



Optimierung von JPA-Anwendungen

Dirk Weil | GEDOPLAN

Dirk Weil

- GEDOPLAN GmbH, Bielefeld
- Java EE seit 1998
- Konzeption und Realisierung
- Vorträge
- Seminare
- Veröffentlichungen



Optimierung von JPA-Anwendungen

Laufzeit

Memory

Providerunabhängig

EclipseLink

Hibernate

...

Id-Generierung

- Entity-Klassen müssen Id haben

- PK in der DB
- Feld oder Property mit **@Id**

```
@Entity
public class SomeEntity
{
    @Id
    private int id;
    ...
}
```

- Empfehlenswert: Technische Id
 - Problem: Erzeugung eindeutiger Werte

Id-Generierung

- JPA-Feature: `@GeneratedValue`
 - Nutzt DB-Sequenzen, Identity Columns oder Sequenz-Tabellen
- Probleme:
 - Id erst nach `persist` gesetzt
→ `equals?`, `hashCode?`
 - Id-Übernahme kostet Zeit

```
@Id
@GeneratedValue
private int id;
```

Id-Generierung

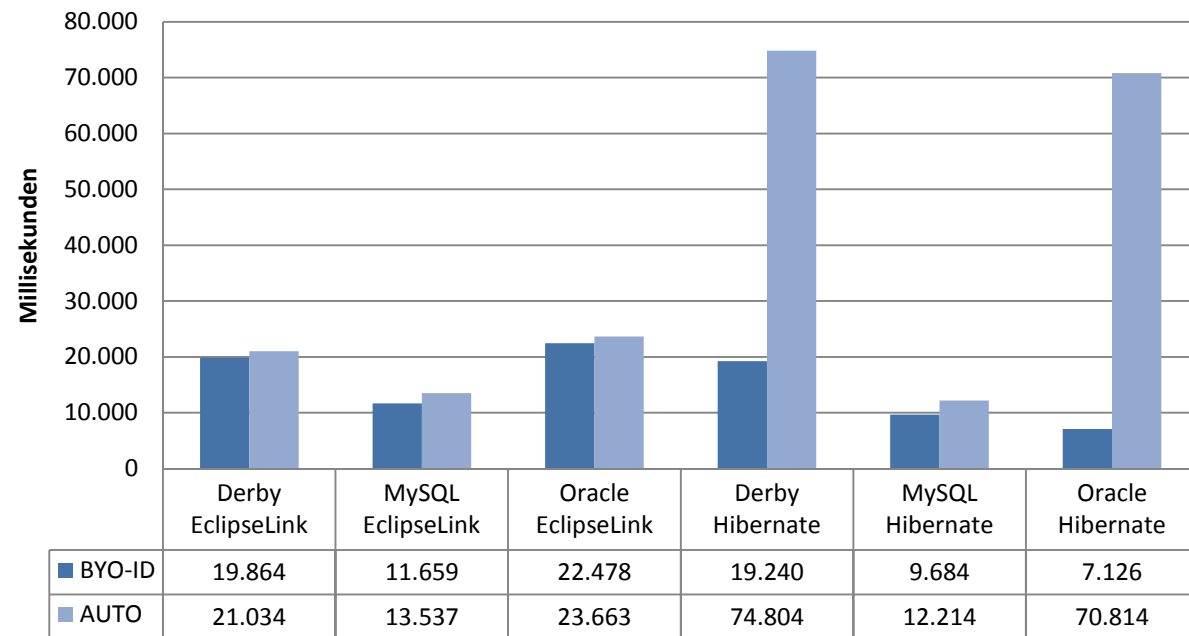
- Alternative: BYO-ID (selbst machen)
 - Id auch in transitiven Objekten gesetzt
 - Insert ohne Zusatzaufwand
 - Achtung: i. A. nicht trivial
- Z. B.: UUID

```
@Id
private String id
    = new com.eaio.uuid.UUID().toString();
```

Id-Generierung

- `@GeneratedValue` signifikant langsamer (OOTB)

Insert 50.000 einfache Entries
in 1.000er Batches



Id-Generierung

- Tuning: Höhere Allocation Size

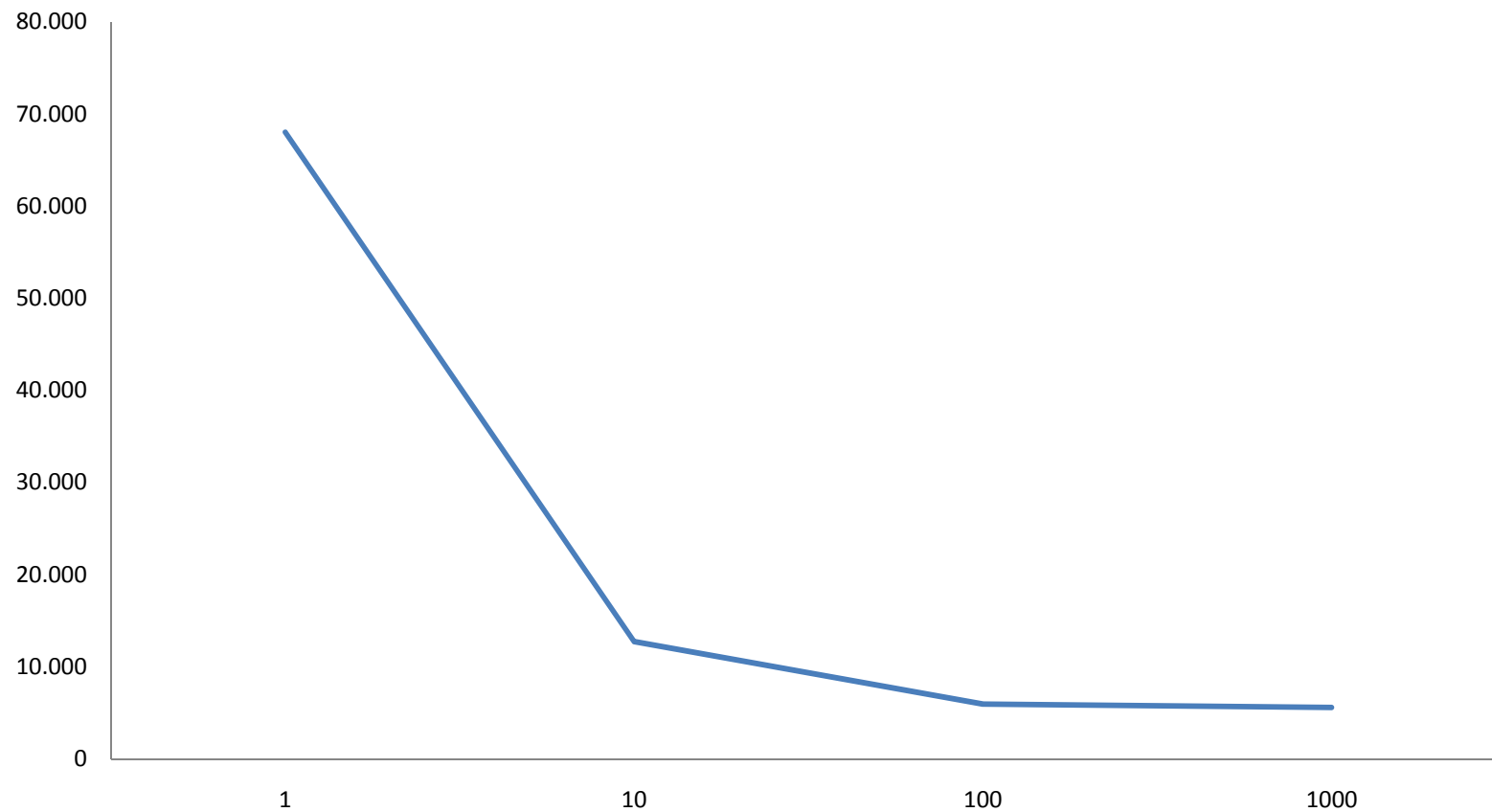
```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE,
                 generator = "ArtikelIdGenerator")
@SequenceGenerator(name = "ArtikelIdGenerator",
                    allocationSize = 1000)
```

```
privat @Id
@GeneratedValue(strategy = GenerationType.TABLE,
                 generator = "ArtikelIdGenerator")
@TableGenerator(name = "ArtikelIdGenerator",
                allocationSize = 1000)
private int id;
```

- Leider nicht verfügbar bei **IDENTITY**

Id-Generierung

Laufzeit vs. Allocation Size

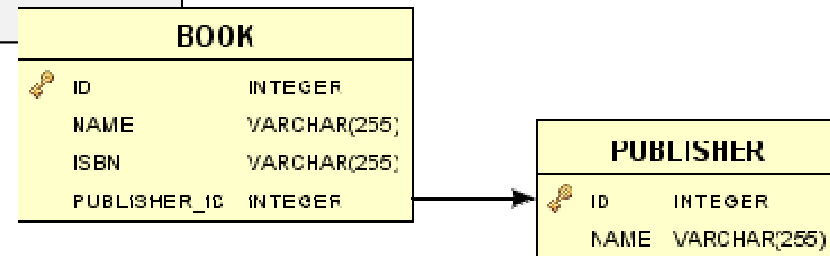


Relationship Loading

- Relationen werden durch Felder mit **@OneToOne**, ..., **@ManyToMany** repräsentiert

```
@Entity
public class Book
{
    @ManyToOne
    public Publisher publisher;
```

```
@Entity
public class Publisher
{
    @OneToMany(mappedBy="publisher")
    public List<Book> books;
```



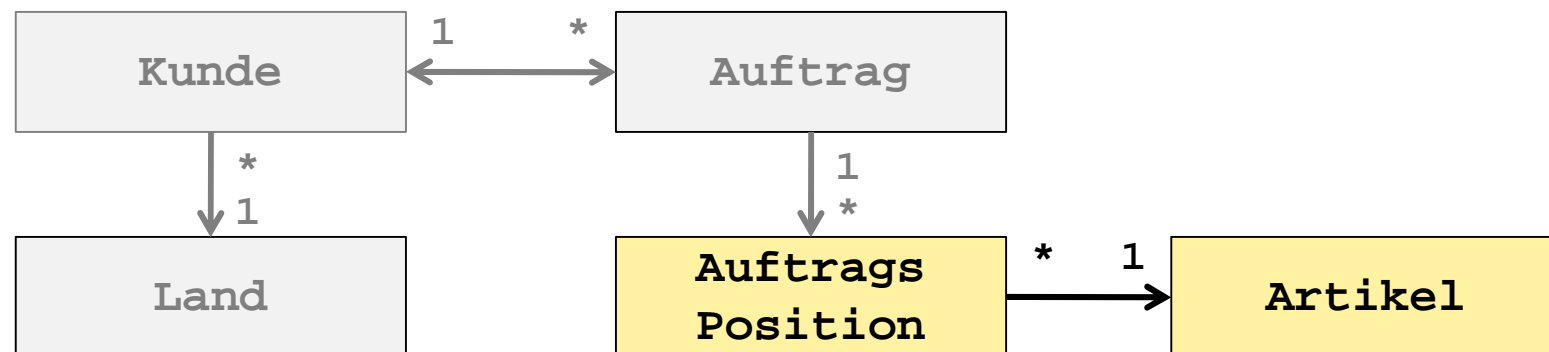
Relationship Loading

- Relationen-Parameter: **fetch**
- Referenzierte Entities direkt laden?
 - **EAGER**: Direkt
 - **LAZY**: Später bei Bedarf

```
@ManyToOne(fetch = FetchType.LAZY)  
private Artikel artikel;
```

Relationship Loading

- Bsp.: Auftragsposition bearbeiten
 - Ist Owner der n:1-Relation zu **Artikel**



```
@Entity
public class AuftragsPosition
{
    @ManyToOne
    private Artikel artikel;
```

Relationship Loading

- Annahme:
Verwendet nur
AuftragsPosition

```
AuftragsPosition aufPos  
= em.find(AuftragsPosition.class, id);  
...
```

```
@ManyToOne  
private Artikel artikel;
```

EAGER

```
select ...  
from AuftragsPosition  
left outer join Artikel  
where ...
```

```
@ManyToOne(fetch=FetchType.LAZY)  
private Artikel artikel;
```

LAZY

```
select ...  
from AuftragsPosition  
where ...
```

Relationship Loading

- Annahme:
Verwendet auch
Artikel

```
AuftragsPosition aufPos  
= em.find(AuftragsPosition.class, id);  
Artikel artikel = aufPos.getArtikel();  
...
```

```
@ManyToOne  
private Artikel artikel;
```

EAGER

```
select ...  
from AuftragsPosition  
left outer join Artikel  
where ...
```

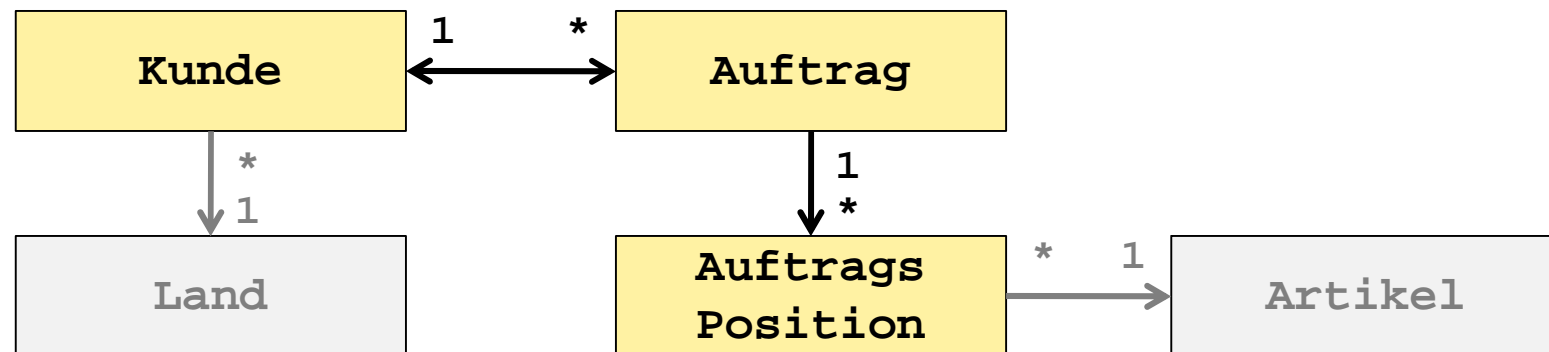
```
@ManyToOne(fetch=FetchType.LAZY)  
private Artikel artikel;
```

LAZY

```
select ... from AuftragsPosition where ...  
select ... from Artikel where ...
```

Relationship Loading

- Bsp.: Kunde bearbeiten
 - Ist Owner der 1:n-Relation zu Auftrag



```
@Entity
public class Kunde
{
    @OneToMany(mappedBy="kunde")
    private Set<Auftrag> auftraege;
```

Relationship Loading

- Annahme:
Verwendet
nur Kunde

```
Kunde kunde  
= em.find(Kunde.class, id);...
```

```
@ManyToOne(fetch=FetchType.EAGER)  
private Set<Auftrag> auftraege;
```

```
@ManyToOne  
private Set<Auftrag> auftraege;
```

EAGER

LAZY

```
select ...  
from Kunde  
left outer join Auftrag  
left outer join AuftragsPosition  
where ...
```

```
select ...  
from Kunde  
where ...
```

Relationship Loading

- Messergebnis
(1000 Interaktionen, Hibernate, MySQL)

	EAGER	LAZY	
Nur AuftragsPosition	2.967 ms	2.505 ms	- 15 %
Auch Artikel	2.959 ms	4.305 ms	+ 45 %
Nur Kunde	30.295 ms	4.848 ms	- 84 %

 = Default-Einstellung

Relationship Loading

- Fazit:
 - Zugriffsverhalten genau analysieren
 - Default ist schon recht gut
 - Besser: Immer LAZY verwenden und bei Bedarf Fetch Joins nutzen

Relationship Loading

- Fetch Joins mit JPQL
 - leider nur einstufig erlaubt

```
select ap from Auftragsposition ap
       left fetch join ap.artikel
       ...
```

Relationship Loading

- Fetch Joins mit Criteria Query

```
CriteriaQuery<Auftrag> cQuery
    = builder.createQuery(Auftrag.class);
Root<Auftrag> a
    = cQuery.from(Auftrag.class);

a.fetch(Auftrag_.auftragsPositionen)
  .fetch(AuftragsPosition_.artikel);

...
```

Basic Attribute Loading

- Fetch-Strategie auch für einfache Werte wählbar

```
@Basic(fetch = FetchType.LAZY)  
private String longAdditionalInfo;
```

- Lazy Loading sinnvoll bei
 - selten genutzten Werten
 - umfangreichen Daten

Basic Attribute Loading

- Messergebnis
 - Lesen von Kunden
 - 10 'ungenutzte' Strings à 150 chars
 - 1000 Iterationen, EclipseLink, Oracle

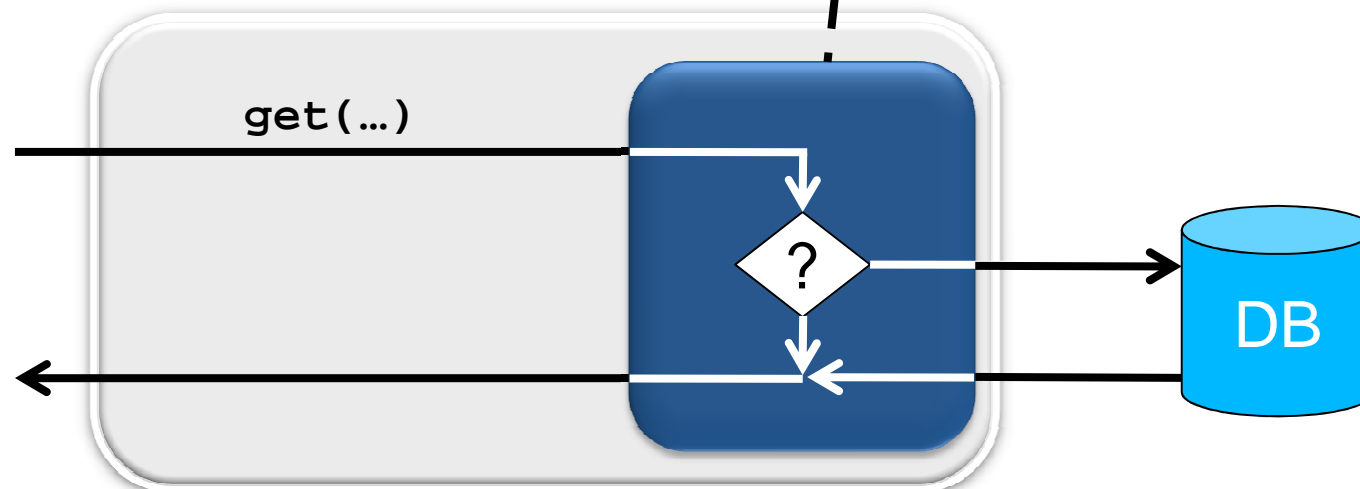
EAGER	LAZY	
7.204 ms	6.820 ms	-5 %

 = Default-Einstellung

Lazy-Load-Verfahren

- Proxy

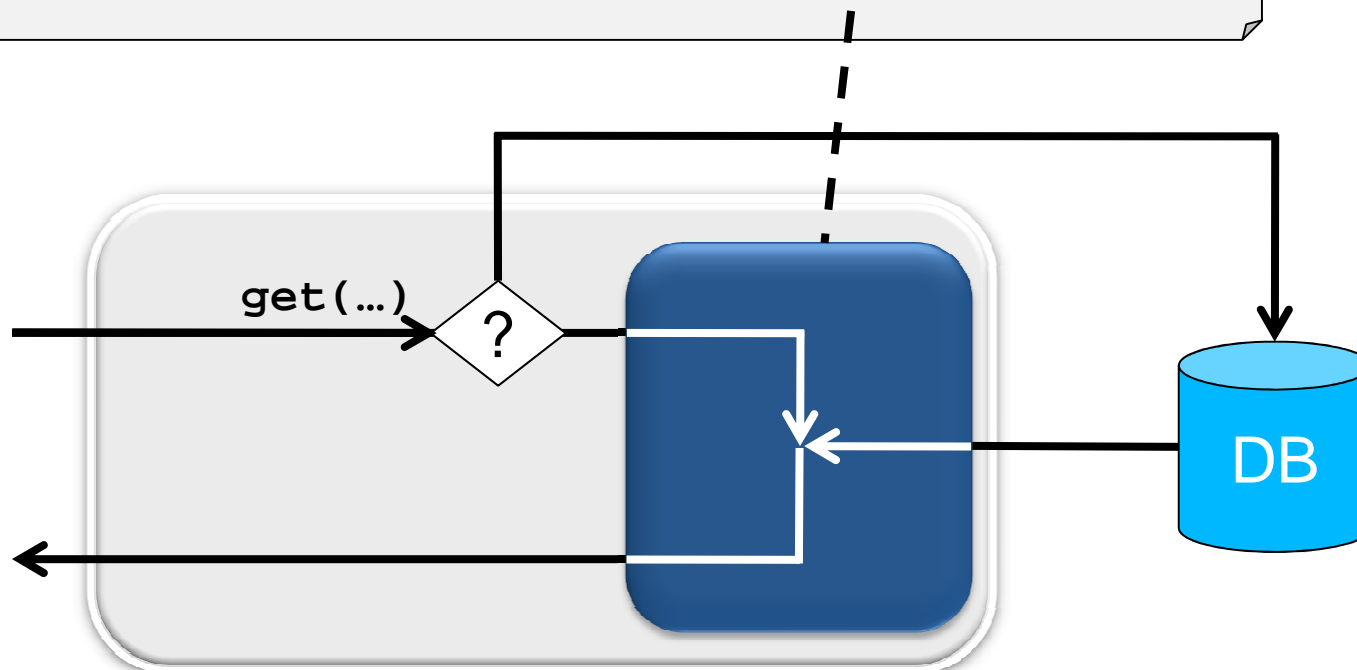
```
@OneToMany  
private Set<Auftrag> auftraege
```



Lazy-Load-Verfahren

- Instrumentierung

```
@Basic(fetch = FetchType.LAZY)  
private String longAdditionalInfo;
```



Bytecode-Instrumentierung

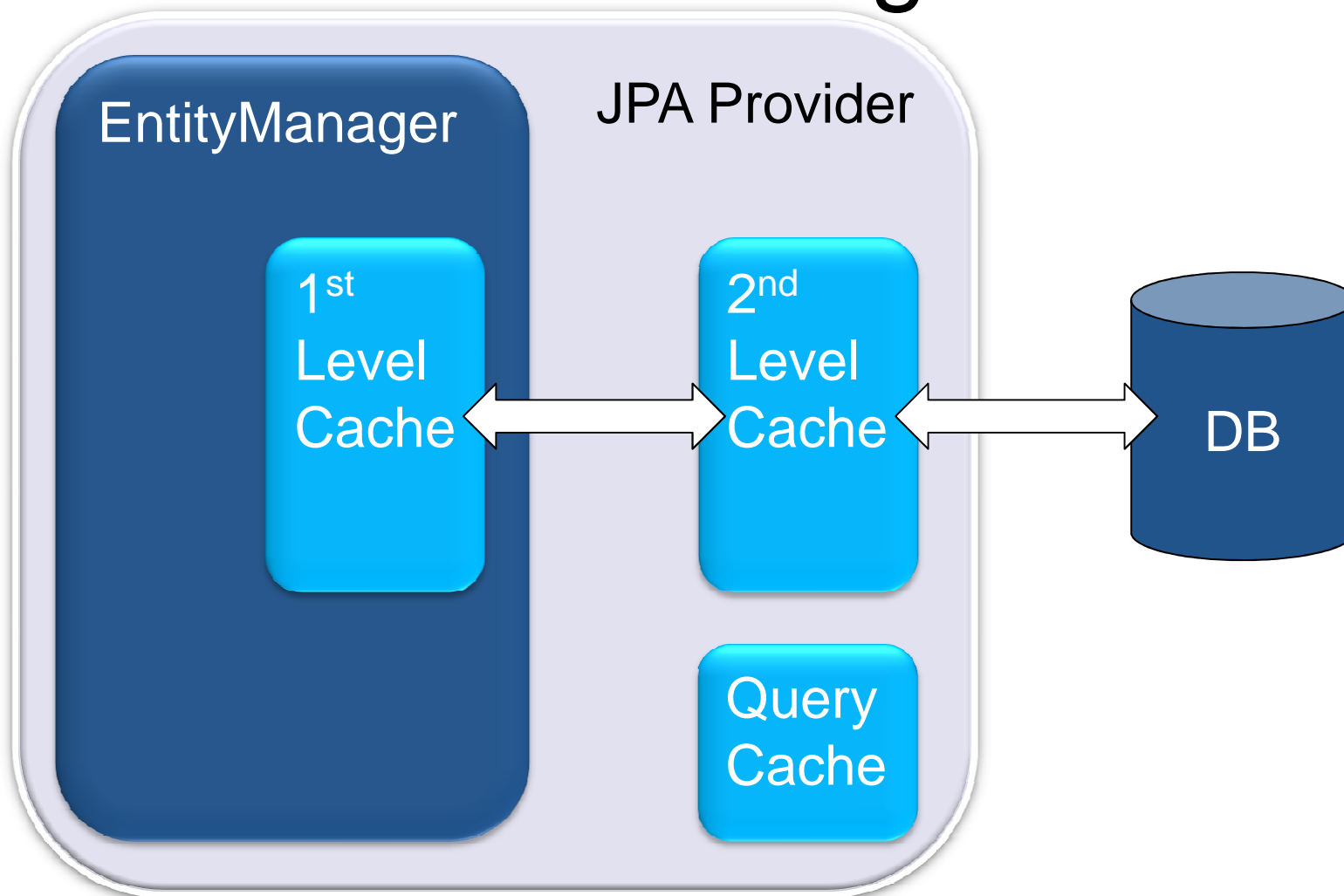
Eclipselink			
	Verfahren	SE	EE
@Basic	Entity Instrumentation	✘	✓
@xxxToOne	Entity Instrumentation	✘	✓
@xxxToMany	Collection Proxy	✓	✓

Hibernate			
	Verfahren	SE	EE
@Basic	Entity Instrumentation	✘	✘
@xxxToOne	Attribute Proxy	✓	✓
@xxxToMany	Collection Proxy	✓	✓

✓ = Standard

✘ = Providerspezifische Konfiguration erforderlich

Caching



First Level Cache

- Standard
- Je EntityManager
- Enthält in Sitzung geladene Objekte
 - Achtung: Speicherbedarf!
 - ggf. explizit entlasten (`clear`, `detach`)



First Level Cache

- Arbeitet
sitzungs-
bezogen

```
// Kunden mit bestimmter Id laden
EntityManager em1 = emf.createEntityManager();
Kunde k1 = em1.find(Kunde.class, id);

// Gleichen Kunden in 2. Session verändern
EntityManager em2 = emf.createEntityManager();
em2.getTransaction().begin();
Kunde k2 = em2.find(Kunde.class, id);
k2.setName("...");
em2.getTransaction().commit();

// Gleichen Kunden in 1. Session erneut laden
Kunde k3 = em1.find(Kunde.class, id);
// ist unverändert!
```

First Level Cache

- HashMap-Semantik
 - benötigt Key
 - wird für Queries nicht benutzt

```
// Kunden mit bestimmter Id laden
EntityManager em = emf.createEntityManager();
Kunde k1 = em.find(Kunde.class, id);

// Query nach gleichem Kunden geht erneut zur DB!
Kunde k2 = em.createQuery("select k from Kunde k " +
                        "where k.id=:id", Kunde.class)
            .setParameter("id", id)
            .getSingleResult();
```

Query Cache

- Provider-spezifisch
- Speichert Result Set IDs zu Queries

```
TypedQuery<Kunde> query
= em.createQuery("select k from Kunde k where k.name=:name",
                Kunde.class);
query.setParameter("name", "OPQ GbR");
... // Query Cache einschalten
Kunde kunde = query.getSingleResult();
```

```
["select k from Kunde k where k.name=:name", "OPQ GbR"] → [id1]
```

Query Cache

- Trotz mehrfacher Query nur ein DB-Zugriff

```
while (...)
{
    TypedQuery<Kunde> query
        = em.createQuery("select k from Kunde k where k.name=:name",
                        Kunde.class);
    query.setParameter("name", "OPQ GbR");
    query.setHint(...) // Query Cache einschalten (providerabh.!)
    Kunde kunde = query.getSingleResult();
    ...
}
```

Query Cache

- EclipseLink

```
TypedQuery<Kunde> query = em.createQuery(...);  
query.setHint("eclipselink.cache-usage",  
             "CheckCacheThenDatabase");  
...
```

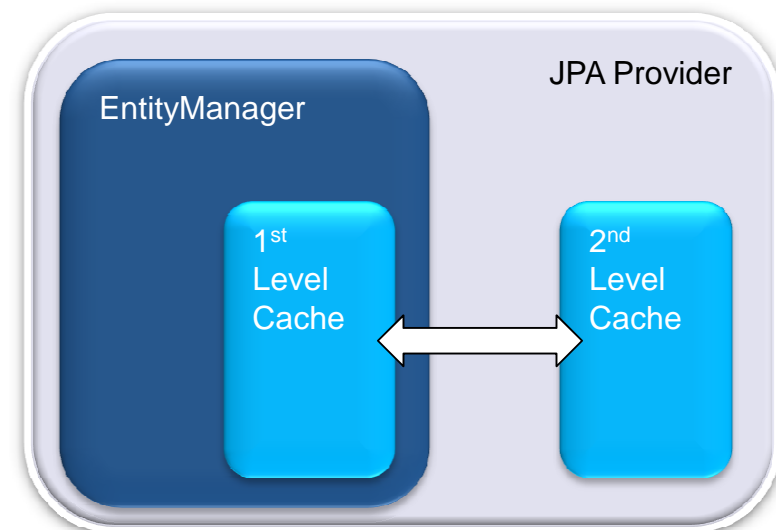
- Hibernate:

```
TypedQuery<Kunde> query = em.createQuery(...);  
query.setHint("org.hibernate.cacheable", true);  
...
```

(Aktivierung in der Konfiguration notwendig)

Second Level Cache

- JPA 2.0 unterstützt 2nd Level Cache
 - nur rudimentäre Konfiguration
 - Providerspezifische Konfiguration in der Praxis unabdingbar



Second Level Cache

- Providerspezifische Implementierung
 - Cache-Provider
EHCache, OSCache, ...
 - Cache-Strategien
read-only, read-write, ...
 - Storage
Memory, Disk, Cluster, ...

Second Level Cache

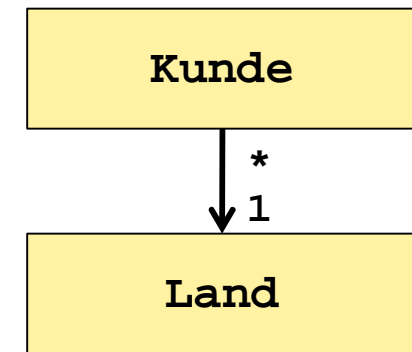
- Wirkt applikationsweit
- Semantik ähnlich **HashMap**
- Ladereihenfolge:
 - 1st Level Cache (**EntityManager**)
 - 2nd Level Cache, falls enabled
 - DB

Second Level Cache

- Vorteil bei häufig genutzten Daten
 - Konstanten
 - selten veränderte Daten
 - nur von dieser Anwendung veränderte Daten

Second Level Cache

- Bsp.: Stammdaten-Entity **Land**
 - wird n:1 von **Kunde** referenziert
 - nur wenige **Land**-Werte
 - Länder ändern sich nahezu nie
 - Länder können dauerhaft im Cache verbleiben

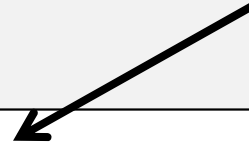


Second Level Cache

- Konfiguration It. Spec

```
<persistence-unit name="...">
  <provider>...</provider>
  <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
  ...
```

```
@Entity
@Cacheable(true)
public class Land
{
  ...
}
```



	Cache aktiv für ...
ALL	alle Entities
NONE	keine Klasse
ENABLE_SELECTIVE	nur @Cacheable(true)
DISABLE_SELECTIVE	alle außer @Cacheable(false)

Second Level Cache

- EclipseLink
 - Default: DISABLE_SELECTIVE
- Hibernate
 - ignoriert Standard-Konfig

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_ONLY)
public class Land
{
    ...
}
```

Second Level Cache

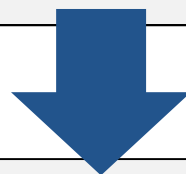
- Messergebnis
(1000 Iterationen, EclipseLink, Oracle)

ohne 2 nd Level Cache:	10.883 ms
mit 2 nd Level Cache für Land :	6.549 ms

Paginierung

- Queries mit großer Ergebnismenge 'häppchenweise' verarbeiten

```
TypedQuery<Artikel> query  
= em.createQuery("select a from Artikel a", Artikel.class);  
query.setFirstResult(50);  
query.setMaxResults(10);  
List<Artikel> result = query.getResultList();
```



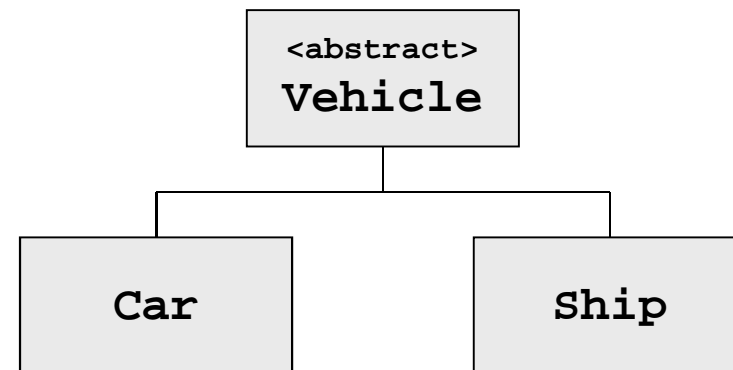
```
select ...  
from Artikel  
where ... and rownum>=50 and rownum<60
```

Paginierung

- Eingeschränkt oder ineffektiv bei 1:n/m:n-Relationen mit:
 - Eager Loading
 - Fetch Joins
- Join erzeugt kartesisches Produkt
- Providerabhängige Lösung:
 - Ausführung im Memory
 - Ausführung mehrerer SQL-Befehle

Inheritance


- Mehrere Abbildungen denkbar:
 - Alles in einer Tabelle
 - Eine Tabelle pro Klasse
 - Eine Tabelle pro konkreter Klasse
- Strategie-Auswahl mit **@Inheritance**




Inheritance

- **SINGLE_TABLE**

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public abstract class Vehicle
{
    ...
}
```

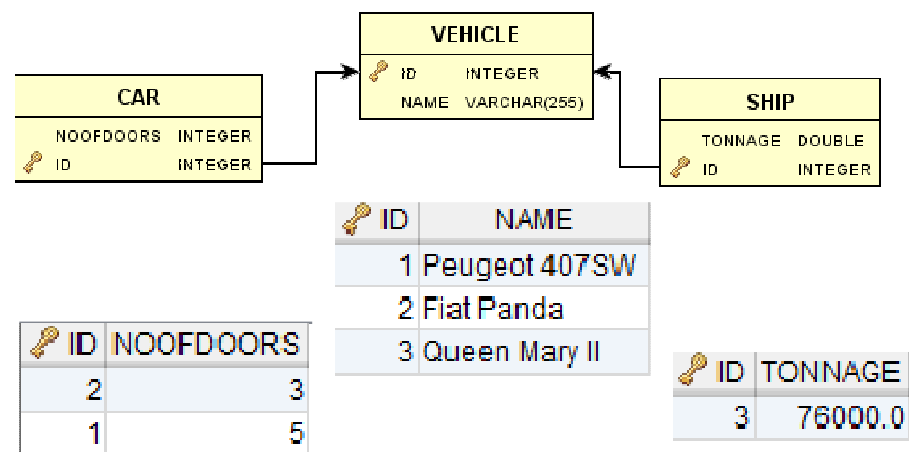
VEHICLE	
 ID	INTEGER
TYPE	VARCHAR(31)
NAME	VARCHAR(255)
TONNAGE	DOUBLE
NOOFDOORS	INTEGER

 ID	TYPE	NAME	TONNAGE	NOOFDOORS
652	C	Fiat Panda	(null)	3
651	C	Peugeot 407SW	(null)	5
653	S	Queen Mary II	76000.0	(null)

Inheritance

- **JOINED**


```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public abstract class Vehicle
{
    ...
}
```





Inheritance


- **TABLE_PER_CLASS**

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Vehicle
{
    ...
}
```

CAR		
 ID	INTEGER	
NAME	VARCHAR(255)	
NOOFDOORS	INTEGER	

SHIP		
 ID	INTEGER	
NAME	VARCHAR(255)	
TONNAGE	DOUBLE	

 ID	NAME	NOOFDOORS
701	Peugeot 407 SW	5
702	Fiat Panda	3

 ID	NAME	TONNAGE
703	Queen Mary II	76000.0

Inheritance

- Laufzeitvergleich für Queries
 - auf Basisklasse
 - auf abgeleitete Klasse

	SINGLE_ TABLE	TABLE_ PER_CLASS	JOINED
Basisklasse	2.705 ms	29.359 ms	3.434 ms
Subklasse	2.505 ms	1.435 ms	3.377 ms

- (1000 Iterationen, Ergebnis ca. 100 Einträge, Hibernate, MySQL)

Inheritance

- Optimale Performanz liefern
SINGLE_TABLE und
TABLE_PER_CLASS
- Aber: Auch andere Implikationen
- Genaue Analyse notwendig

Providerabhängiges

- Batch Size
- Lazy-Varianten
- Cache-Strategien
- Prepared Statement Cache

Providerabhängiges

- Load Groups
- Change Detection
- DB Dialects
- ...

Fazit

- Viele Optimierungen providerunabhängig möglich
- Wesentlich:
 - Lazy Loading
 - Caching
- Genaue Analyse notwendig
- Messen
- Kein Selbstzweck

The logo for 'jax 2011' features the word 'jax' in a bold, blue, lowercase sans-serif font, followed by '2011' in a bold, yellow, lowercase sans-serif font. The 'j' in 'jax' is stylized with a yellow dot above it and a yellow line extending from its top curve to another yellow dot. A similar yellow dot and line are positioned below the 'x'.

jax[®] 2011