

14.–17. 09. 2009  
in Nürnberg



# Herbstcampus

Wissenstransfer  
par excellence

## Tuning von Hibernate und JPA Anwendungen

Michael Plöd

Senacor Technologies AG

# Ihre Persistenzschicht?

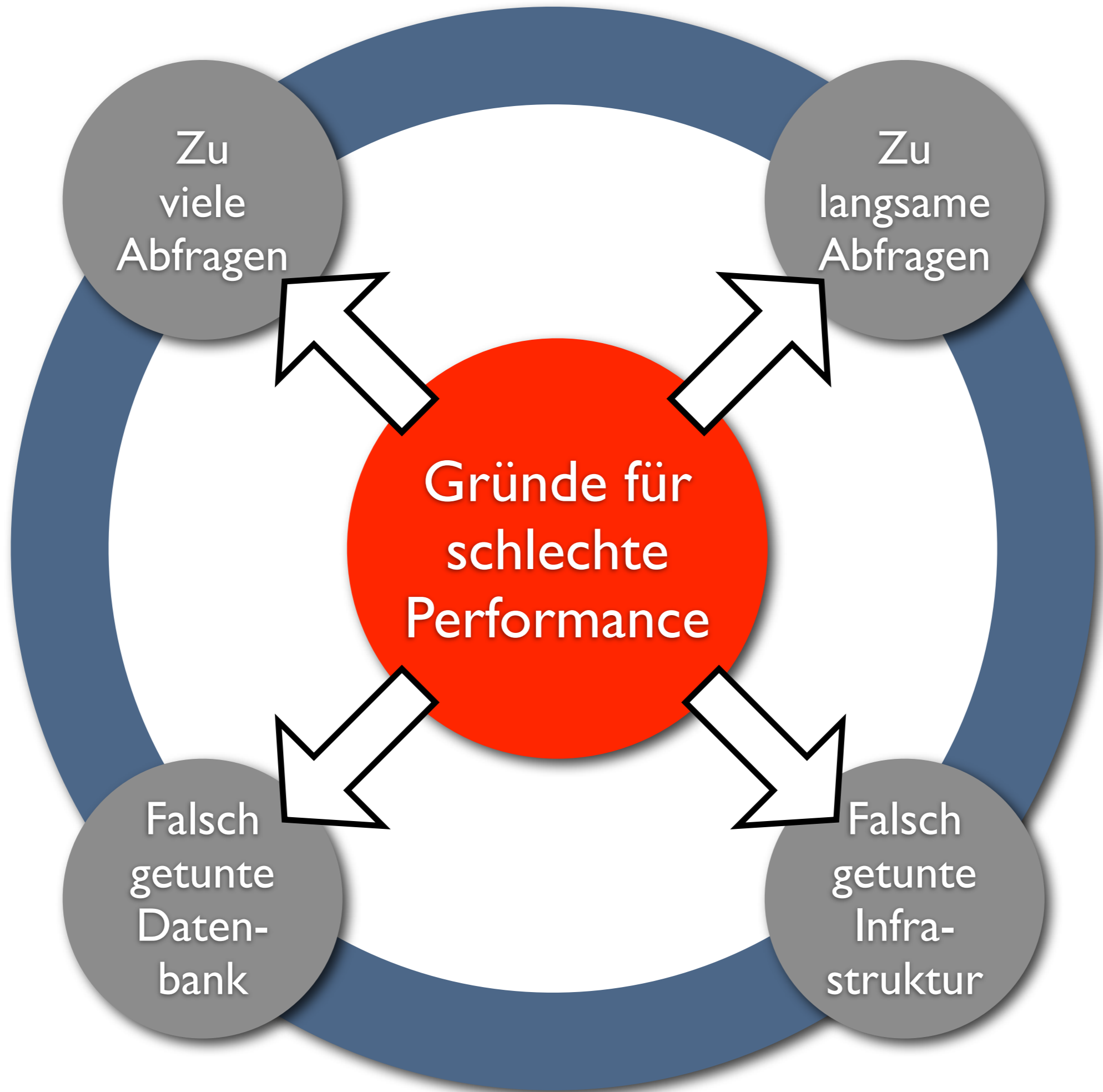


# Michael Plöd

- Architekt bei Senacor Technologies AG in Nürnberg
- <http://www.senacor.com>
- [michael.ploed@senacor.com](mailto:michael.ploed@senacor.com)
- <http://rockingcode.blogspot.com>
- Twitter: @bitboss

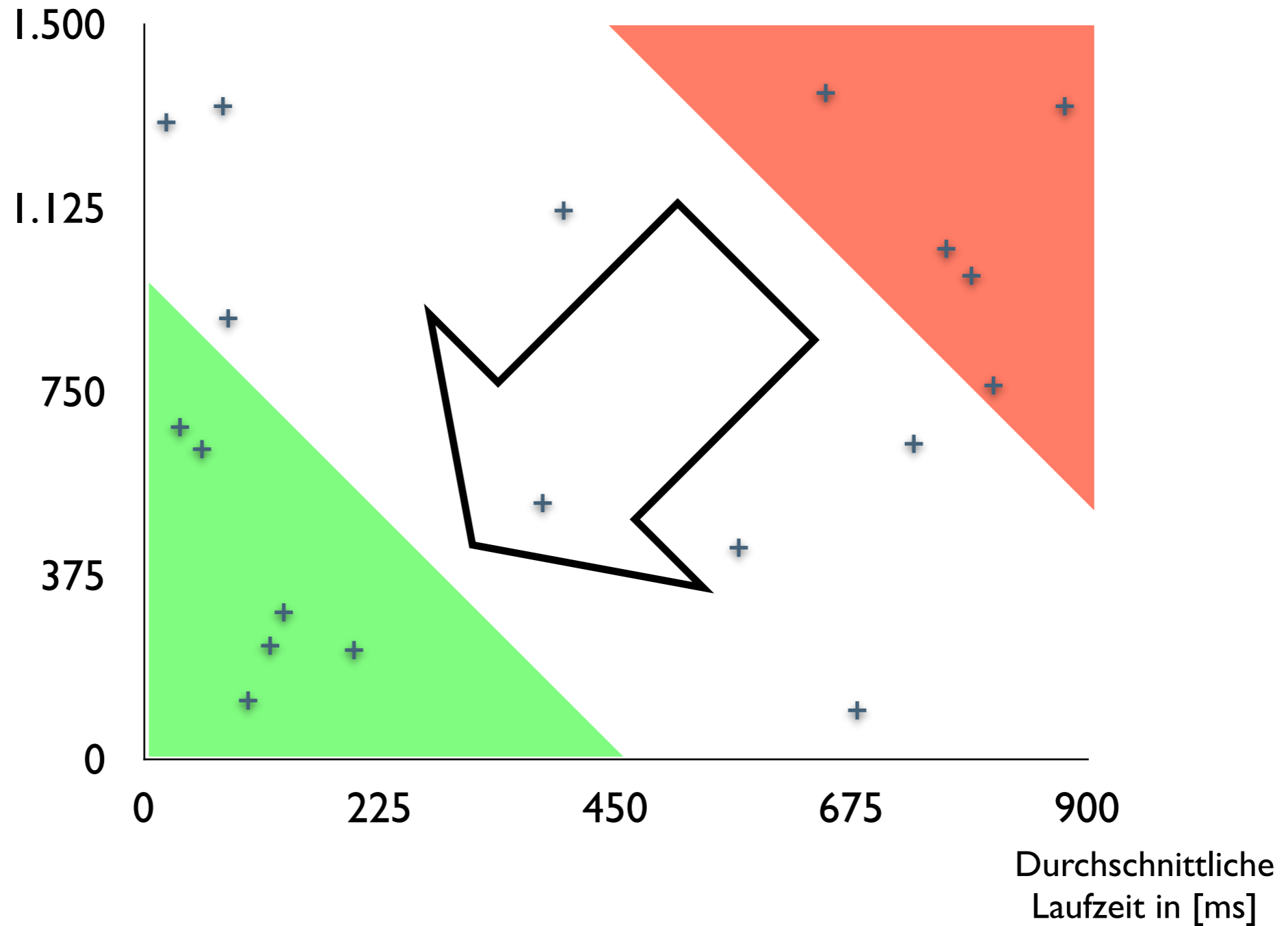
**IST ORM  
LANGSAM**





# Klassifizierung von Abfragen

Anzahl der Abfragen



# Ursachen

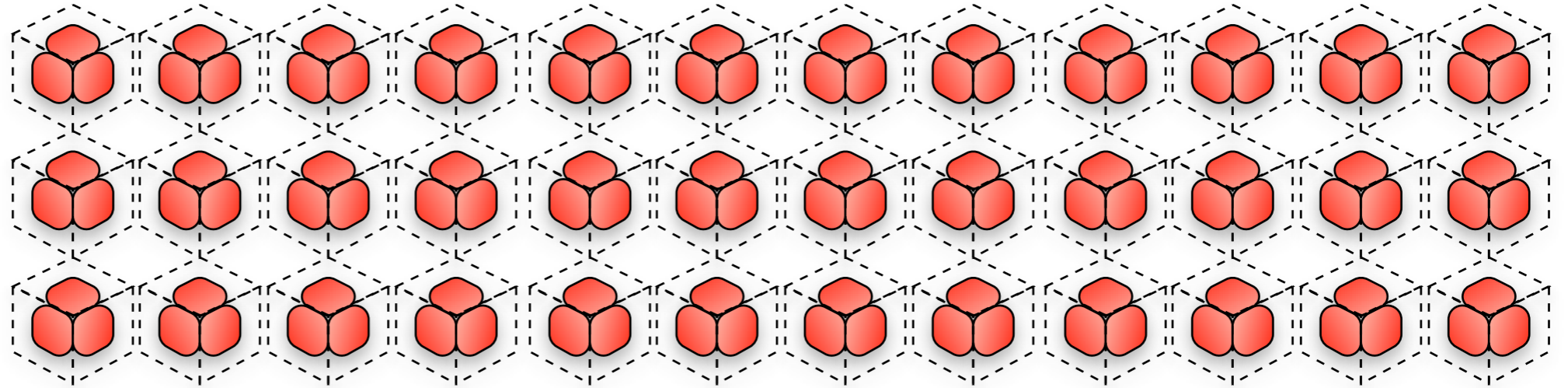
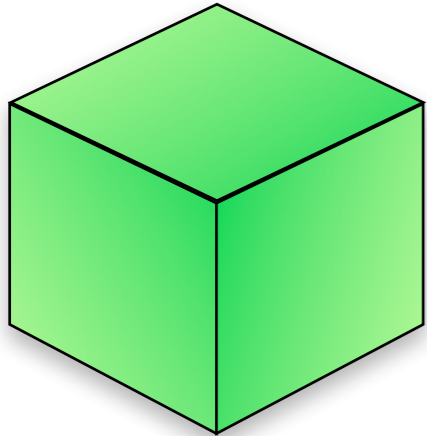
## Hohe Häufigkeit

- ★ Applikations-Logik
- ★ Mappings
- ★ Kein Caching
- ★ N+1 Selects Problem

## Hohe Laufzeit

- ★ Zu hohe Selektivität
- ★ Variablen-Übergabe
- ★ Fehlende Indizes
- ★ Karthesisches Produkt
- ★ Locks
- ★ Datenbankstruktur

# 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

SELECT \* FROM PERSONEN WHERE PERSON\_ID = ?

SELECT \* FROM PERSONEN WHERE PERSON\_ID = ?

SELECT \* FROM PERSONEN WHERE PERSON\_ID = ?

...

+1

N



# 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
```



Gutes

**TUNING**

ist

**IMMER**

eine Frage der

**BALANCE**



## **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

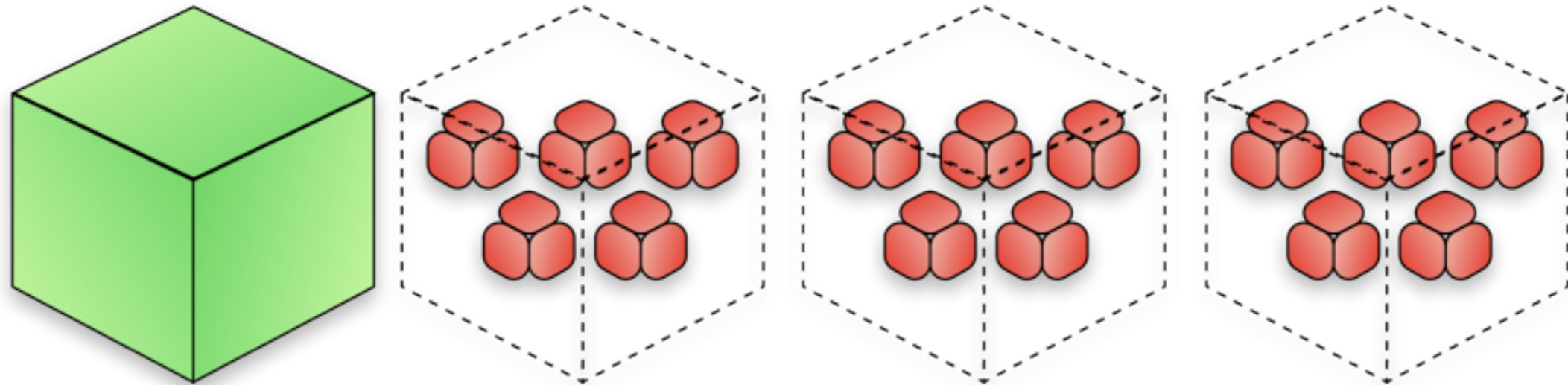
## **Runtime**

- ★ DB Entwurf
- ★ Konfiguration
- ★ Connection Pool



# Mappings

# Batch Fetching



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

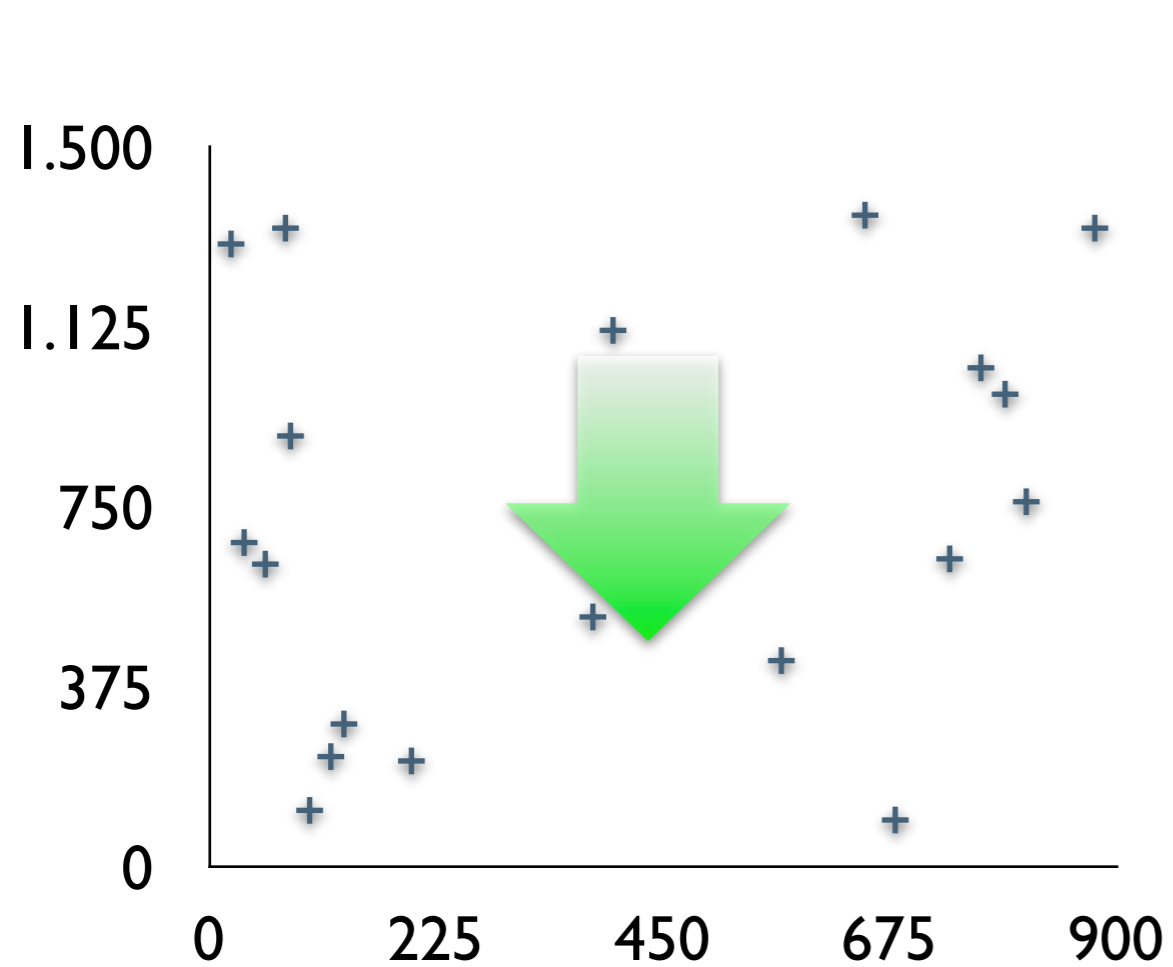
```
SELECT k.* FROM KONTEN k
```

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

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

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

# Batch Fetching



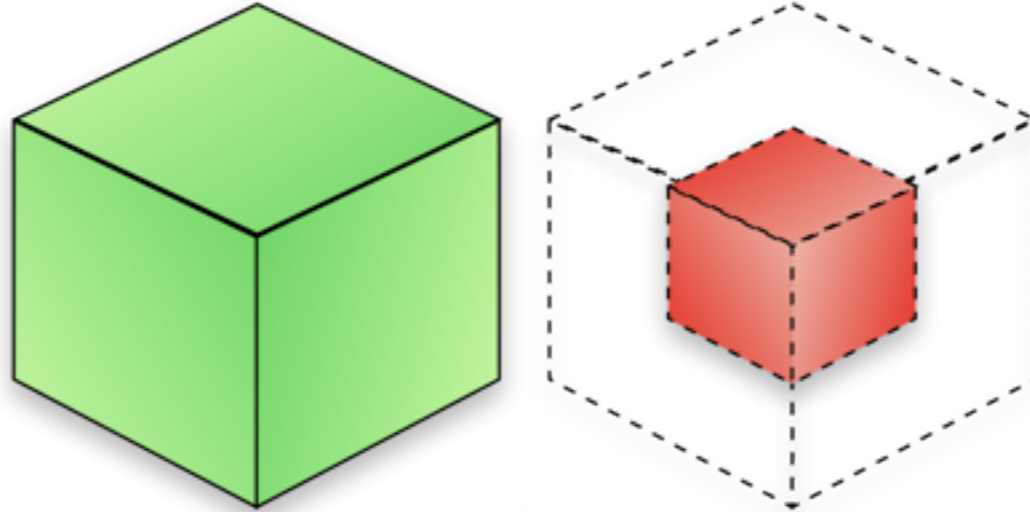
★ Schätzung

★ Einfach

★ Lazy

★  $(N / \text{Batch Size}) + 1$

# Subselect Fetching



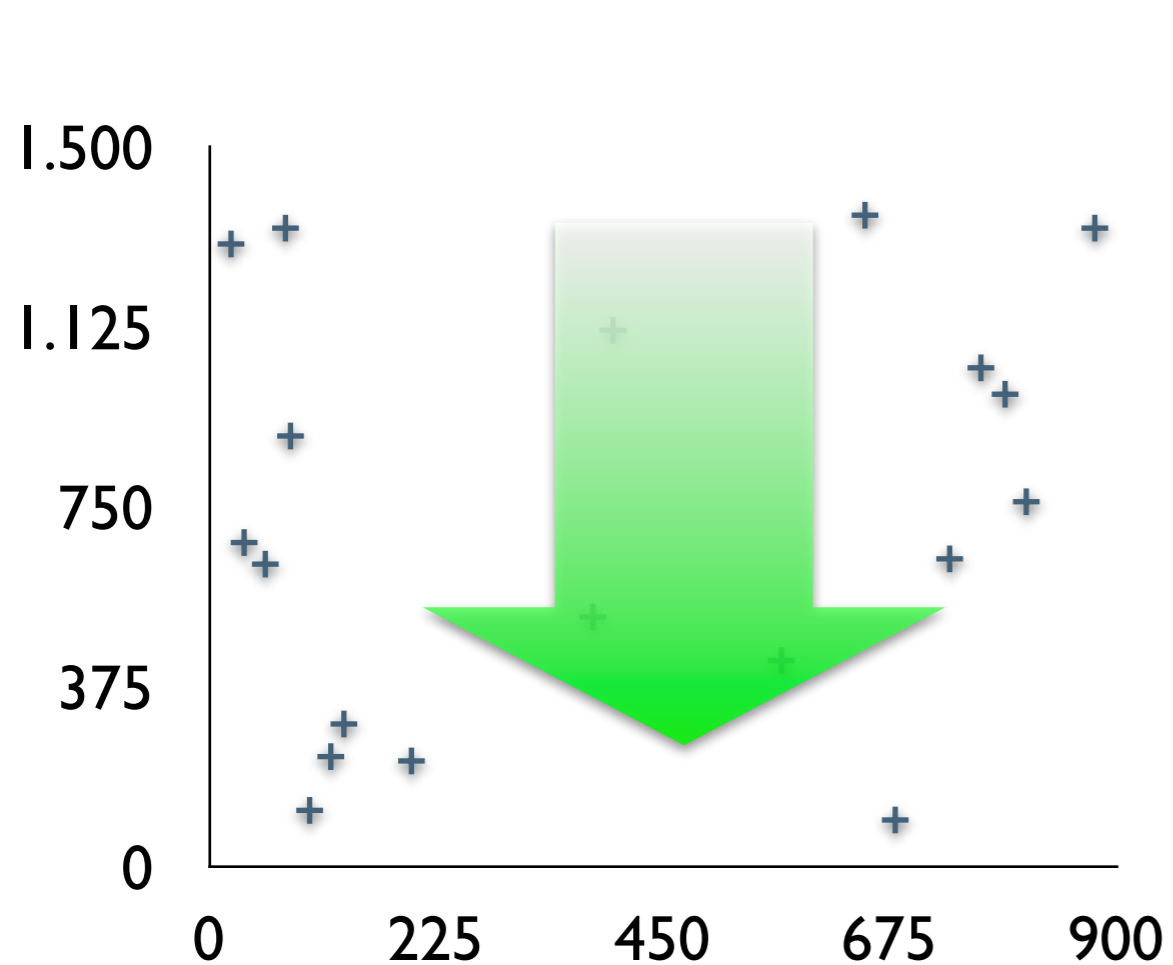
```
@Entity
public class Konto {
    @OneToMany
    @Fetch(FetchMode.SUBSELECT)
    public Set getBuchungen()
    {...}
    ...
}
```

```
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

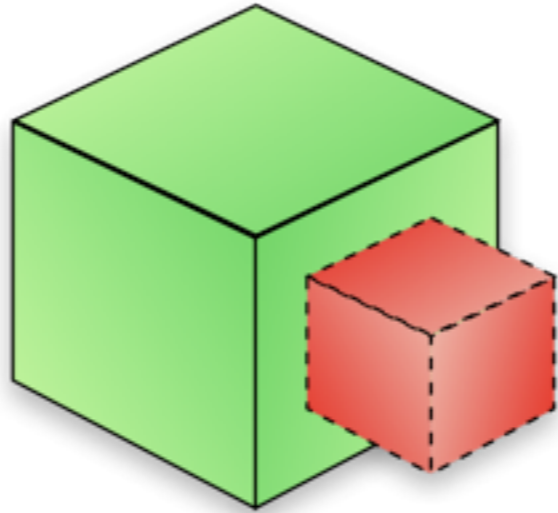
★ Keine Schätzung

★ Einfach

★ Lazy

★ 2 Abfragen

# Eager Fetching

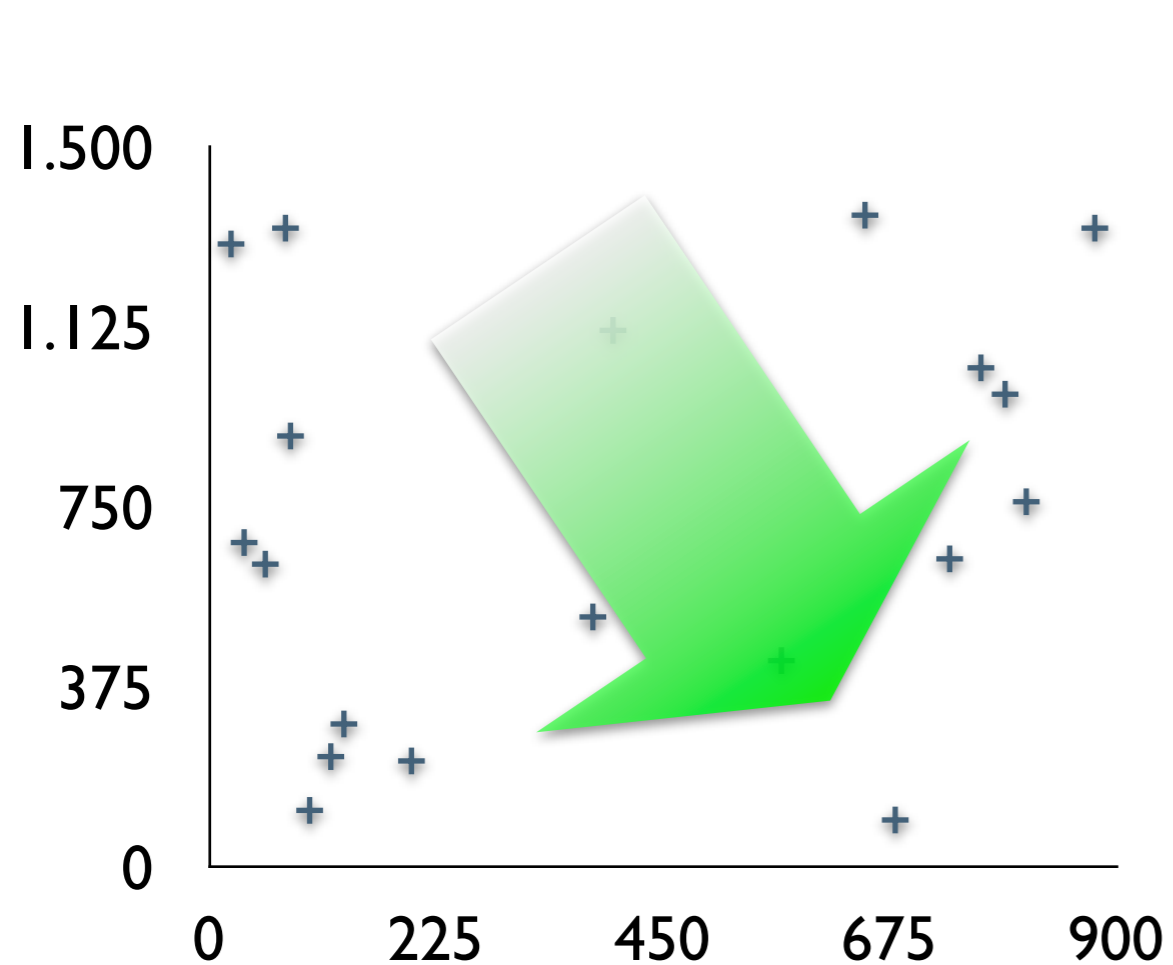


```
@Entity
public class Konto {
    @OneToMany(
        fetch = FetchType.EAGER
    )
    public Set getBuchungen()
    {...}

    @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



# Caching

# Caching Architektur

## First Level Cache

Persistenz Kontext A

Persistenz Kontext B

Persistenz Kontext C

## Second Level Cache

Cache Concurrency Strategy

Query Cache

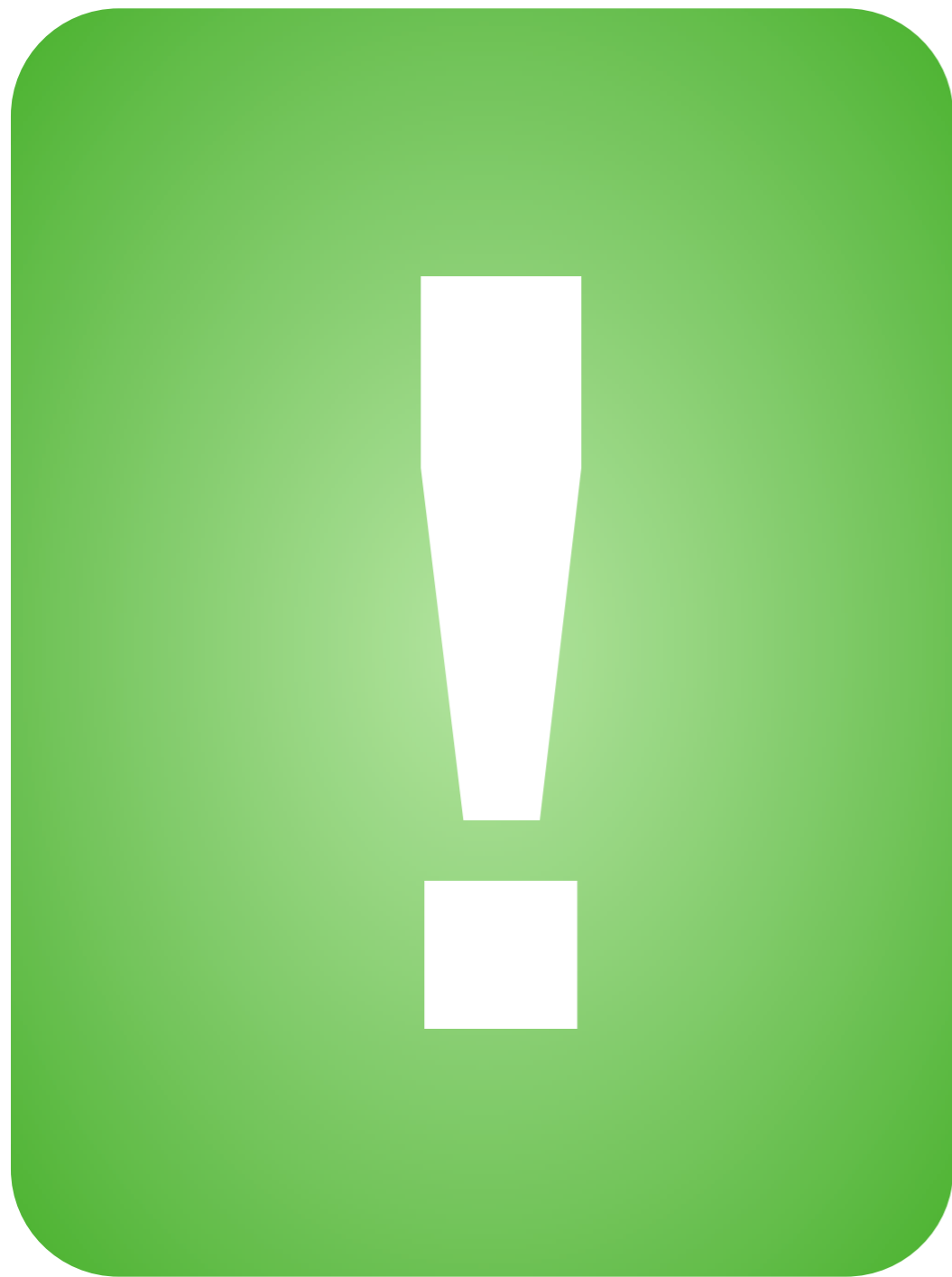
## Cache Implementierung

Klassen Cache  
Region

Collection Cache  
Region

Query Cache  
Region

Update Timestamps  
Cache Region



Hibernate  
verwaltet den  
**Ist 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 <b>repeatable read</b>     |
| Read-Write           | Isolation bis zu <b>read commited</b>       |
| Nonstrict-read-write | Keine Konsistenz Garantie, aber<br>Timeouts |
| Read-only            | Nur für Daten, die sich <b>nie</b> ändern   |



# Cache Provider

|                       | Transactional | Read-write | Nonstrict<br>Read-write | Read-only |
|-----------------------|---------------|------------|-------------------------|-----------|
| EHCache               |               | X          | X                       | X         |
| OSCache               |               | X          | X                       | X         |
| <del>SwarmCache</del> |               |            | X                       | X         |
| JBoss Cache           | X             |            |                         | X         |

# Konfiguration

`org.hibernate.cache.provider_class`

|             |   |
|-------------|---|
| EHCache     | <code>org.hibernate.cache.EhCacheProvider</code>    |
| OSCache     | <code>org.hibernate.cache.OsCacheProvider</code>    |
| SwarmCache  | <code>org.hibernate.cache.SwarmCacheProvider</code> |
| JBoss Cache | <code>org.hibernate.cache.TreeCacheProvider</code>  |

# 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

|  |   |
|--|---|
| <b>Klasse</b><br>de.allschools.domain.Band             | <b>Voll qualifizierter Name</b><br>de.allschools.domain.Band        |
| <b>Collection</b><br>de.allschools.domain.Record#bands | <b>Klasse + „.“ + Attribut</b><br>de.allschools.domain.Record.bands |

# EhCache Beispiel

## ehcache.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>
```

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 benötigte Daten laden
- ★ Möglichst früh einschränken
- ★ Projection verwenden

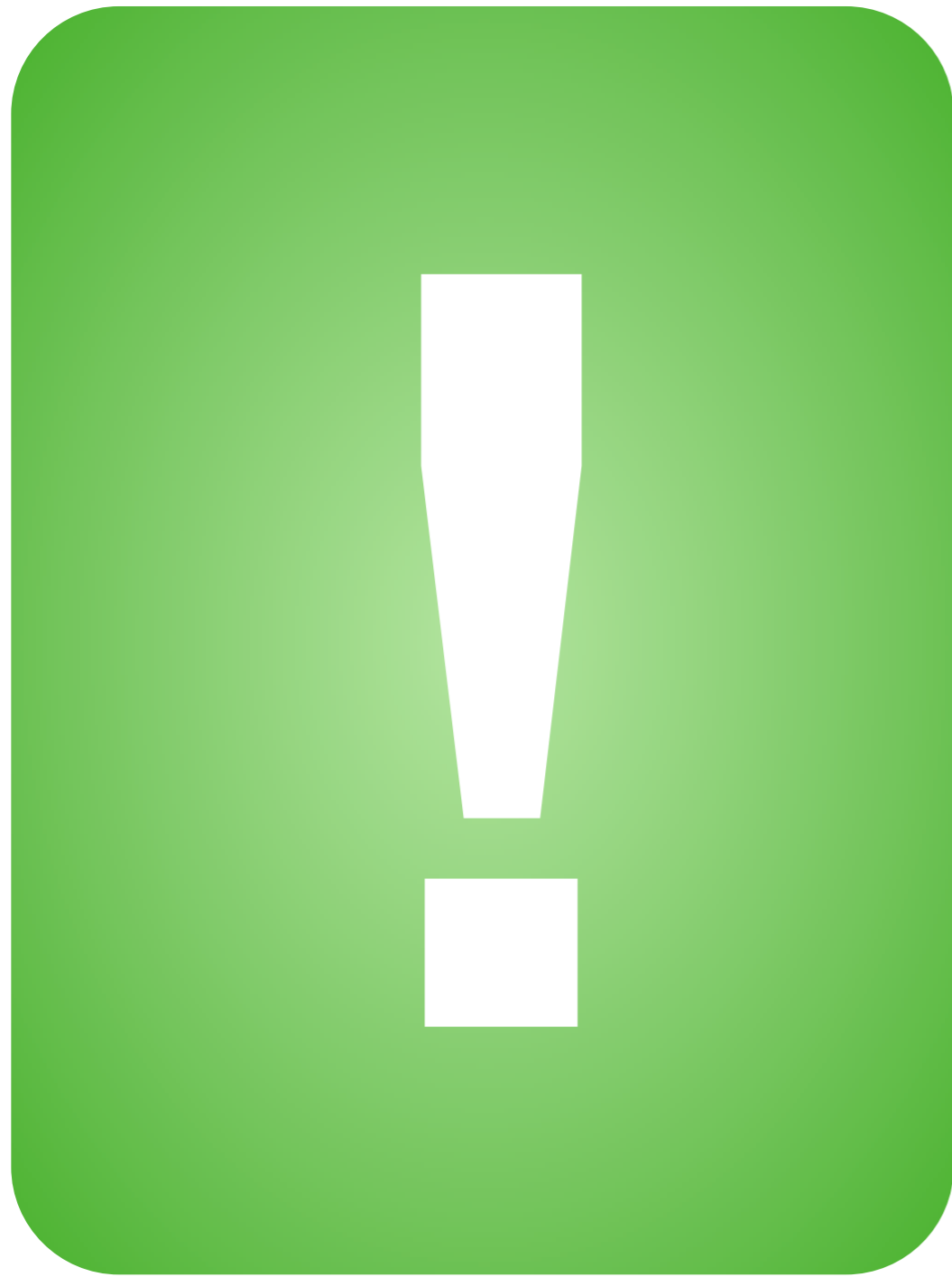
```
Query query = getSession().createQuery(
    "select
    new TourCityInfo(t.name, d.timestamp, l)
    from Tour t
      inner join t.tourDates d
        with d.timestamp > :datum
      inner join d.location l
    where l.stadt=:stadt
    order by t.name asc"
);
```





```
"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);`



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

# Statistics

- ★ Extrem wertvolle Informationen
- ★ Müssen extra aktiviert werden
  - Konfiguration:  
`hibernate.generate_statistics`
  - Programmatisch: `sessionFactory.getStatistics().setStatisticsEnabled(true)`
- ★ Zugriff
  - Programmatisch: `sessionFactory.getStatistics()`
  - JMX



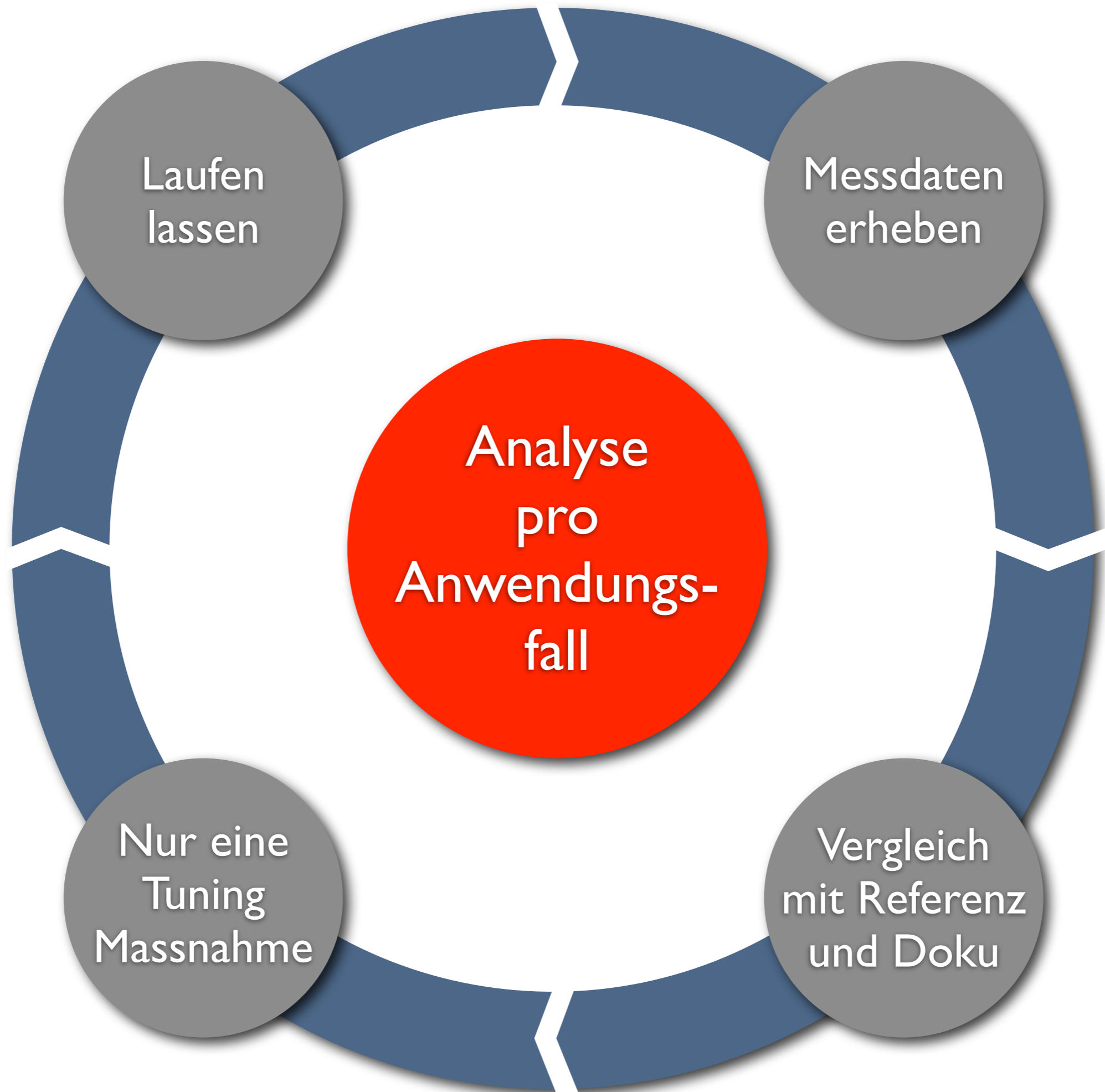
# Datenbank + Infrastruktur

## ★ 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 Anwendungsfall

Nur eine Tuning Massnahme

Vergleich mit Referenz und Doku

# Lasttest

## ★ Arten

- Normale Last
- Stresstest
- Lange Lauzeiten

## ★ Setup:

- Realistisches Hardware Sizing
- Realistische Datenmenge

14.–17. 09. 2009  
in Nürnberg



# Herbstcampus

Wissenstransfer  
par excellence

Vielen Dank!

Michael Plöd

Senacor Technologies AG