.nutzer
```

Einige O/R-Mapper erlauben hier unter Aufgabe der Portabilität auch die vereinfachte Abfragesyntax der einwertigen Beziehungen.

Objekt-Konstruktion

Die Art und Weise der Objektkonstruktion wird durch den *Select*-Teil der Abfrage bestimmt. Die bisherigen Abfragen haben sich in diesem Aspekt nicht unterschieden; die Kandidatenklasse der Selektion war immer die erste Klasse aus dem *From*-Teil der Abfrage. Dies muss nicht so ein. Möchte man alle Nutzer ermitteln, die gerade etwas ausgeliehen haben, so genügt

```
SELECT DISTINCT a.leiher FROM
Ausleihung a WHERE a.rueckgabedatum
IS NULL
```

Zu beachten ist, dass keine mehrwertigen Navigationen im *Select*-Teil Verwendung finden dürfen. Diese Einschränkung lässt sich aber durch die bereits bekannte Navigationssyntax umgehen. Alle Bewertungen für Bücher aus dem Verlag X erhält man über

```
SELECT bew FROM Buch b,
IN(b.bewertung) bew WHERE
b.verlag.name = :x
```

▶

▷ Fortsetzungszeichen

Um die Quelltextzeilen lesbarer zu gestalten, wurden diese nach Bedarf mit dem Fortsetzungszeichen ▷ umbrochen. Dieses Fortsetzungszeichen dürfen Sie bei der Eingabe der jeweiligen Anweisung nicht mit eingeben. Es dient lediglich als Verkettungssymbol mit der jeweils nachfolgenden Zeile. Das ▷-Verkettungssymbol wird nicht genutzt, wenn Quelltextanweisungen bereits aufgrund der in den Anweisungen vorhandenen Leerzeichen direkt umbrochen werden können.

oder über

```
SELECT bew FROM Buch b JOIN b.bewertung
bew WHERE b.verlag.name = :x
```

Selektionen können prinzipiell auch als

```
SELECT OBJECT(b) FROM Buch b
```

geschrieben werden. Dies ist ein syntaktisches Relikt aus der Vorgängersprache EJBQL der EJB2-Spezifikation. Möchte man bestimmte Felder laden, die dem Objektmodell zufolge nichts miteinander zu tun haben, bietet sich ein Selektionskonstruktor an. Alle erfassten Kombinationen aus Autor und Verlag bekommt man so über

```
SELECT NEW
buchverwaltung.Engagement(a.name,
b.verlag.name) FROM Buch b,
IN(b.autoren) a
```

Die Klasse *Engagement* muss hierzu lediglich einen geeigneten Konstruktor

```
Engagement(String autor, String
verlag)
```

besitzen. Über den Selektionskonstruktor lassen sich auch Objekte des Objektmodells erzeugen, falls sie einen entsprechenden Konstruktor aufweisen. Exemplare, die nur den Standort gesetzt haben, kann man erzeugen über

```
SELECT NEW
buchverwaltung.Exemplar(e.id,
e.standort) FROM Exemplar e
```

Zu beachten ist hier unbedingt, dass so konstruierte Objekte nicht als geladen, sondern als neu betrachtet werden. Sie stehen deshalb nach dem Erzeugen nicht unter der Kontrolle des Persistenz-Frameworks. Mit Hilfe von JPQL lassen sich auch Tupel erzeugen. Die Zuordnung von Autor und Verlag kommt dann auch ohne die Zwischenobjekte der Klasse *Engagement* aus.

```
SELECT a.name, b.verlag.name FROM Buch
b, IN(b.autoren) a
```

erzeugt eine Ergebnisliste, deren Einträge vom Typ eines Arrays sind. Eines dieser Arrays besteht in dem Fall aus zwei Feldern; das erste Feld entspricht dem Namen des Autors, das zweite Feld dem Namen des Verlags. Mit dieser Variante ist es ebenso möglich, ganze Objekte zu laden:

```
SELECT a, b.verlag FROM Buch b,
IN(b.autoren) a
```

In diesem Fall stehen die geladenen Objekte, also der Inhalt des Arrays, unter der Kontrolle des Persistenz-Frameworks. Ein wesentlicher Vorteil von JPQL gegenüber reinen SQL-Abfragen ist die Tatsache, dass man sich keine Gedanken darüber machen muss, wie und woher die Objekte letztendlich geladen werden. Das Stichwort hier heißt Polymorphie. Eine Abfrage nach der abstrakten Klasse *Publikation* der Form

```
SELECT p FROM Publikation p
```

wird sowohl Objekte der Klasse *Buch* als auch Objekte der Klasse *Zeitschrift* hervorbringen, unabhängig davon, in welchen Datenbanktabellen sich die Informationen befinden. Die Abfrage

```
SELECT DISTINCT e.publikation FROM
Exemplar e
```

wird ebenfalls Buch- und Zeitschriften-Objekte aus der Datenbank laden, genau so, wie die Referenz von Exemplar auf Publikation ursprünglich gesetzt war.

Paging

Bisher konnte man noch nicht sehen, wie JPQL-Abfragen innerhalb von Java-Code ausgeführt werden. Um dies nachzuholen, soll das Paging

Subqueries

(1) nie auf Deutsch

```
SELECT v FROM Verlag v WHERE NOT EXISTS (
   SELECT p FROM Publikation p WHERE p.verlag = v AND p.sprache = :sprache
)
```

(2) manchmal auf Deutsch

```
SELECT v FROM Verlag v WHERE :sprache = SOME (
   SELECT p.sprache FROM Publikation p WHERE p.verlag = v
)
```

Semantisch äquivalent ist die Abfrage:

```
SELECT v FROM Verlag v WHERE EXISTS (
   SELECT p FROM Publikation p WHERE p.verlag = v AND p.sprache = :sprache
)
```

oder noch kürzer:

```
SELECT v FROM Verlag v, IN(v.publikationen) p WHERE p.sprache = :sprache
```

(3) immer auf Deutsch

```
SELECT v FROM Verlag v WHERE :sprache = ALL (
  SELECT p.sprache FROM Publikation p WHERE p.verlag = v
)
```

„Immer auf Deutsch“ ist äquivalent zu „nie in einer anderen Sprache als Deutsch“:

```
SELECT v FROM Verlag v WHERE NOT EXISTS (
  SELECT p FROM Publikation p WHERE p.verlag = v AND p.sprache <> :sprache
)
```

Für den Parameter *sprache* muss das Enumerationliteral *Sprache.DEUTSCH* beim Ausführen der Abfragen übergeben werden.

der Ergebnismenge demonstriert werden (vergleiche Listing 1, Seite 39. Anders als in SQL erfolgt die seitenweise Abfrage der Ergebnismenge über das API und nicht über Query-Schlüsselwörter wie *LIMIT*, *OFFSET* oder *ROWNUM*. Wird eine Abfrage in JPQL gestellt, so lädt der O/R-Mapper normalerweise die komplette Ergebnismenge in den Arbeitsspeicher. Dies kann bei vielen Datensätzen zu Speicherproblemen in der Anwendung führen. Über die Methode

```
Query#setMaxResults(int)
```

lässt sich Anzahl der zu ladenden Objekte begrenzen. Die Methode

```
Query#setFirstResult(int)
```

hingegen bestimmt, wie viele Datensätze übersprungen werden, bevor das erste Objekt geladen wird. Mit Hilfe beider Methoden lässt sich ein einfaches Paging realisieren. Zu beachten ist jedoch, dass beide Methoden datensatzorientiert und nicht objektorientiert arbeiten – mit der Konsequenz, dass Joins zu mehrwertigen Beziehungen innerhalb der Abfrage die Zählweise stören. JPA-Frameworks bieten über Query-Hints ausgefeilte Möglichkeiten der Cursor-Steuerung an.

Subqueries

JPQL spielt sicherlich seine Stärke im Bereich der Subqueries aus. Bei JPA 1.0 dürfen Subqueries sowohl im *WHERE* als auch im *HAVING*-Teil der Abfrage vorkommen. Man unterscheidet hier zwischen atomaren Ergebnissen, mehrwertigen Ergebnissen und Existenz-Quantoren. Liefert eine Subquery ein atomares Ergebnis zurück, kann dieses für arithmetische Ausdrücke verwendet werden. Das Buch mit den meisten Seiten erhält man durch die Abfrage

```
SELECT b FROM Buch b WHERE b.seiten =
(SELECT MAX(bs.seiten) FROM Buch bs)
```

Mehrwertige Teilergebnisse können über einen *IN*-Ausdruck einbezogen werden. Alle Exemplare, die vom Typ *Buch* sind, erhält man über

```
SELECT e FROM Exemplar e WHERE
e.publikation IN(SELECT b FROM Buch b)
```

Keine Subquery im Sinne von JPQL, jedoch im Sinne von SQL, bilden die Ausdrücke *IS EMPTY* und *SIZE*. Mit Hilfe von *IS EMPTY* lässt sich abfragen, ob eine mehrwertige Assoziation leer ist:

```
SELECT e FROM Exemplar e WHERE
e.ausleihungen IS EMPTY
```

liefert alle Exemplare, die noch nie ausgeliehen wurden. In ähnlicher Weise erhält man alle Buchexemplare, die mindestens zehnmal ausgeliehen wurden durch

```
SELECT e FROM Exemplar e
WHERE SIZE(e.ausleihungen) >= 10
```

Den interessantesten und kompliziertesten Teil der Subqueries bilden die Existenz-Quantoren. JPQL definiert dafür *EXISTS*, *ALL*, *ANY* und *SOME*. *ANY* und *SOME* sind semantisch gleichwertig. Mit diesen Schlüsselwörtern lassen sich logische Bedingungen der Form „für alle“, „für einige“ und „für keine“ formulieren. Der Textkasten „Subqueries“ stellt die Beispiel-Abfragen dar, die sich aus der Aufgabe ergeben: Selektiere alle Verlage, die (1) nie auf Deutsch publizieren, (2) manchmal auf Deutsch publizieren beziehungsweise (3) immer auf Deutsch publizieren. Hinsichtlich des Schlüsselworts *ALL* ist noch ein Wort der Warnung angebracht: Entgegen der Intuition liefert dieser Ausdruck auch dann den Wert *wahr* zurück, wenn die Ergebnismenge der Subquery leer ist. Dieselbe Einschränkung gilt übrigens auch auf SQL-Ebene. Besteht die Gefahr einer leeren Ergebnismenge, so muss dies zusätzlich abgeprüft werden. Im Fall der auf Deutsch publizierenden Verlage hieße dies:

```
SELECT v FROM Verlag v WHERE
:sprache = ALL (...) AND
v.publikationen IS NOT EMPTY
```

Aggregatfunktionen

Obwohl JPQL in erster Linie zur Selektion von Objektmengen dient, bietet es dennoch eine gute Auswahl von Aggregatfunktionen. Zu diesen gehören: *AVG*, *MIN*, *MAX*, *SUM* und *COUNT*. Die Anzahl der Bücher innerhalb der Datenbank erhält man unter Anwendung der Aggregatfunktion *COUNT* durch die einfache Abfrage

```
SELECT COUNT(b) FROM Buch b
```

Interessanter ist schon die Frage nach der Anzahl von Exemplaren pro Publikation:

```
SELECT e.publikation, COUNT(e) FROM
Exemplar e GROUP BY e.publikation
```

Unter Erhöhung des Schwierigkeitsgrads könnte man danach nach allen Büchern fragen, die ▶

JPA 2.0

Die Spezifikation des Nachfolgestandards JPA 2.0 (JSR 317) steht unmittelbar vor seiner Verabschiedung. Zwar nicht wie geplant noch im Jahr 2008, aber spätestens bis zur Java One im Juni 2009 wird der Persistenz-Standard eine Vielzahl kleinerer Verbesserungen erfahren. Neben Erweiterungen des Mappings für zusammengesetzte Primärschlüssel, Maps und Listen und wenigen Ergänzungen des APIs, insbesondere hinsichtlich pessimistischer Sperren, hat auch die Abfragesprache wichtige Ergänzungen erfahren. Der weitaus interessanteste Aspekt von JPA 2.0 ist jedoch die Criteria Query API. Statt zu formulieren:

```
SELECT b FROM Buch b WHERE
 b.seiten BETWEEN 350 AND 600
```

kann nun über einen Expression-Builder die Abfrage programmatisch erstellt werden. Die Entsprechung wird in JPA 2.0 wie folgt aussehen:

```
EntityManager em = ...;
QueryBuilder qb =
 em.getQueryBuilder();
DomainObject b =
qb.createQueryDefinition▷
 (Buch.class);
List<Buch> buecher =
em.createQuery(b.select().▷
 where( b.get("seiten").▷
 between(350, 600))).▷
 getResultList();
```

Bulk-Alternativen

Sollten die Möglichkeiten der Bulk-Operationen nicht ausreichen, kann problemlos auf native SQL-Statements oder gar Stored Procedures ausgewichen werden. Da auch die JPQL-Bulk-Operationen vollständig an der Java-Anwendung vorbei ausgeführt werden, würde man sich durch natives SQL nicht schlechter stellen. Nur die Ermittlung der korrekten Tabellennamen müsste man in diesem Fall selbst übernehmen.

durchschnittlich mit mehr als drei Sternen bewertet wurden. Unter der Voraussetzung, dass die Enumeration *Sterne* als Ordinalwert gespeichert wird, sieht das JPQL-Statement so aus:

```
SELECT b FROM Buch b JOIN
b.bewertungen r GROUP BY b HAVING
AVG(r.sterne) > 3
```

Für JPQL gilt wie für SQL: Die Reihenfolge der Auswertung ist immer *FROM*, *WHERE*, *GROUP BY*, *HAVING*. *HAVING* ist damit der letzte Schritt, der Einfluss auf die Ergebnismenge nehmen kann. Aggregatfunktionen können neben dem *SELECT*- und *HAVING*-Teil auch über eine Subquery im *WHERE*-Teil der Abfrage vorkommen. Das Buch mit den wenigsten Seiten erhält man mit diesem JPQL-Statement:

```
SELECT b FROM Buch b WHERE b.seiten
= (SELECT MIN(bs.seiten) FROM Buch bs)
```

Auch zu den Aggregatfunktionen noch ein warnender Hinweis: In jedem Fall ist eine Plausibilitätsprüfung des generierten SQL-Statements angebracht. Ein aufwendiger Filter im *WHERE*-Teil der Abfrage kann zu zusätzlichen SQL-Joins führen. Wird dabei entlang mehrwertiger Assoziationen navigiert, entsteht durch die Kreuzproduktbildung ein künstlich vergrößertes Result-Set. Aggregatfunktionen auf diesem Result-Set sind dann nicht mehr plausibel. Wird beispielsweise nach der Gesamtzahl an vergebenen Sternen für ein Buch gefragt, so liefert dieses JPQL-Statement die passende Antwort:

```
SELECT b, SUM(r.sterne) FROM Buch b
JOIN b.bewertungen r GROUP BY b
```

Die Bezugnahme auf einen Autor in der Form

```
SELECT b, SUM(r.sterne) FROM Buch b
JOIN b.bewertungen r, IN(b.autoren)
a WHERE a.name LIKE 'M%' GROUP BY b
```

führt zu einem falschen Ergebnis, wenn ein Buch mehrere Autoren hat, deren Namen alle mit dem Buchstaben „M“ anfangen. In diesem Fall hilft es nur, den störenden Join in eine Subquery auszulagern.

Bulk-Operationen

Neben den vorgestellten *SELECT*-Abfrageoperationen kennt JPQL auch die Möglichkeit der Massendatenverarbeitung. Ohne Objekte in die Anwendung laden zu müssen, sind auf diese Weise sehr viele Datensätze direkt in der Datenbank änderbar oder löschbar. Die Entfernung aller abgelaufenen Ausleihinformationen geschieht beispielsweise über das folgende JPQL-Statement:

```
DELETE FROM Ausleihung a WHERE
a.rueckgabedatum IS NOT NULL
```

Auch eine Massenänderung ist möglich. Um einen bestimmten Standort umzubenennen, reicht das JPQL-Statement

```
UPDATE Exemplar e SET e.standort =
'Büro' WHERE e.standort = 'Diele'
```

Interessant wird es, wenn navigiert werden muss. Dieser Anwendungsfall ist in der Spezifikation von JPA nicht vorgesehen, so dass man sich über eine Subquery behelfen muss. Möchte man alle Nutzer sperren, die eine überfällige Ausleihung aufweisen, kann man formulieren:

```
UPDATE Nutzer n SET n.status =
buchverwaltung.Status.GESPERRT WHERE
EXISTS(SELECT a FROM Ausleihung a
WHERE a.rueckgabedatum IS NULL AND
a.faelligkeitsdatum < CURRENT_DATE AND
a.leiher = n)
```

Fazit

JPQL ist eine mächtige Abfragesprache. Ihre syntaktische Nähe zu SQL macht einen Umstieg von nativen SQL-Abfragen auf das Java Persistence API sicherlich deutlich einfacher. Trotzdem besteht ein kleiner Wermutstropfen: Der objektorientierte Aspekt ist an manchen Stellen zu schwach ausgeprägt. Somit leckt sprichwörtlich die Abstraktion, und einige Besonderheiten der SQL-Abfragesprache treffen den unbekümmerten JPQL-Anwender.

Die effektivste Verbesserung der Abfragesprache wäre sicherlich die Aufgabe der Join-Syntax zugunsten der Punkt-Notation für Objektnavigationen. Das statische Mapping würde diesen Ansatz sogar unterstützen, da dort hinterlegt ist, ob eine Beziehung optional oder verpflichtend ist. Die Abwägung zwischen Inner- und Left-Outer-Joins würde dann der Vergangenheit angehören. Obwohl die Operationen und Ausdrucksmöglichkeiten der JPA-Abfragesprache bereits sehr mächtig sind, gibt es immer Verbesserungsmöglichkeiten. Welche davon der Nachfolgestandard JPA 2.0 aufzubieten hat, soll in einem weiteren Artikel gezeigt werden. **[am]**

[1] JPQL-Sprachreferenz; http://edocs.bea.com/kodo/docs41/full/html/ejb3_langref.html
[2] jpox – JPQL und JPQL-Abfragen; www.jpox.org/docs/1_2/jpa/jpql.html sowie www.jpox.org/docs/1_2/jpa/query.html
[3] O/R-Mapping im Detail; www.agiledata.org/essays/mappingObjects.html