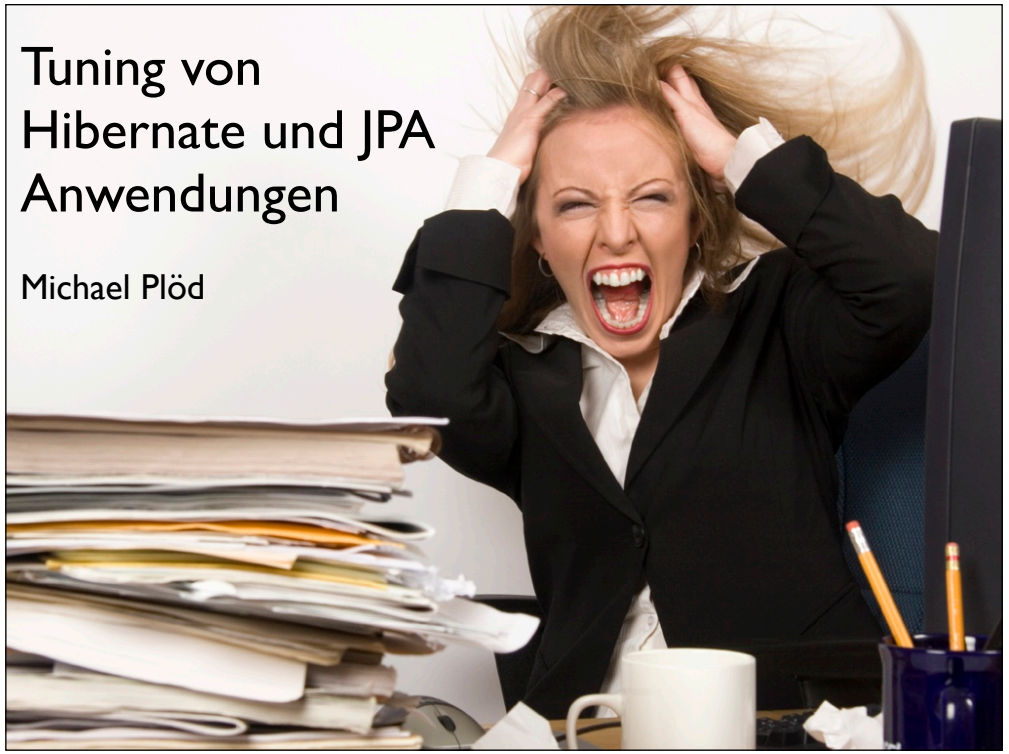
Ihre Persistenzschicht?

Wahrnehmung endet oft mit Fingerpointing richtung ORM
Unnützer Konflikt mit DBAs (wertvolle Hilfe beim Debugging)

Tuning von Hibernate und JPA Anwendungen

Michael Plöd



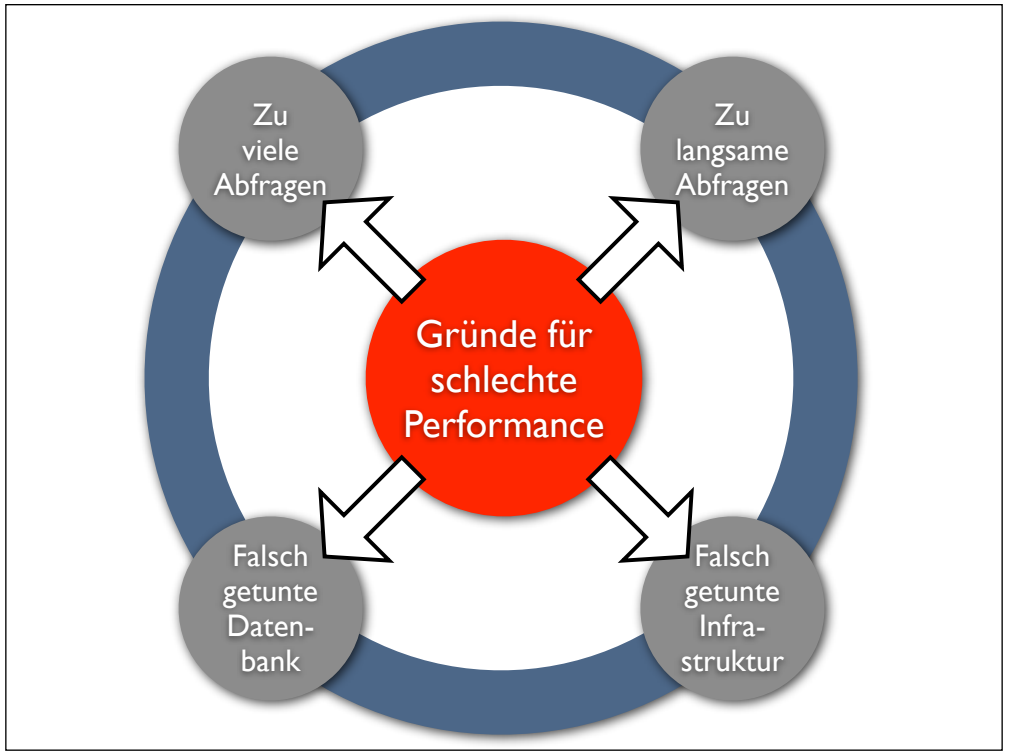
Michael Plöd

- Partner und Principal Architect bei Senacor Technologies AG in Nürnberg
- http://www.senacor.com
- michael.ploed@senacor.com
- Twitter: @bitboss

Jochen Mader

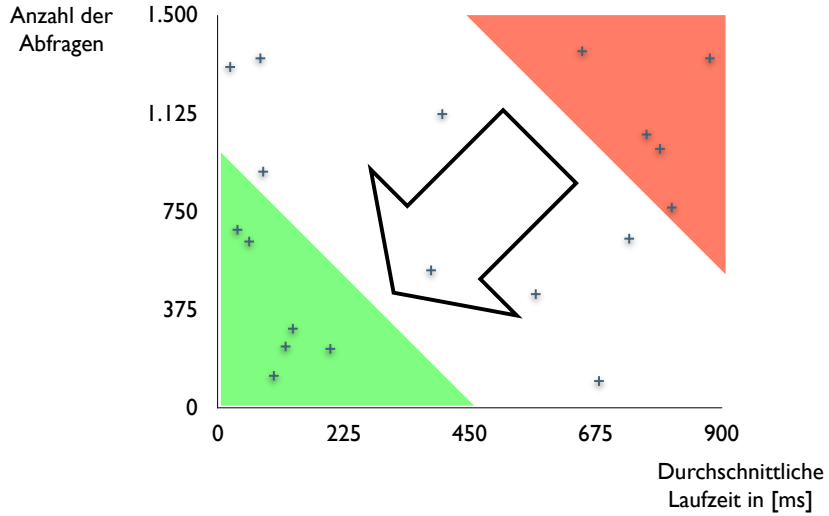
- Senior Developer bei Senacor Technologies AG in Nürnberg
- http://www.senacor.com
- jochen.mader@senacor.com
- Twitter: @codepitbull

Vieleicht
Mapping-Overhead
ABER
Nutzung von DB-Spezifikas die Entwickler nicht kennt
Transactional Write Behind
Caching
also ein bisschen

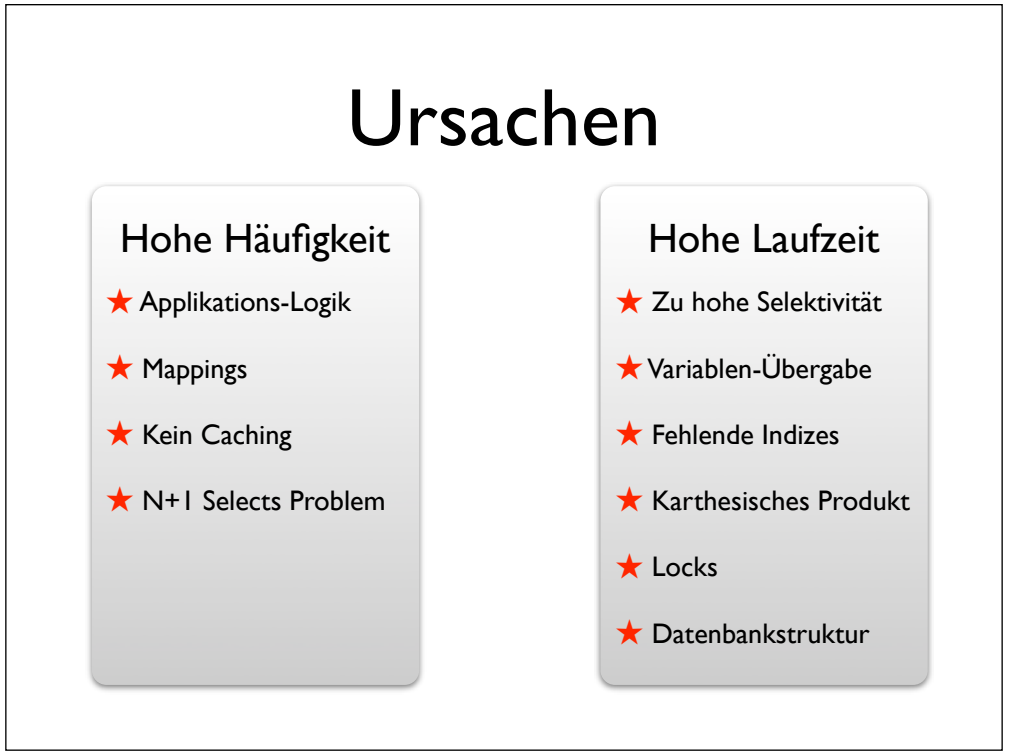


Nur die oberen 2 Thema des Vortrags

Klassifizierung von Abfragen

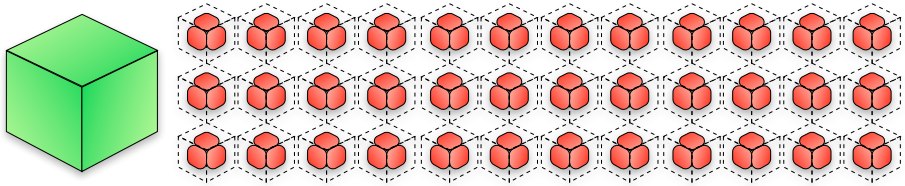


Messen
Klassifizieren
Optimieren



Variablen-Übergabe:
Number in DB, String im Mapping => Kein Index!
java.util.Date => Temporal Type

N+1 Selects Problem



```
List list = s.createCriteria(Konto.class).list();
for (Iterator it = list.iterator(); it.hasNext();) {
  Konto kto = (Konto) it.next();
  kto.getKunde().getName();
}
```

```
SELECT * FROM KONTEN                              +1
SELECT * FROM PERSONEN WHERE PERSON_ID = ?
SELECT * FROM PERSONEN WHERE PERSON_ID = ?        N
SELECT * FROM PERSONEN WHERE PERSON_ID = ?
...
```

```
@Entity
public class Konto {
  ...
 @ManyToOne(fetch=FetchType.LAZY)
 public Person getKunde() {...}
  ...
}
```

Karthesisches Produkt

```
@Entity
public class Konto {
 ...
 @OneToMany(fetch=FetchType.EAGER)
 public Set<Buchung> getBuchungen() {...}

 @OneToMany(fetch=FetchType.EAGER)
 public Set<Vollmacht> getVollmachten() {...}
 ...
}
```

```
select konto.*, buchung.*, vollmacht.*
  from KONTEN konto
  left outer join BUCHUNGEN buchung
     on konto.ID = buchung.KTO_ID
  left outer join VOLLMACHTEN vollmacht
     on konto.ID = vollmacht.KTO_ID
```

Karthesisches Produkt

```
select konto.*, buchung.*, vollmacht.*
  from KONTEN konto
  left outer join BUCHUNGEN buchung
     on konto.KTO_ID = buchung.KTO_ID
  left outer join VOLLMACHTEN vollmacht
     on konto.KTO_ID = vollmacht.KTO_ID
```

KTO_ID	KTO_NR	...	BUCH_ID	BUCH_BETRAG	...	VM_ID	VM_NAME	...
1	12344	...	10	1000,00 €	...	200	Michael Plöd	...
1	12344	...	10	1000,00 €	...	300	Martin Plöd	...
1	12344	...	20	-324,23 €	...	200	Michael Plöd	...
1	12344	...	20	-324,23 €	...	300	Martin Plöd	...
1	12344	...	30	543,11 €	...	200	Michael Plöd	...
1	12344	...	30	543,11 €	...	300	Martin Plöd	...
2	21300	...	40	-4323,23 €	...	400	Sandra Ulrich	...
2	21300	...	40	-4323,23 €	...	400	Melanie Ulrich	...
2	21300	...	40	-4323,23 €	...	400	Herbert Ulrich	...
...	...	...	...	...	...	...	...	...

Beispiel:
Vererbungshierarchie
Table per Class: Alle Constraints funktionieren, schlechtere Performance
Table per Classhierarchie: Hohe Performance, kein not null

FETCHING

- ★ Batch
- ★ Subselect
- ★ Eager

CACHING

- ★ 1st Level Cache
- ★ 2nd Level Cache
- ★ Stateless Session

ABFRAGEN

- ★ Selektivität
- ★ Query Cache
- ★ Bind Variablen

LOGIK

- ★ Schleifen
- ★ Datenmenge

LOCKS

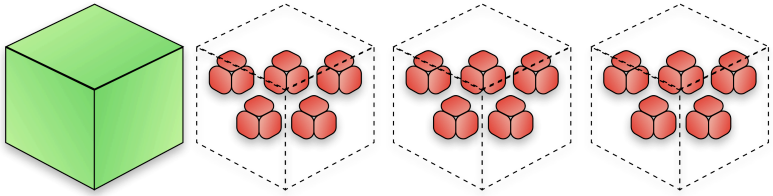
- ★ Optimistic Locking

Runtime

- ★ DB Entwurf
- ★ Konfiguration
- ★ Connection Pool

Fetching

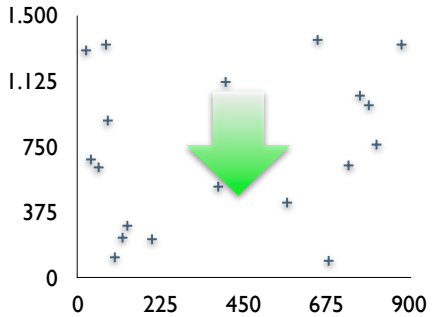
Batch Fetching



```
@Entity
public class Konto {
 @ManyToOne(...)
 public Person getKunde()
 {...}
 ...
}
@Entity
@BatchSize(size=5)
public class Person {
 ...
}
```

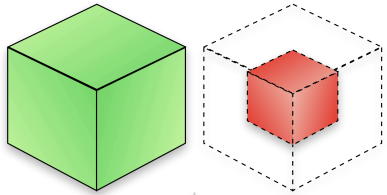
```
SELECT k.* FROM KONTEN k

SELECT * FROM PERSONEN
WHERE PERSON_ID IN (?, ?, ?, ?, ?)

SELECT * FROM PERSONEN
WHERE PERSON_ID IN (?, ?, ?, ?, ?)

SELECT * FROM PERSONEN
WHERE PERSON_ID IN (?, ?, ?)
```

Batch Fetching



- ★ Schätzung
- ★ Einfach
- ★ Lazy
- ★ (N / Batch Size) + 1

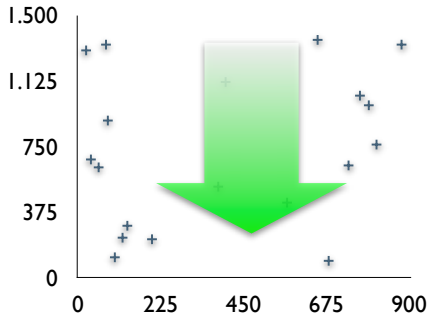
Subselect Fetching



```
@Entity
public class Konto {
 @OneToMany
 @Fetch(FetchMode.SUBSELECT)
 public Set getBuchungen()
 {...}
 ...
}
```

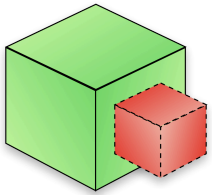
```
SELECT k.* FROM KONTEN k

SELECT b.* FROM BUCHUNGEN b
WHERE b.KTO_ID IN (
  SELECT k.KTO_ID FROM KONTEN k
)
```

Subselect Fetching



- ★ Nur für Collections
- ★ Keine Schätzung
- ★ Einfach
- ★ Lazy
- ★ 2 Abfragen

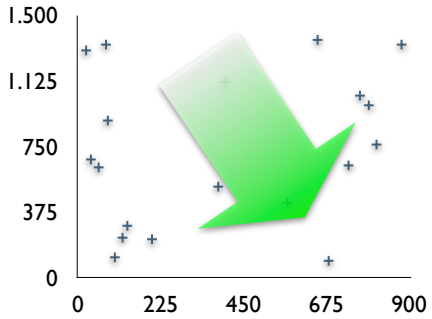
Eager Fetching



```
@Entity
public class Konto {
 @OneToMany(
   fetch = FetchType.EAGER
 )
 public Set getBuchungen()
 {...}

 @ManyToOne(
   fetch = FetchType.EAGER
 )
 public Kunde getEigentuemer()
 {...}
}
```

```
SELECT ko.*, b.*, ku.*
  FROM KONTEN ko
  LEFT OUTER JOIN BUCHUNGEN b
    on b.KTO_ID = ko.KTO_ID
  LEFT OUTER JOIN KUNDEN ku
    on ko.KU_ID = ku.KU_ID
```

Eager Fetching

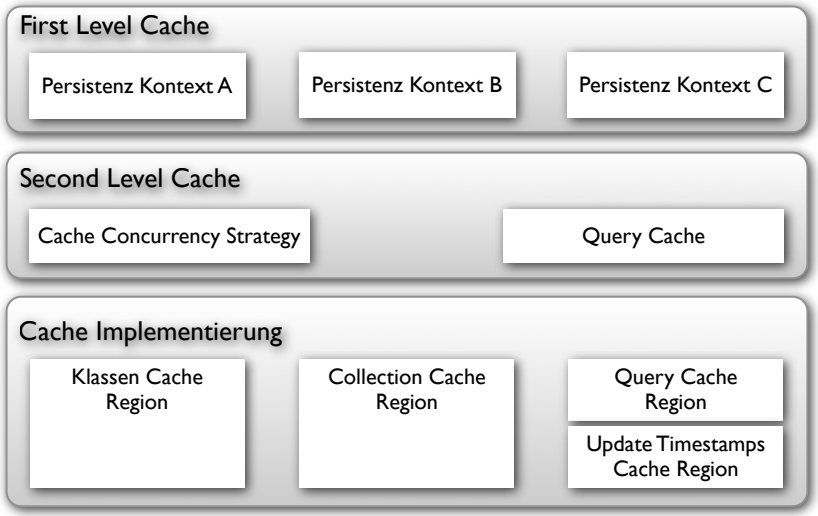


- ★ Nie bei 2+ Collections!
- ★ Nicht Lazy
- ★ 1 Abfrage
- ★ Nie in globalen Fetch Plan aufnehmen

HQL deaktiviert Fetchplan
Criteria überschreibt

Caching

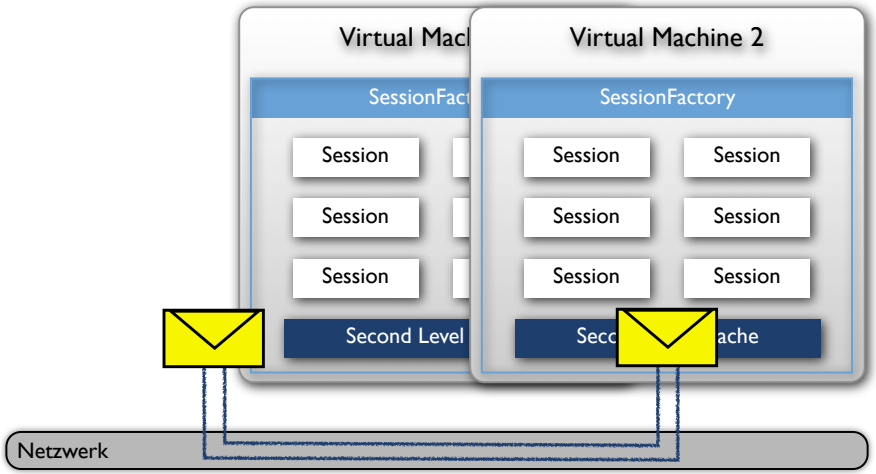
Caching Architektur



Infrastruktur

Verteilte Second Level Cache

Terracotta 2nd Level Caches



Welche
EXCEPTION
bekomme ich, wenn ich
10.000.000 Objekte lade?

OutOfMemory

Hibernate
verwaltet den
1st Level Cache
nicht von selbst!

Grundregeln

- ★ ORM ist kein Batch Tool!
- ★ Bei Massen-Verarbeitung regelmässig flushen und clearen!
- ★ JDBC Batch-Size anpassen

```
for ( int i=0; i<100000; i++ ) {
    Konto konto = new Konto(...);
    session.save(konto);
    if ( i % 50 == 0 ) {
        session.flush();
        session.clear();
    }
}
```

Concurrency Strategies

Transactional	Isolation bis zu **repeatable read**
Read-Write	Isolation bis zu **read commited**
Nonstrict-read-write	Keine Konsistenz Garantie, aber Timeouts
Read-only	Nur für Daten, die sich **nie** ändern

Nonstrict-read-write:
only one transaction updates the items at a time.

Read Committed
Im Unterschied zur vorhergehenden Ebene sind hier Änderungen einer parallel ablaufenden Transaktion erst nach einem commit sichtbar. Das bedeutet, dass Transaktionen lediglich vor Daten geschützt sind, die am Ende einer anderen Transaktion nicht übernommen werden. Solche Daten sind üblicherweise fehlerhaft

Repeatable Read
Bei dieser Isolationsebene ist sichergestellt, dass wiederholte Leseoperationen mit den gleichen Parametern auch die selben Ergebnisse haben. Üblicherweise wird dies sichergestellt, indem eine Transaktion nur Daten sieht, die vor ihrem Startzeitpunkt vorhanden waren. Eine parallele Änderung führt somit auch nach commit nicht zu Inkonsistenzen während einer Transaktion. Dennoch ist es möglich, dass zwei Transaktionen parallel den selben Datensatz modifizieren und nach Ablauf dieser beiden Transaktionen nur die Änderungen von einer von ihnen übernommen wird.

Cache Provider

	Transactional	Read-write	Nonstrict Read-write	Read-only
EHCache		x	x	x
OSCache		x	x	x
SwarmCache			x	x
JBoss Cache	x			x

EHCache is a cache provider intended for a simple process scope cache in a single JVM. It can cache in memory or on disk, and it supports the optional Hibernate query result cache. (The latest version of EHCache now supports clustered caching, but we haven't tested this yet.)

OpenSymphony OSCache is a service that supports caching to memory and disk in a single JVM, with a rich set of expiration policies and query cache support.

SwarmCache is a cluster cache based on JGroups. It uses clustered invalidation but doesn't support the Hibernate query cache.

JBoss Cache is a fully transactional replicated clustered cache also based on the JGroups multicast library. It supports replication or invalidation, synchronous or asynchronous communication, and optimistic and pessimistic locking. The Hibernate query cache is supported, assuming that clocks are synchronized in the cluster.

Konfiguration

org.hibernate.cache.provider_class

EHCache	org.hibernate.cache.EhCacheProvider
OSCache	org.hibernate.cache.OsCacheProvider
SwarmCache	org.hibernate.cache.SwarmCacheProvider
JBoss Cache	org.hibernate.cache.TreeCacheProvider

Mappings

- Annotation:
  @Cache(usage=CacheConcurrencyStrategy.READ_WRITE)
- XML:
  <cache usage="read-write">
- Sowohl auf Klassen als auch auf Collection Level
- Volles Caching: Klasse + Collection!

Beispiel

```
@Entity
@Cache(usage=CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Author implements Serializable {
...
   @Override
   public int hashCode() { ... }
   @Override
   public boolean equals(Object obj) { ... }
}

@Entity
@Cache(usage=CacheConcurrencyStrategy.READ_WRITE)
public class RecordReview implements Article {
...
    @OneToMany(fetch=FetchType.LAZY)
    @Cache(usage=CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
    private Set<Author> authors = new HashSet<Author>();
...
}
```

Cache Regions

- ★ Einteilung in Cache Regions mit Naming Convention
- ★ Cache Regions werden in Cache Provider Konfiguration referenziert

Klasse de.allschools.domain.Band	Voll qualifizierter Name de.allschools.domain.Band
Collection de.allschools.domain.Record#bands	Klasse + „.“ + Attribut de.allschools.domain.Record.bands

EhCache Beispiel

ehchache.xml

```
<ehcache>
    <diskStore path="java.io.tmp"/>

    <defaultCache maxElementsInMemory="10000" eternal="true"
        overflowToDisk="true" />

    <cache name="de.allschools.domain.Author"
       maxElementsInMemory="30"
       eternal="false"
       timeToIdleSeconds="900"
       timeToLiveSeconds="1800"
       overflowToDisk="true" />
    <cache name="de.allschools.domain.RecordReview.authors"
       maxElementsInMemory="500"
       eternal="false"
       timeToIdleSeconds="600"
       timeToLiveSeconds="1200"
       overflowToDisk="true" />
      ...
</ehcache>
```

Collection Cache hält nur IDs

Legacy Feature Property Reference auf nicht Primary Key => Kein Cache greift

Auswahl von Caching Kandidaten?

KONSERVATIV

- ★ Wenige Inserts und Updates
- ★ Viele Lesezugriffe
- ★ Unkritische Daten
- ★ Von vielen Sessions benötigt
- ★ Von vielen Usern benötigt

Stateless Session

- ★ sessionFactory.openStatelessSession()
- ★ Command orientierte API
- ★ Kein Persistenz Kontext
- ★ Kein Caching
- ★ Kein Transaktionales write-behind
- ★ Kein Cascading
- ★ Keine Interceptors und Events

Abfragen

Selektivität

- ★ Nur benötige Daten laden
- ★ Möglichst früh einschränken
- ★ Projection verwenden

```
Query query = getSession().createQuery(
  "select
   new TourCityInfo(t.name, d.timestamp, l)
   from Tour t
     inner join t.tourDates d
       with d.timestamp > :datum
     inner join d.location l
   where l.stadt=:stadt
   order by t.name asc"
);
```


"from User u
where u.name=" + name
SQL Injection
Performance Killer
Heimtückisch

IMMER

BIND VARIABLEN

verwenden

```
Query query =
session.createQuery("from User u where u.name= :name");
q.setString("name", "michael");
```

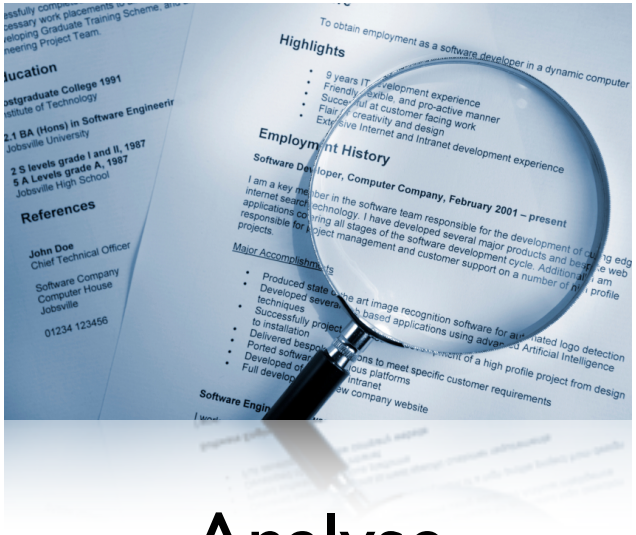
Query Cache

- ★ Wird selten benötigt
- ★ Nur für bestimmte Queries geeignet
- ★ Wird extra konfiguriert:

```
hibernate.cache.use_query_cache=true
```

- ★ Muss pro Query / Criteria aktiviert werden:

```
query.setCacheable(true);
```


Highlights
References
John Doe
Chief Technical Officer
Software Company
Computer House
Jobsville
01234 123456

Analyse

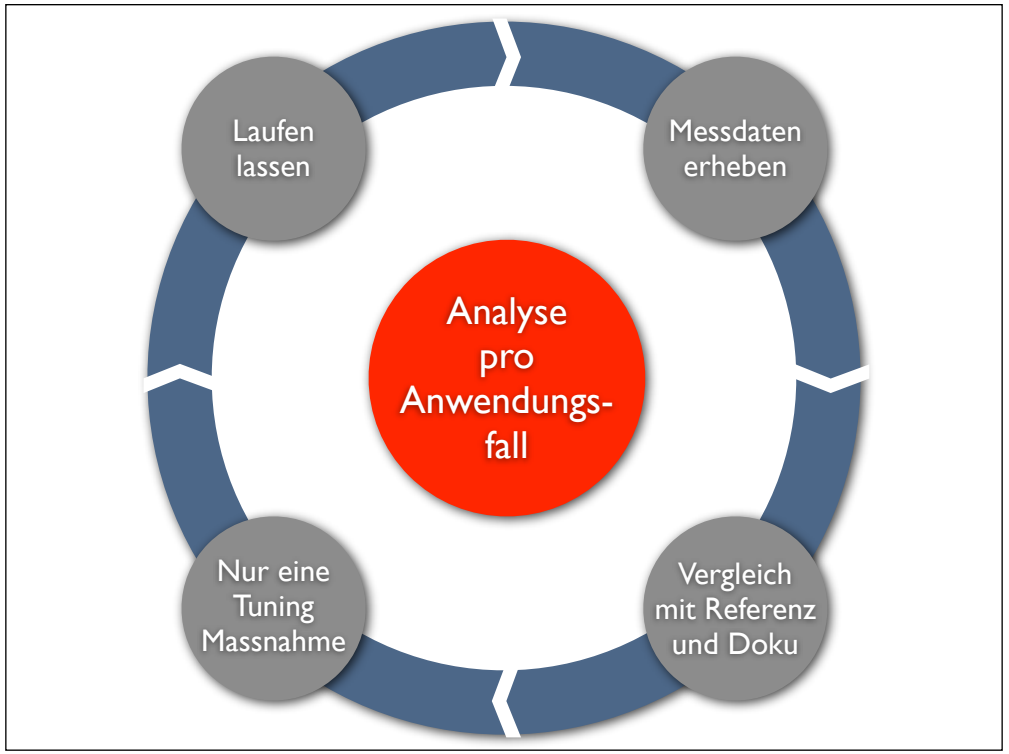
Logging

- ★ Sehr detailliert, viele Informationen
- ★ Interessant sind für Tuning:
  - ➡ org.hibernate.jdbc - TRACE
  - ➡ org.hibernate.SQL - TRACE
- ★ Logging auch in User Types integrieren
- ★ Sicht auf plain SQL

Keine Statistics für Critereia API doooooooooh!!!!!!

Datenbank + Infrastruktur

- ★ Auch in der Datenbank analysieren
  - ➡ Sind alle Indizes korrekt gesetzt?
  - ➡ Wie ist das Laufzeitverhalten?
- ★ Gleiches gilt für Infrastruktur
  - ➡ Connection Pool
  - ➡ Transaktions Monitor
  - ➡ Applikations Server


Laufen lassen
Messdaten erheben
Analyse pro Anwendungs-fall
Nur eine Tuning Massnahme
Vergleich mit Referenz und Doku

Lasttest

- Arten
  - Normale Last
  - Stresstest
  - Lange Laufzeiten
- Setup:
  - Realistisches Hardware Sizing
  - Realistische Datenmenge

DBunit
Lange Laufzeit: Hibernate <3.6
Hibernate Soft References

VIELEN DANK!

FRAGEN?

Jochen Mader
Senacor Technologies AG
jochen.mader@senacor.com