Java Persistence API:
Best Practices

SUN TECH DAYS 2008–2009
A Worldwide Developer Conference

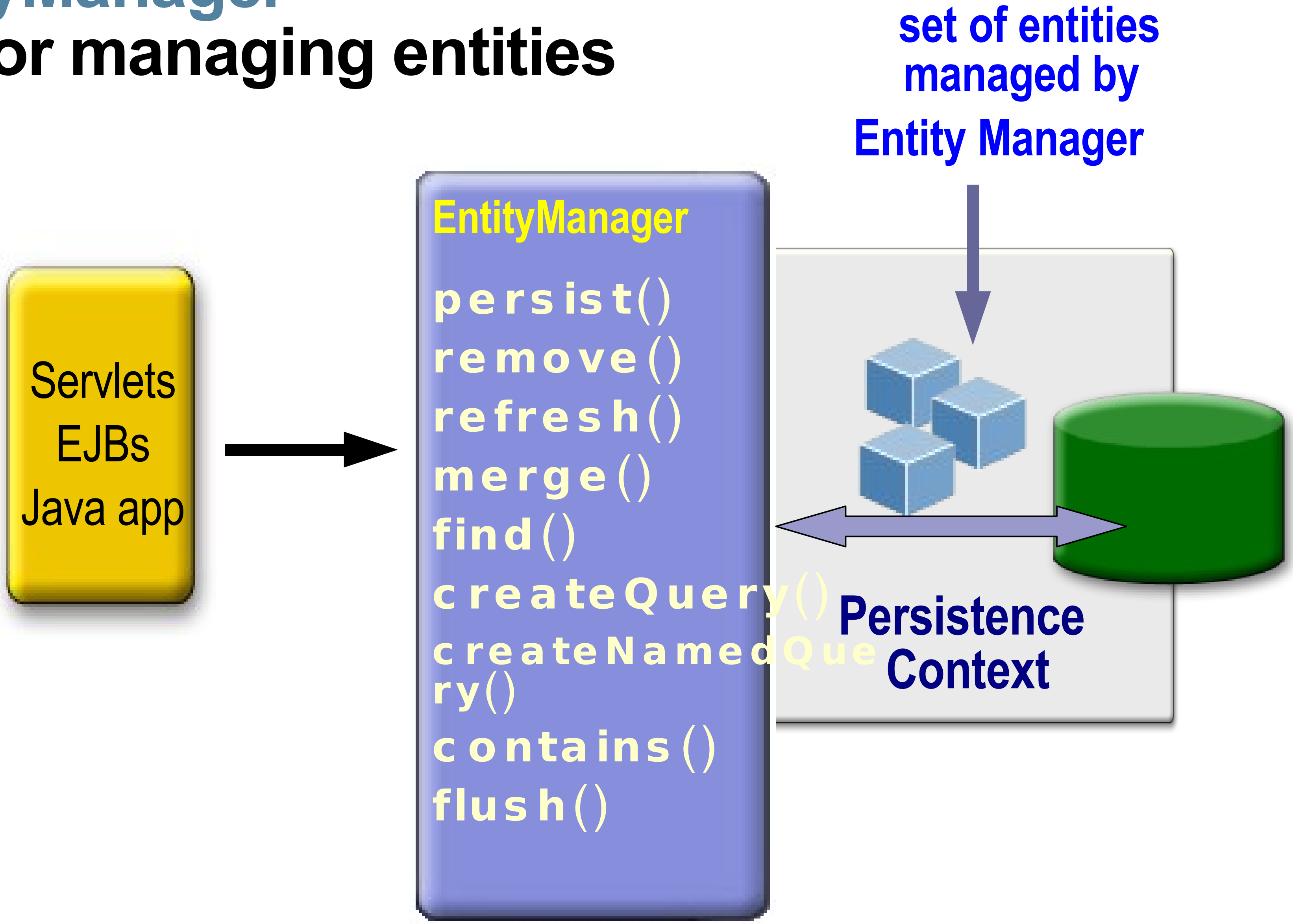
Carol McDonald
Java Architect

Agenda

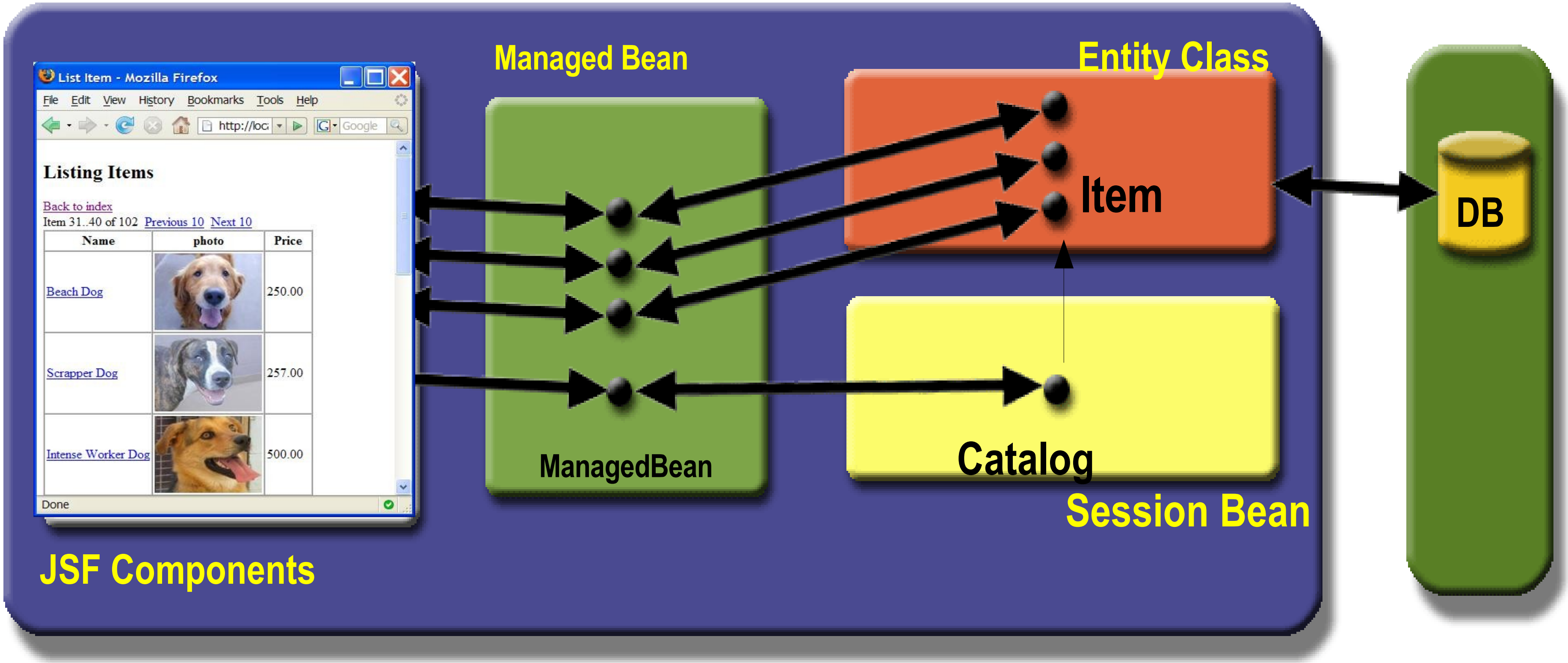
- **Entity Manager**
- Persistence Context
- Entities
- Schema & Queries
- Transaction

EntityManager

API for managing entities



Catalog Java EE Application



List Item - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://loc

Listing Items

Back to index
Item 31..40 of 102 Previous 10 Next 10

Name	photo	Price
Beach Dog		250.00
Scrapper Dog		257.00
Intense Worker Dog		500.00

Done

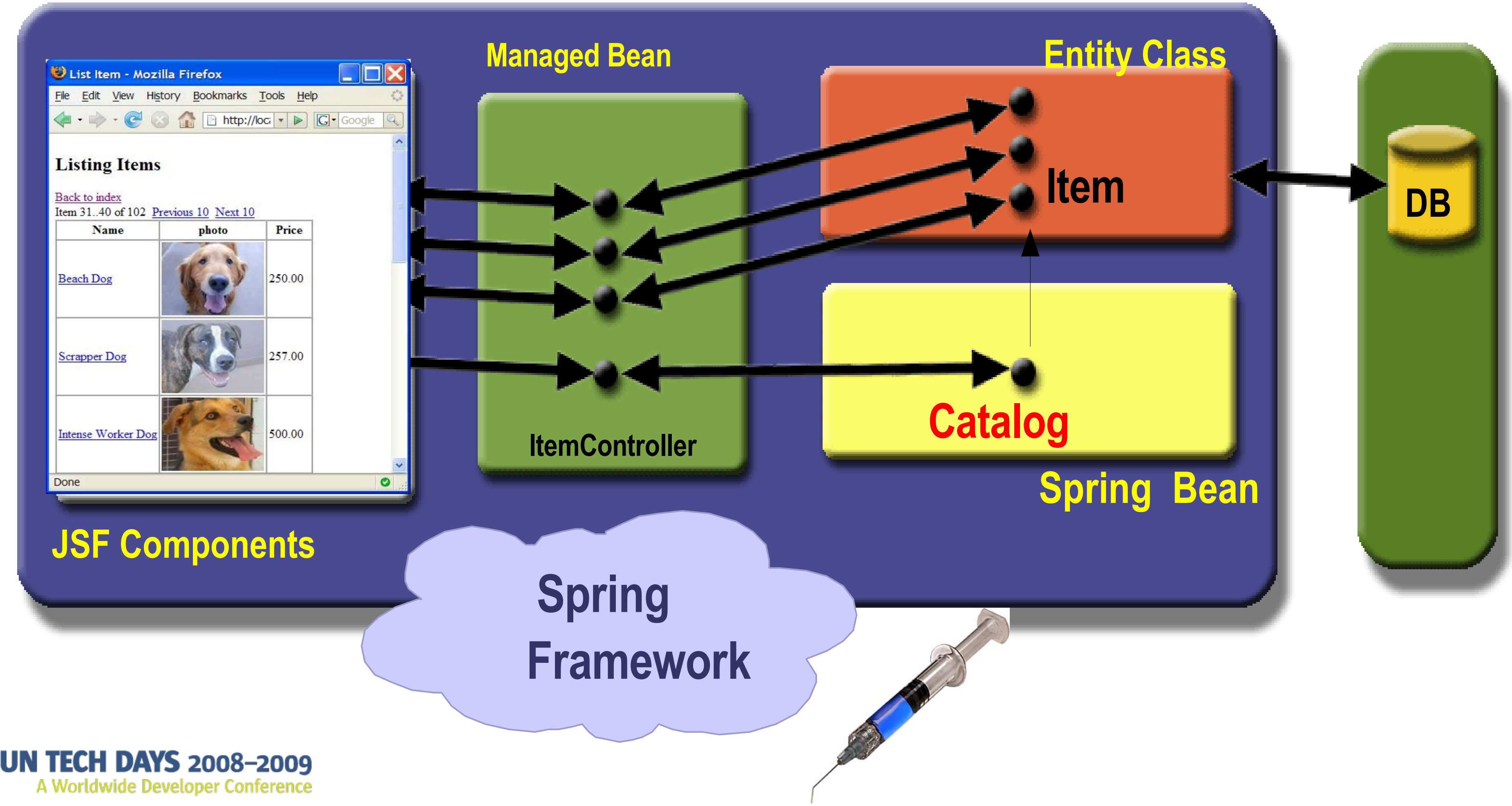
Stateless Session Bean Annotation

EJB EntityManager Example

Dependency Injection

```
@Stateless
public class Catalog implements CatalogService {

   @PersistenceContext(unitName=”PetCatalogPu”)
   EntityManager em;

    @TransactionAttribute(NOT_SUPPORTED)
    public List<Item>  getItems(int firstItem,
         int batchSize) {
         Query q = em.createQuery
            ("select i from Item as i");
         q.setMaxResults(batchSize);
         q.setFirstResult(firstItem);
         List<Item> items= q.getResultList();
         return items;
    }
}
```

Catalog Spring JPA Application



List Item - Mozilla Firefox

Listing Items

Back to index

Item 31..40 of 102 Previous 10 Next 10

Name	photo	Price
Beach Dog		250.00
Scrapper Dog		257.00
Intense Worker Dog		500.00

Done

Managed Bean

Entity Class

Item

DB

ItemController

Catalog

Spring Bean

JSF Components

Spring Framework

Spring with JPA

Component Stereotype

Spring transactions use aop

```
@Repository
@Transactional
public class CatalogDAO implements CatalogService {

@PersistenceContext(unitName="PetCatalogPu")
private EntityManager em;


@Transactional(readOnly=true)
public List<Item> getItems(int firstItem,int batchSize) {
  Query q =
     em.createQuery("select object(o) from Item as o");
  q.setMaxResults(batchSize);
  q.setFirstResult(firstItem);
  List<Item> items= q.getResultList();
  return items;
}
```

Container vs Application Managed

Container managed entity managers (EJB, Spring Bean, Seam component)

- Injected into application
- Automatically closed
- JTA transaction – propagated

Application managed entity managers

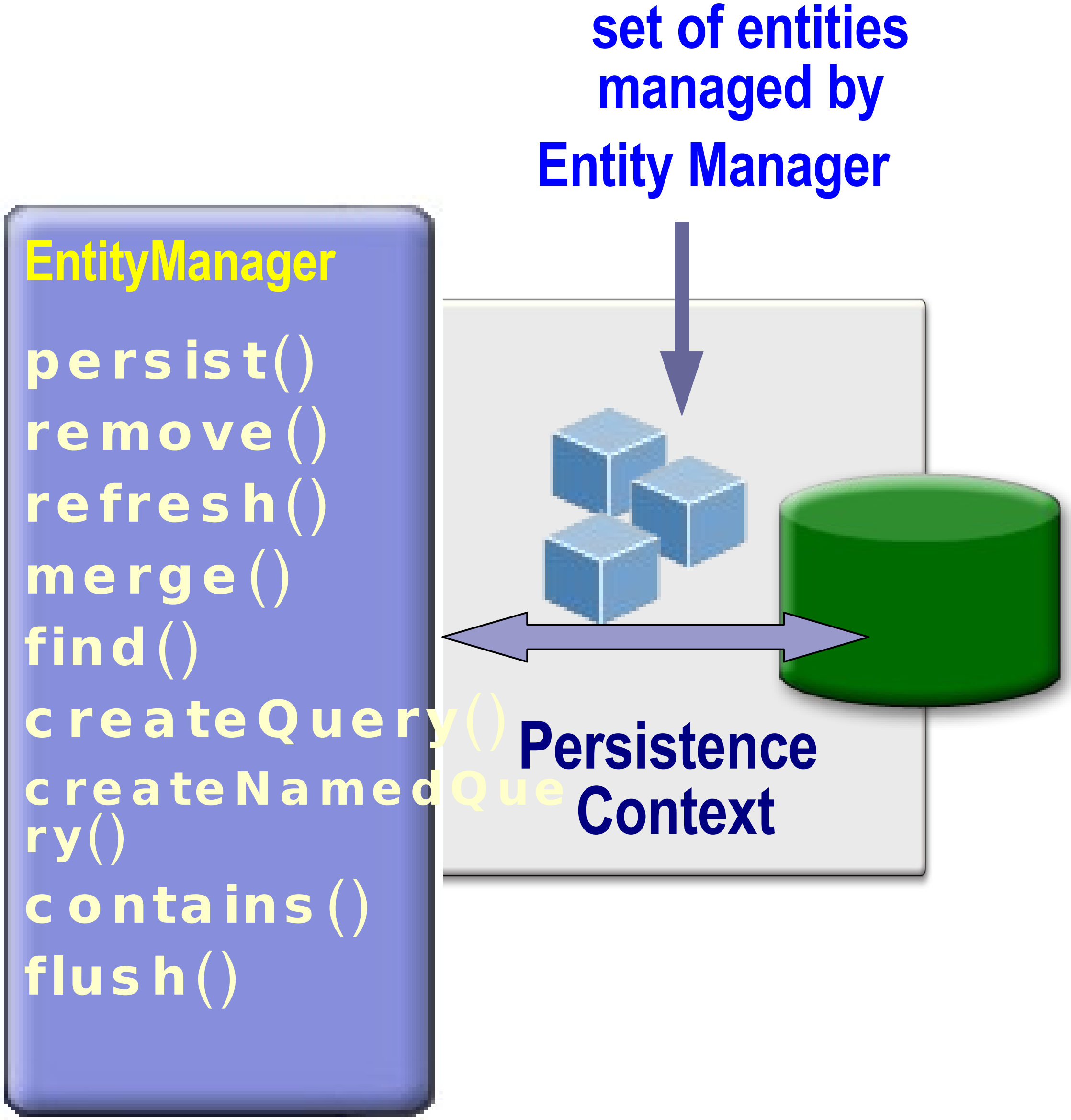
- Used outside of the JavaEE 5 platform
- Need to be explicitly created
  - Persistence.createEntityManagerFactory()
- RESOURCE_LOCAL transaction – not propagated
- Need to explicitly close entity manager

Agenda

- >Entity Manager
- >**Persistence Context**
- >Entities
- >Queries
- >Transaction

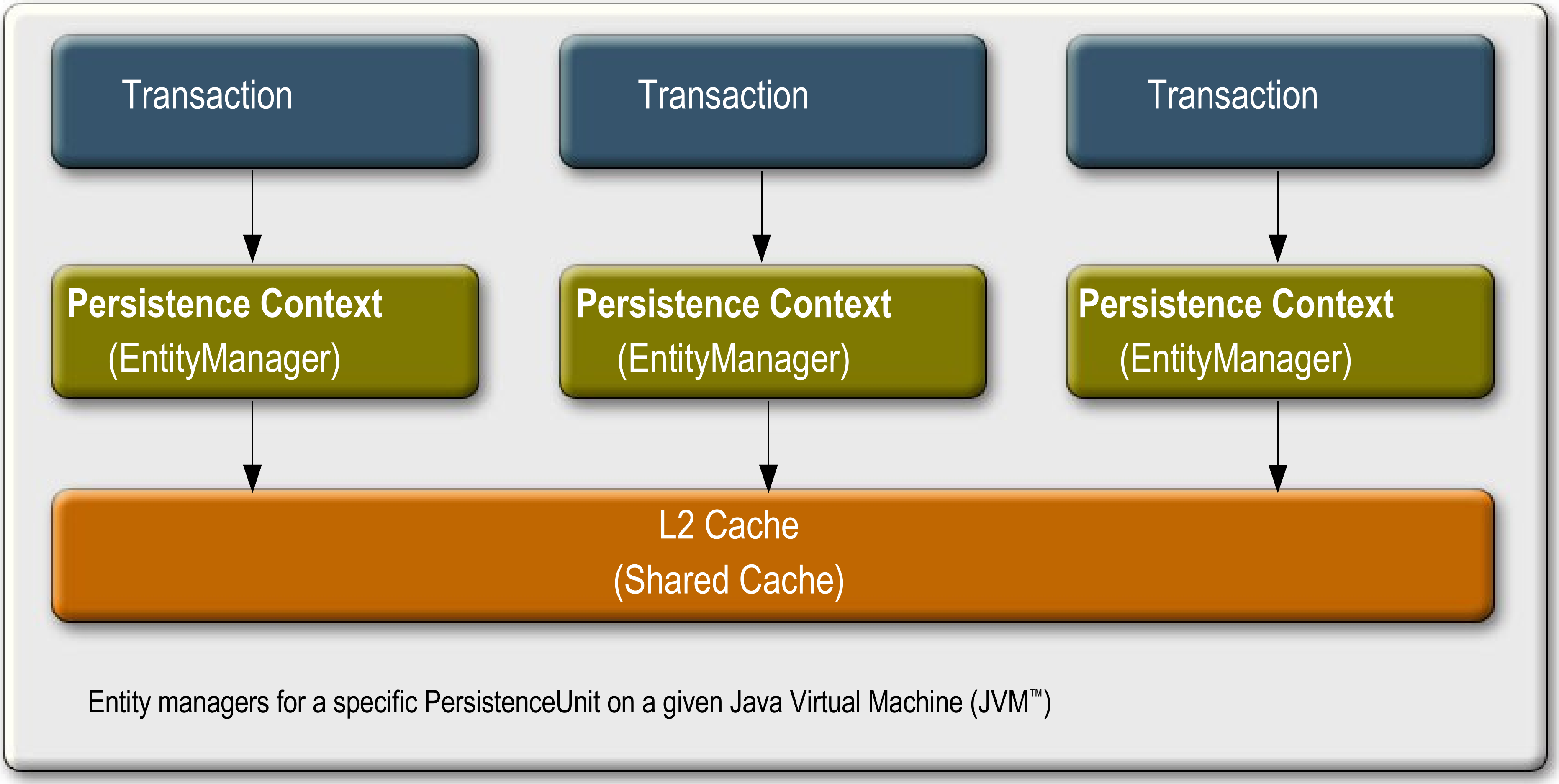
Persistence Context

- Persistence context acts as a first level cache for entities
- Two types of persistence context
  - Transaction scoped
  - Extended scoped persistence context



Level1 and Level2 caches

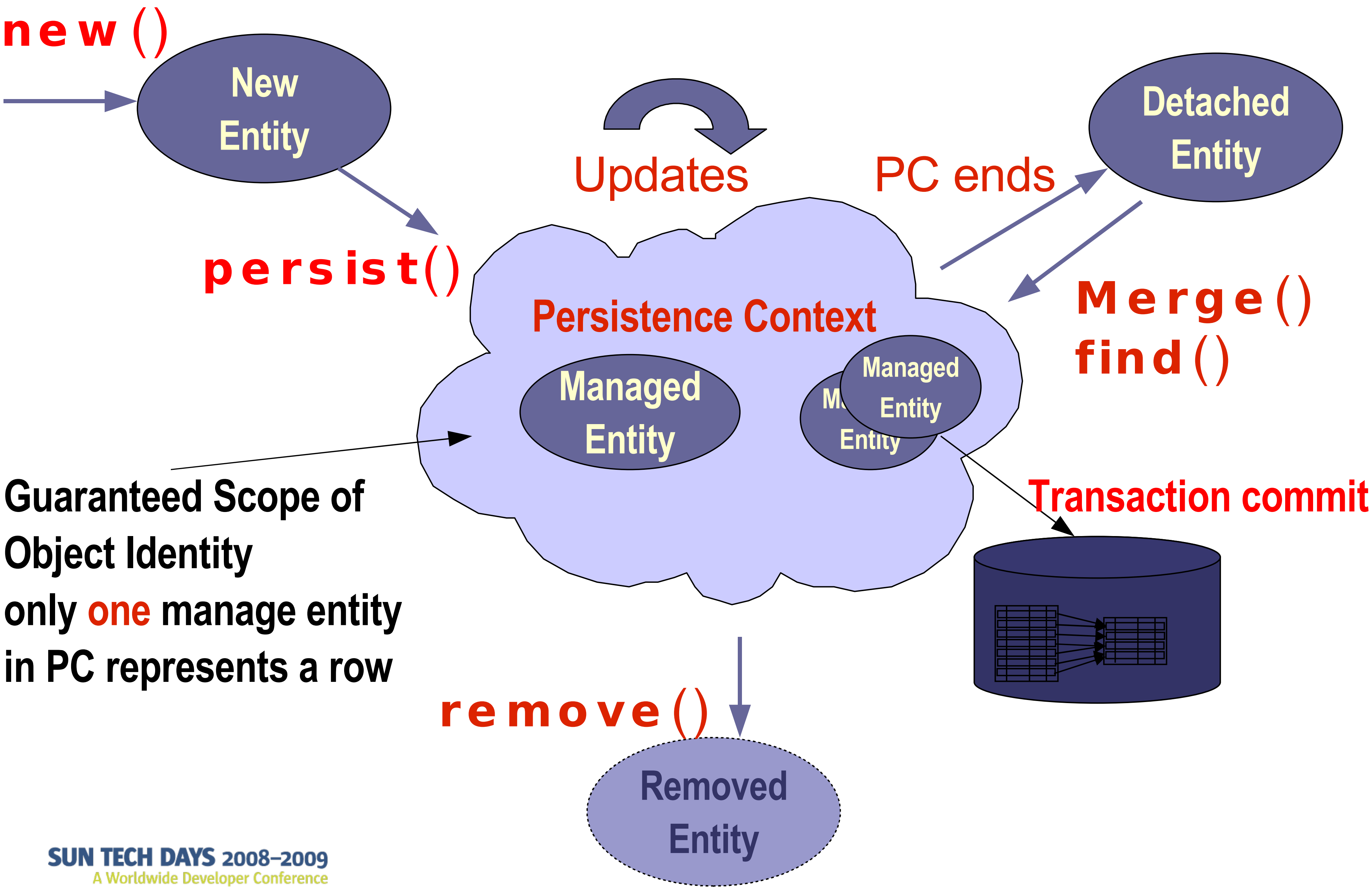
Persistence Context is a Level 1 cache


Transaction
Transaction
Transaction
Persistence Context (EntityManager)
Persistence Context (EntityManager)
Persistence Context (EntityManager)
L2 Cache (Shared Cache)

Entity managers for a specific PersistenceUnit on a given Java Virtual Machine (JVM™)

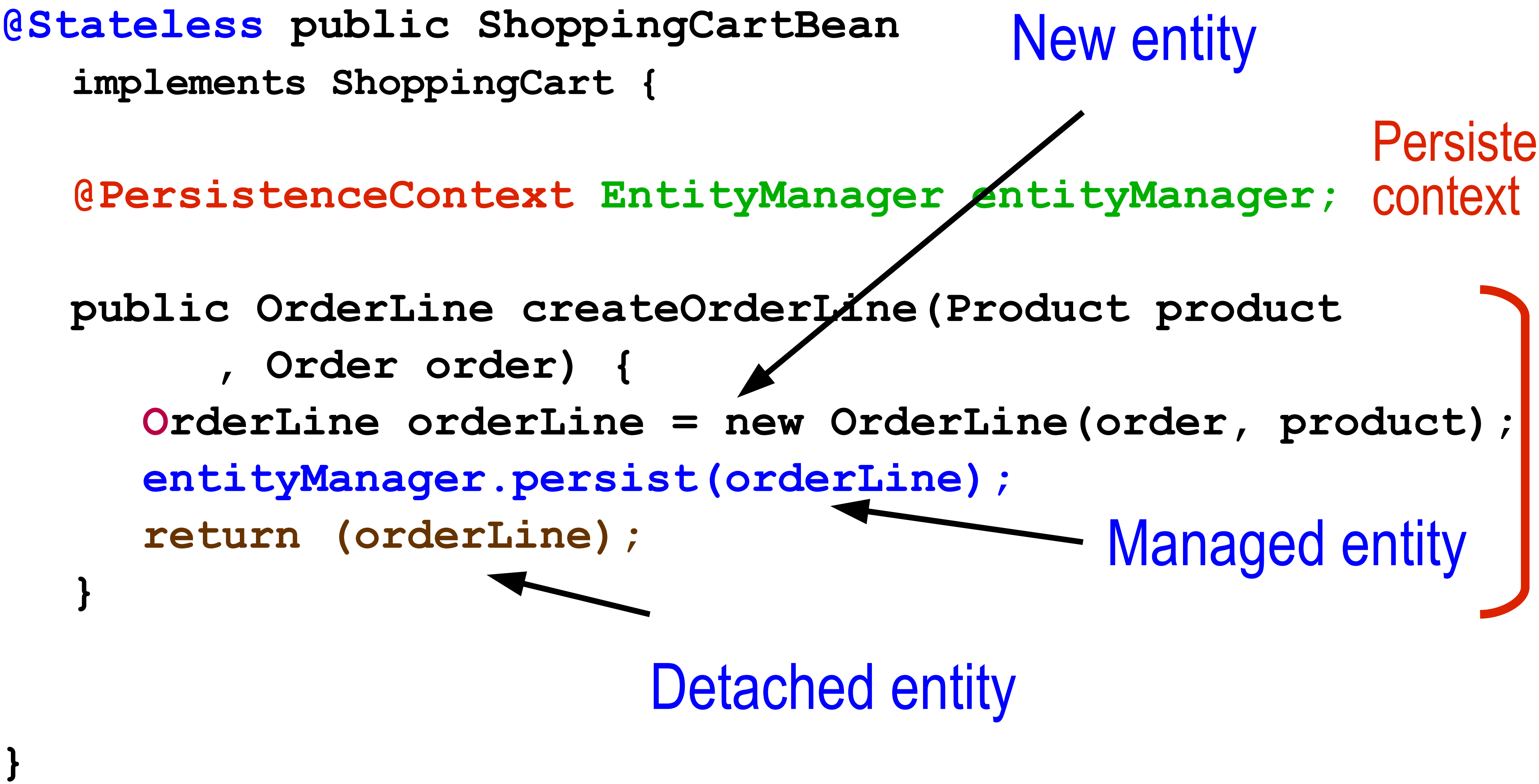
The terms “Java Virtual Machine” and “JVM” mean a Virtual Machine for the Java™ Platform.
Source:http://weblogs.java.net/blog/guruwons/archive/2006/09/understanding_t.html

Entity Lifecycle



Guaranteed Scope of Object Identity
only one manage entity in PC represents a row

Entity Lifecycle Illustrated – The Code



```
@Stateless public ShoppingCartBean
   implements ShoppingCart {

   @PersistenceContext EntityManager entityManager;

   public OrderLine createOrderLine(Product product
         , Order order) {
      OrderLine orderLine = new OrderLine(order, product);
      entityManager.persist(orderLine);
      return (orderLine);
   }

}
```

New entity

Persister
context

Managed entity

Detached entity

Scope of Identity

```
@Stateless public ShoppingCartBean implements ShoppingCart {

 @PersistenceContext EntityManager entityManager;

 public OrderLine createOrderLine(Product product,Order order)
      OrderLine orderLine = new OrderLine(order, product);
      entityManager.persist(orderLine);
      OrderLine orderLine2 =entityManager.find(OrderLine,
                orderLine.getId()));
      (orderLine == orderLine2) // TRUE
      return (orderLine);
   }

}
```

Persistence context

Multiple retrievals of the same object return references to the same object instance

Persistence Context

- Two types of persistence context
- Transaction scoped
  - Used in stateless components
  - Typically begins/ends at request entry/exit points respectively
- Extended scoped persistence context

Persistence Context Propagation

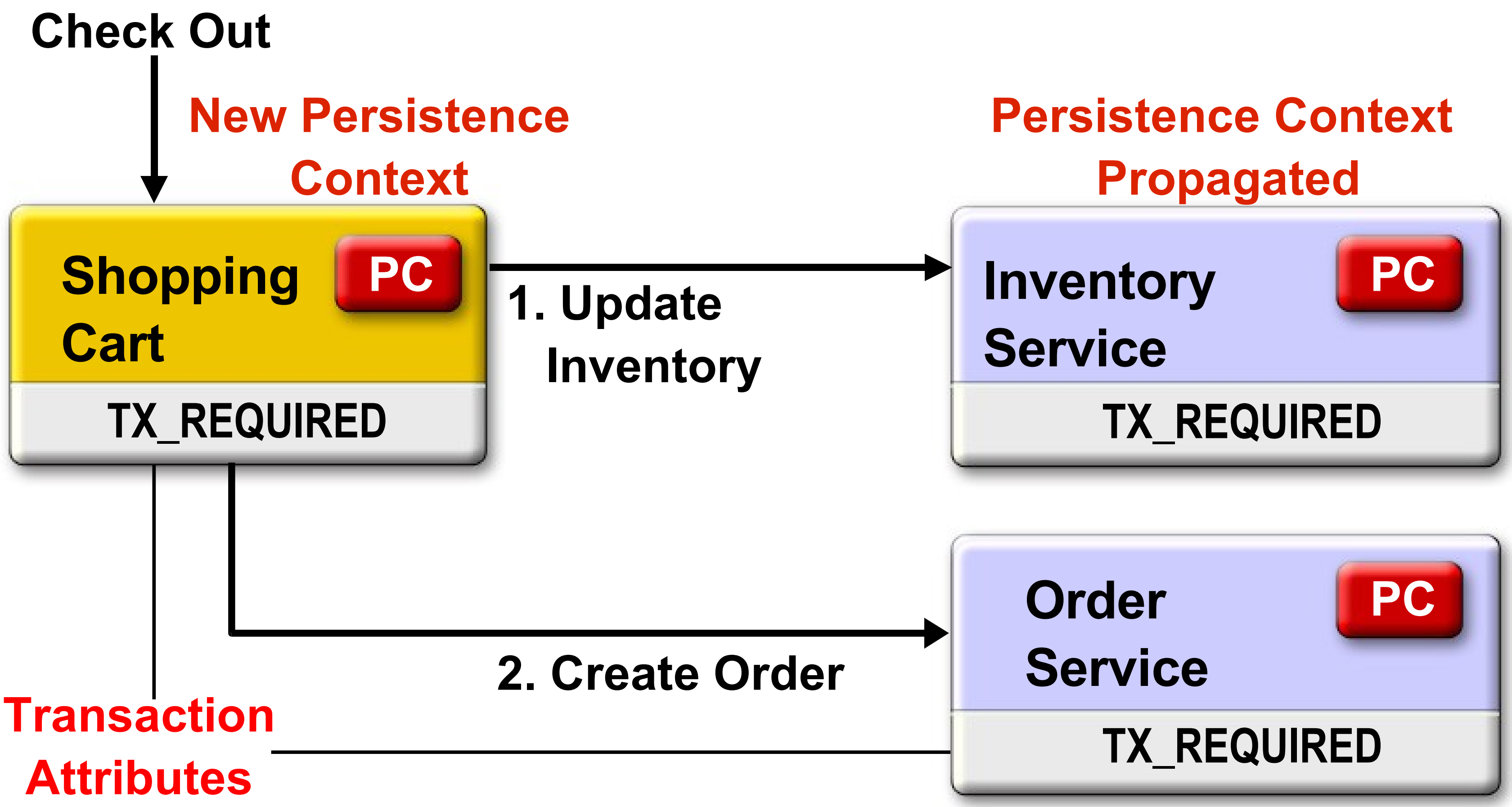
```
@Stateless public class ShoppingCartBean implements
   ShoppingCart {
@EJB InventoryService inv;
@EJB OrderService ord;

    public void checkout(Item i, Product p) {
        inv.createOrder(item);
        ord.updateInventory(Product p)
    }
}
```

Persistence context

Persistence Context Propagation

```
@Stateless public class OrderServiceBean implements
   OrderService {
@PersistenceContext EntityManager em1;
    public void createOrder(Item item) {
        em1.persist(new Order(item));
    }
}
@Stateless public class InventoryServiceBean implements
   InventoryService {
@PersistenceContext EntityManager em2;
    public void updateInventory(Product p) {
        Product product = em2.merge(p);
        . . .
    }
}
```

Declarative Transaction Management Example

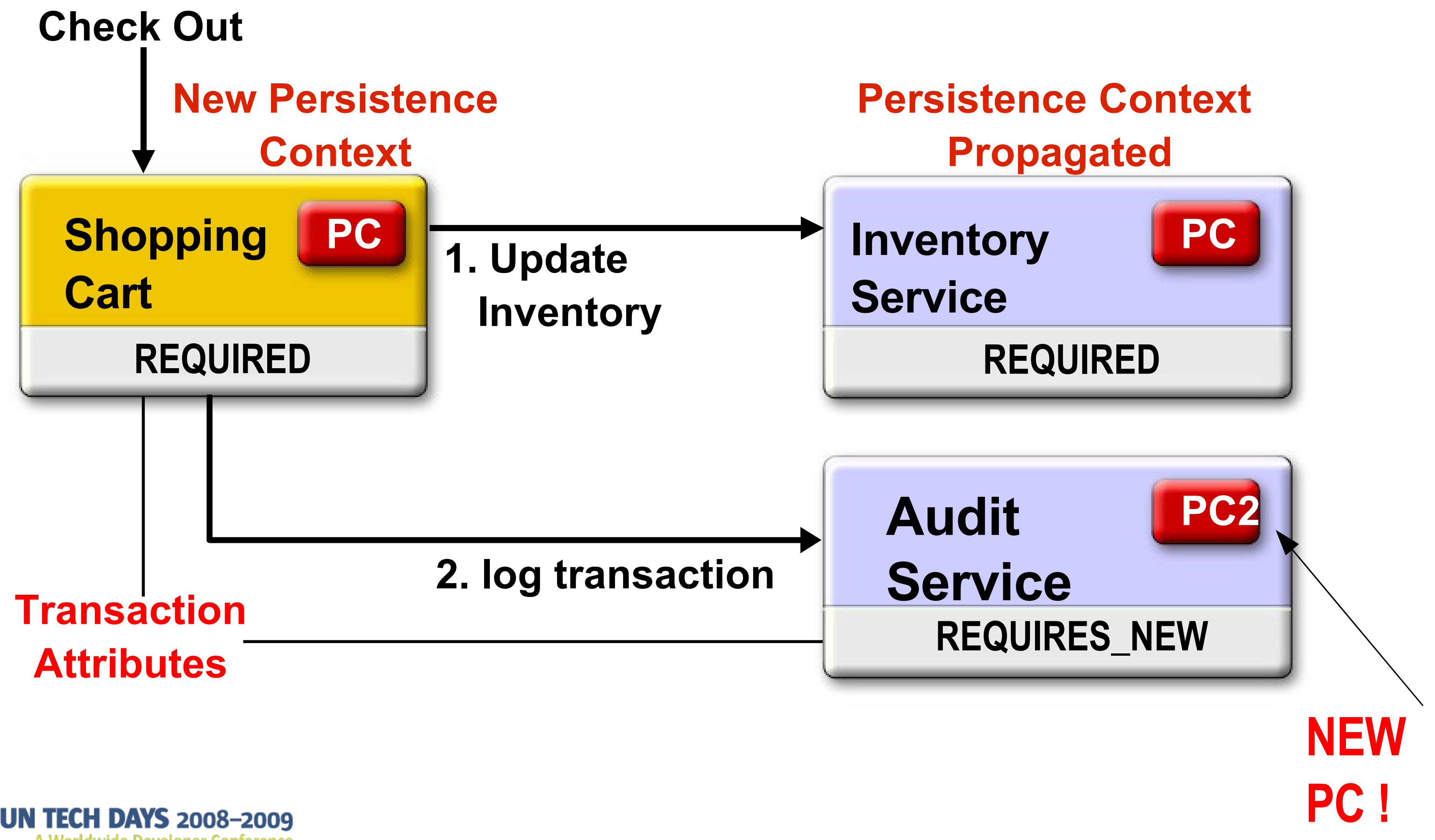


AuditServiceBean

```
@Stateless
public class AuditServiceBean implements AuditService {
    @PersistenceContext
    private EntityManager em;


   @TransactionAttribute(REQUIRES_NEW)
    public void logTransaction2(int id, String action) {
                LogRecord lr = new LogRecord(id, action);
        em.persist(lr);
    }
```

NEW
PC !

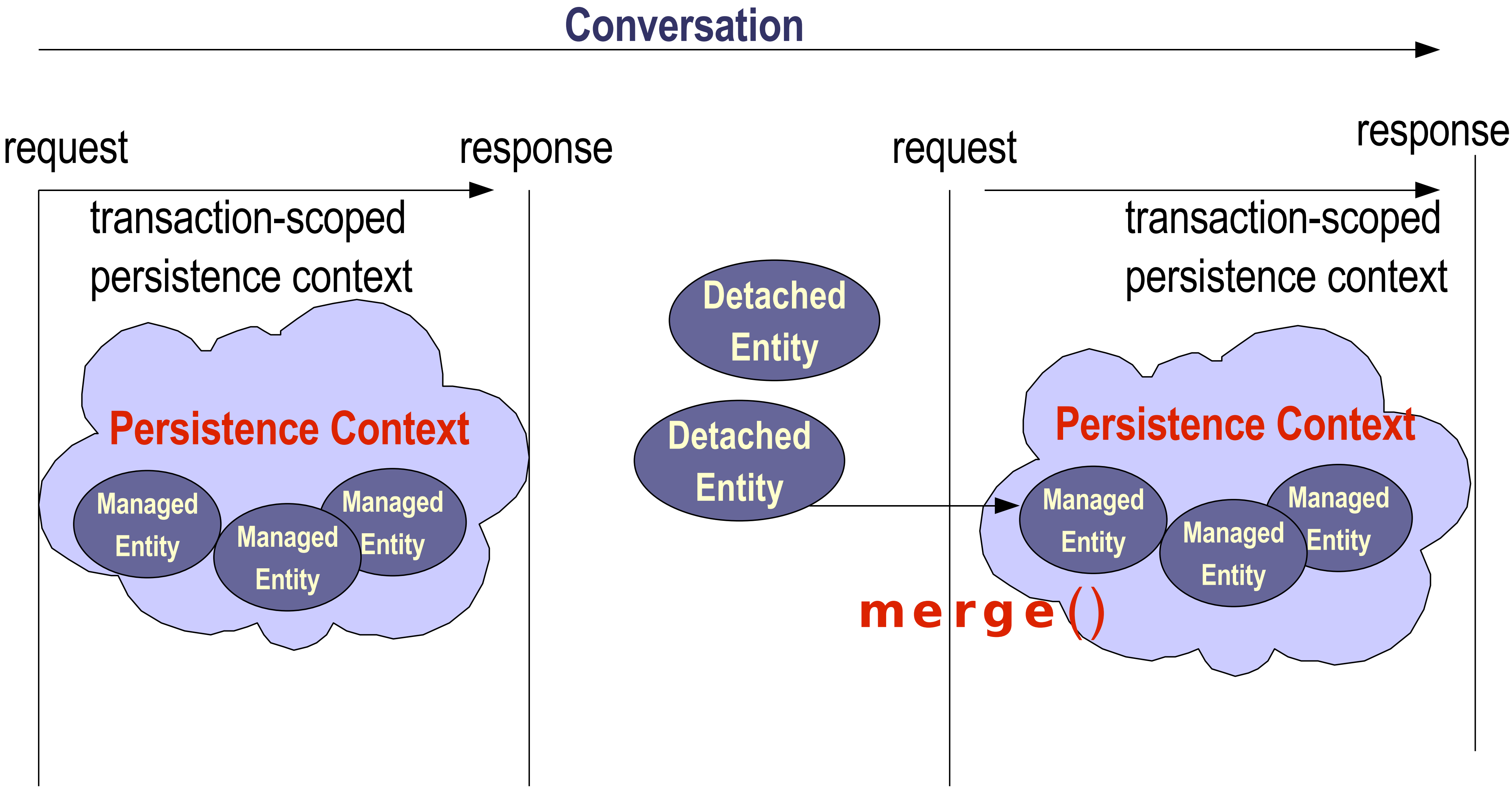
Declarative Transaction Management Example 2



Persistence Provider PC Transaction Features

- Attribute-level change tracking
- Only the minimal updates are sent to the database
- Orders INSERT, UPDATE and DELETE statements
- Minimizes database interactions
- EntityManager flush SQL prior to commit

Conversation with detached entity



Conversation with detached entity

```
@Stateless public ShoppingCartBean implements ShoppingCart {
 @PersistenceContext EntityManager entityManager;

 public OrderLine createOrderLine(Product product,Order order) {
      OrderLine orderLine = new OrderLine(order, product);
      entityManager.persist(orderLine);
      return (orderLine);
   }

   public OrderLine updateOrderLine(OrderLine orderLine){
       OrderLine orderLine2 =entityManager.merge(orderLine));
       return orderLine2;
   }
}
```

Managed entity

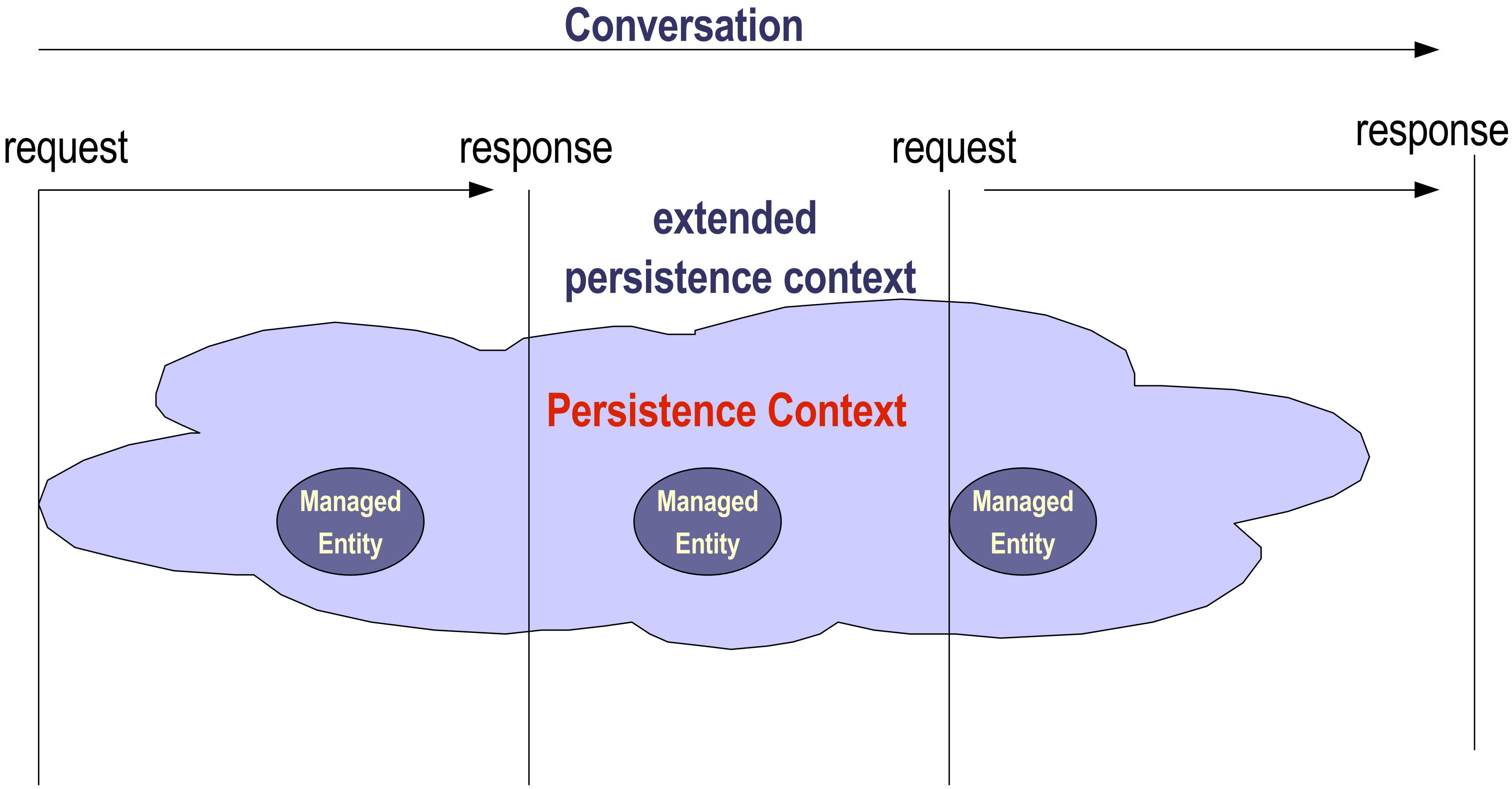
Detached entity

Managed entity

Types of Persistence Context

- Persistence Context
  - lifetime maybe transaction-scoped or extended
- Transaction-scoped persistence context
- Extended persistence context
  - spans **multiple** transactions

Conversation with Exented Persistence Context



Conversation

request

response

request

response

extended
persistence context

Persistence Context

Managed Entity

Managed Entity

Managed Entity

Extended Persistence Context

```
@Stateful public class OrderMgr {

   //Specify that we want an EXTENDED
   @PersistenceContext(type=PersistenceContextType.EXTENDED)
        EntityManager em;

   //Cached order
   private Order order;

   //create and cache order
   public void createOrder(String itemId) {
      //order remains managed for the lifetime of the bean
      Order order = new Order(cust);
      em.persist(order);
   }

  public void addLineItem(OrderLineItem li){
      order.lineItems.add(li);
  }
```

Managed entity

Managed entity

Extended Persistence Context

```
@Stateful public class DeptMgr {
   @PersistenceContext(type=PersistenceContextType.EXTENDED)
         EntityManager em;

   private Department dept;

   @TransactionAttribute(NOT_SUPPORTED)
   public void getDepartment(int deptId)  {
      dept = em.find(Department.class,deptId);
   }

   @TransactionAttribute(NOT_SUPPORTED)
  public void addEmployee(int empId){
       emp = em.find(Employee.class,empId);
       dept.getEmployees().add(emp);
       emp.setDepartment(dept);
  }
  @Remove
  @TransactionAttribute(REQUIRES_NEW)
   public void endUpdate(int deptId) {
      dept = em.find(Department.class,deptId);
   }
```

Persistence Context-Transactional vs. Extended

```
@Stateless
public class OrderMgr implements OrderService {

@PersistenceContext EntityManager em;
public void addLineItem(OrderLineItem li){
  // First, look up the order.
  Order order = em.find(Order.class, orderID);
  order.lineItems.add(li);
}
```

look up the order

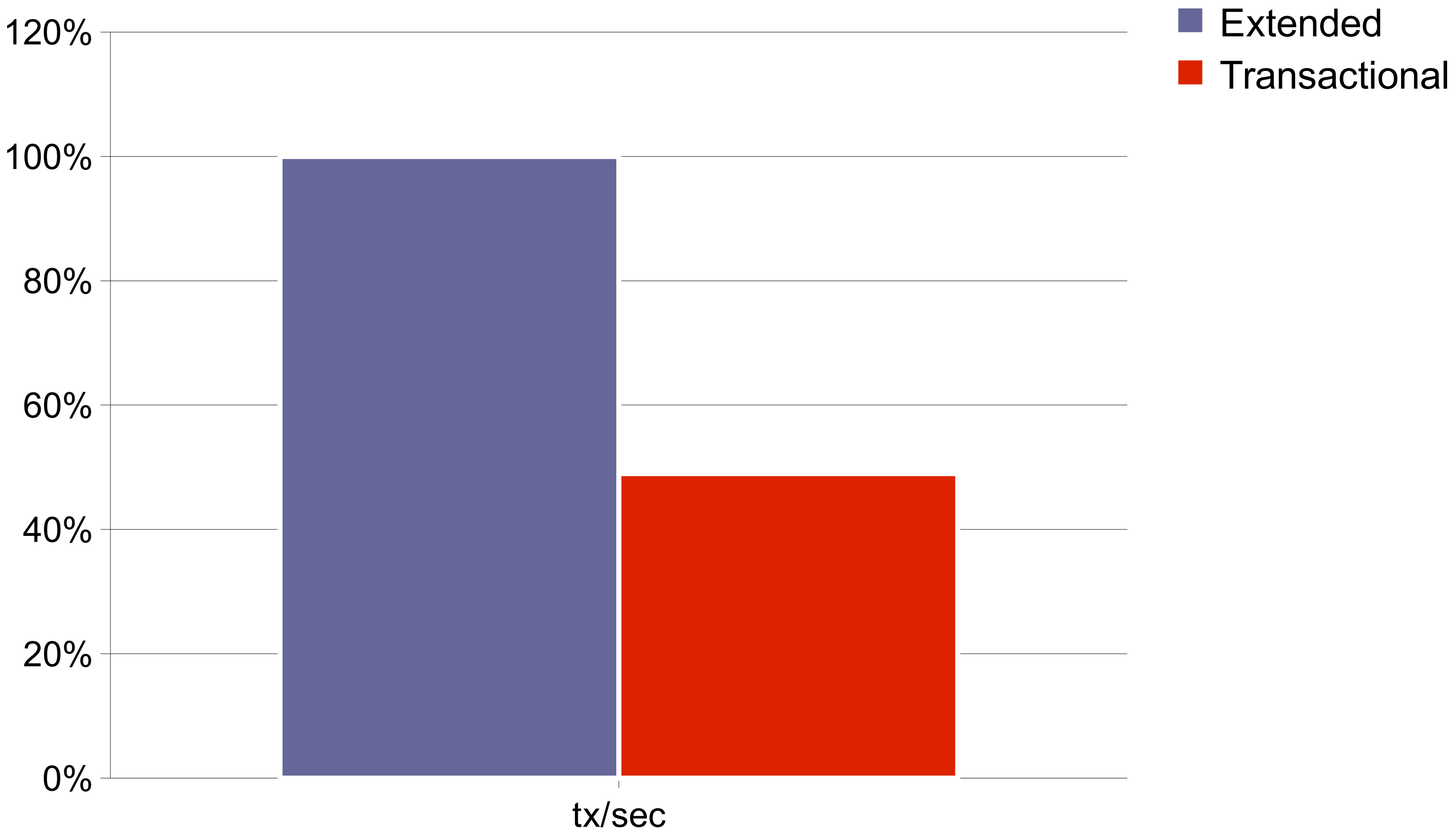
```
@Stateful
public class OrderMgr implements OrderService {
@PersistenceContext(type = PersistenceContextType.EXTENDED))
EntityManager em;
// Order is cached
Order order
public void addLineItem(OrderLineItem li){
     // No em.find invoked for the order object
     order.lineItems.add(li);
}
```

Managed entity

No em.find invoked

Persistence Context Micro Benchmark

- Micro benchmark with lots of lookups
- Persistence context is caching entities



Source: Internal benchmarks

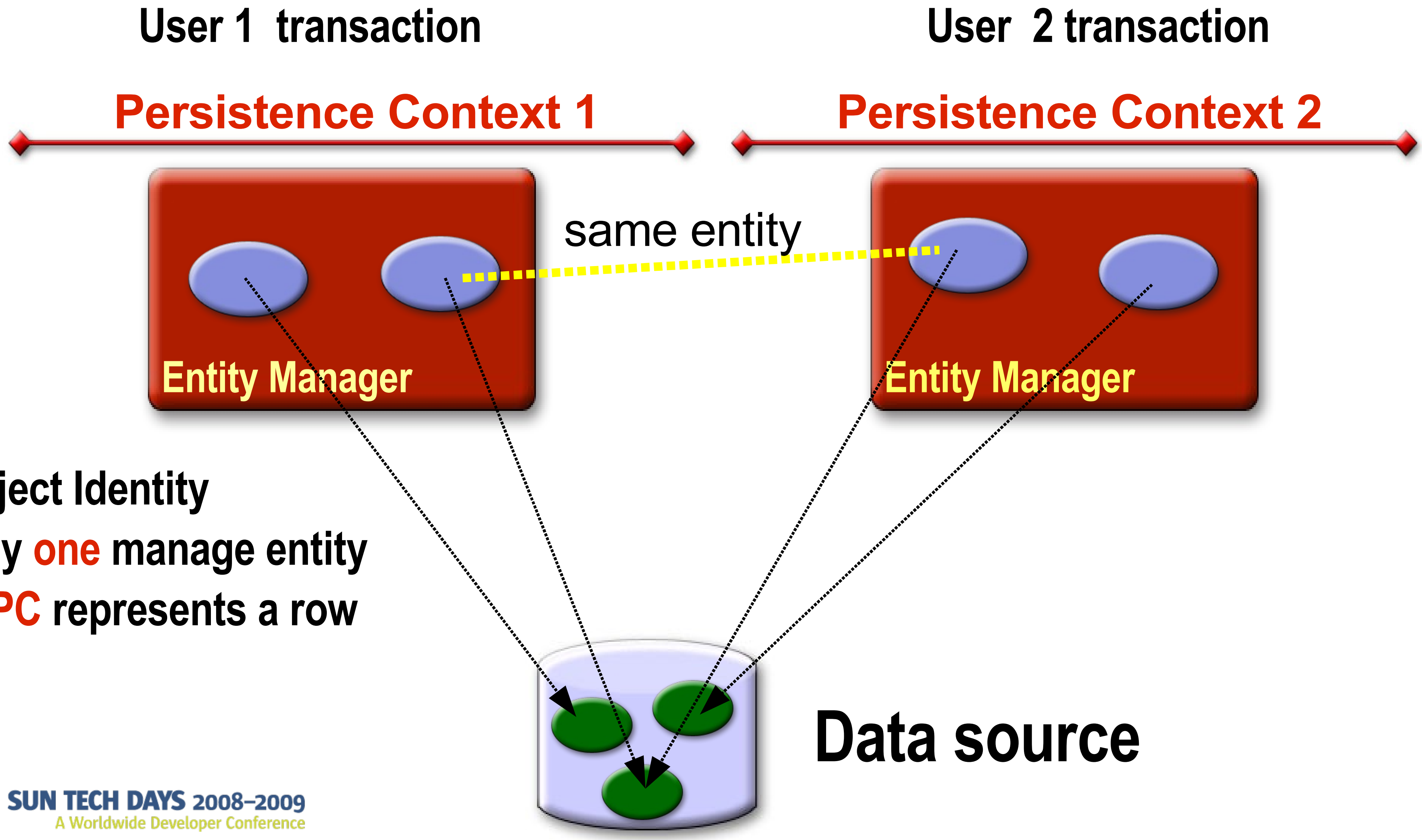
SEAM Conversations

SEAM injected

```
@Name("shopper")
@Scope(CONVERSATION)
public class BookingManager {

    @In EntityManager entityManager;
    private Booking booking;

    @Begin public void selectHotel(Hotel selectedHotel){
       hotel = em.merge(selectedHotel);
    }
    @End public void confirm()  {
       em.persist(booking);
    }
}
```

SEAM conversation

Concurrency and Persistence Context



Object Identity
only one manage entity
in PC represents a row

Optimistic versus Pessimistic Concurrency

- Optimistic Concurrency
  - Pros—No database locks held
  - Cons—Requires a version attribute in schema
    - user or app must refresh and retry failed updates
  - Suitable when application has **few parallel updates**
- Pessimistic Concurrency
  - Lock the row when data is read in
    - database locks the row upon a select
    - (SELECT . . . FOR UPDATE [NOWAIT])
  - Pros—Simpler application code
  - Cons—Database locks
    - limits concurrent access to the data = **scalability** ↓
    - May cause **deadlocks**
    - Not in JPA 1.0 (vendor specific), **supported** in **JPA 2.0**
  - Suitable when application has **many parallel updates**

Preventing Parallel Updates

use @Version for optimistic locking

```
public class Employee {
    @ID int id;
    @Version int version;
    ...
```

Can be int, Integer,
short, Short, long,
Long, Timestamp

Used by persistence manager , Results in following SQL

```
"UPDATE Employee SET ..., version = version + 1
              WHERE id = ? AND version = readVersion"
```

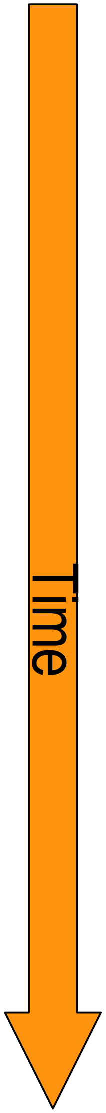
Version Updated when transaction commits, merged or acquiring a write lock

OptimisticLockException if mismatch

Not used by the application!

Preventing Parallel Updates – 1

```
tx1.begin();
//Joe's employee id is 5
//e1.version == 1
e1 = findPartTimeEmp(5);

//Joe's current rate is $9
e1.raise(2);

tx1.commit();
//e1.version == 2 in db

//Joe's rate is $11
```



```
tx2.begin();
//Joe's employee id is 5
//e1.version == 1
e1 = findPartTimeEmp(5);




//Joe's current rate is $9
if(e1.getRate() < 10)
   e1.raise(5);


//e1.version == 1 in db?
tx2.commit();
~~//Joe's rate is $14~~
//OptimisticLockException
```

Preventing Stale Data JPA 1.0

- Perform read or write locks on entities
- Prevents non-repeatable reads in JPA

entityManager.lock(entity, READ);

perform a version check on entity before commit

OptimisticLockException if mismatch

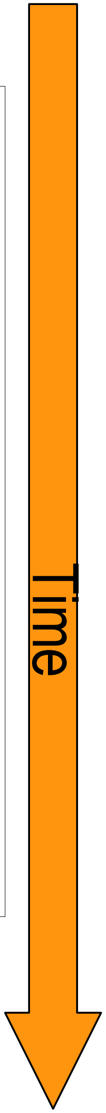
entityManager.lock(entity, WRITE);

perform a version check on entity

OptimisticLockException if mismatch
and increment version before commit

Preventing Stale Data

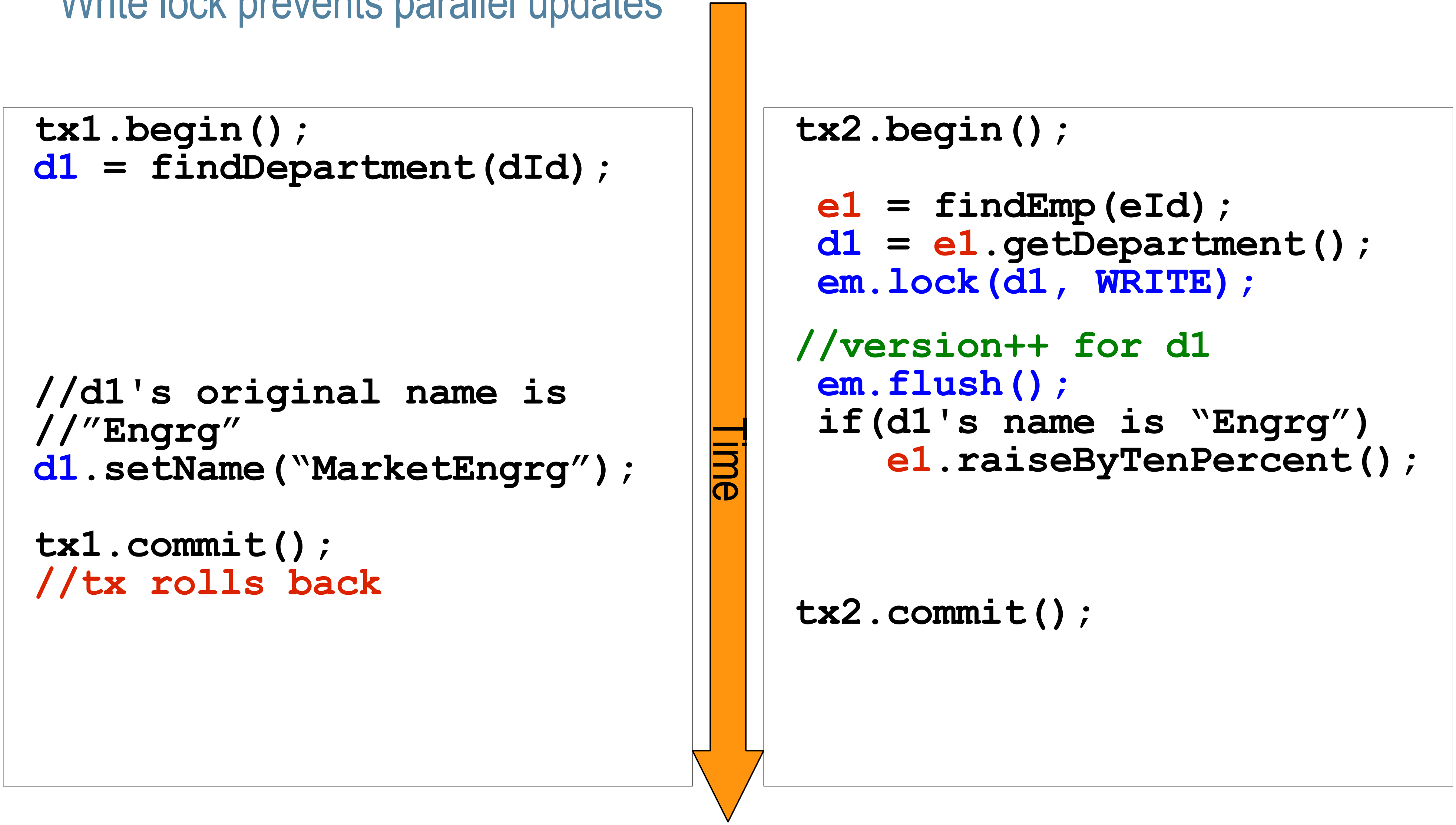
```
tx1.begin();
d1 = findDepartment(dId);

//d1's original name is
//”Engrg”
d1.setName(“MarketEngrg”);

tx1.commit();
```



```
tx2.begin();

e1 = findEmp(eId);
d1 = e1.getDepartment();
em.lock(d1, READ);
if(d1's name is “Engrg”)
  e1.raiseByTenPercent();



//Check d1.version in db
tx2.commit();
~~//e1 gets the raise he does~~
~~//not deserve~~
//Transaction rolls back
```

Preventing Parallel Updates – 2

Write lock prevents parallel updates



```
tx1.begin();
d1 = findDepartment(dId);






//d1's original name is
//”Engrg”
d1.setName(“MarketEngrg”);

tx1.commit();
//tx rolls back
```

```
tx2.begin();

 e1 = findEmp(eId);
 d1 = e1.getDepartment();
 em.lock(d1, WRITE);

//version++ for d1
 em.flush();
 if(d1's name is “Engrg”)
    e1.raiseByTenPercent();



tx2.commit();
```

Bulk Updates

- Update directly against the database, can be Faster But
  - By pass `EntityManager`
  - `@Version` will not be updated
  - Entities in PC not updated

```
>tx.begin();
int id = 5;  //Joe's employee id is 5
e1 = findPartTimeEmp(id); //Joe's current rate is $9

//Double every employee's salary
em.createQuery(
   “Update Employee set rate = rate * 2”).executeUpdate();

//Joe's rate is still $9 in this persistence context
if(e1.getRate() < 10)
   e1.raiseByFiveDollar();

tx.commit();
//Joe's salary will be $14
```

JPA 2.0 Locks

- JPA1.0 only supported optimistic locking, JPA 2.0 also supports pessimistic locks
- **JPA 2.0** LockMode values :
  - > OPTIMISTIC (= READ)
  - > OPTIMISTIC_FORCE_INCREMENT (= WRITE)
  - > PESSIMISTIC
  - > PESSIMISTIC_FORCE_INCREMENT

database locks the row
(SELECT . . . FOR UPDATE)

- Multiple places to specify lock

JPA 2.0 Locking

```
//Read then lock:

Account acct = em.find(Account.class, acctId);

// Decide to withdraw $100 so lock it for update

em.lock(acct, PESSIMISTIC);

int balance = acct.getBalance();

acct.setBalance(balance - 100);
```

Lock after read, risk **stale**, could cause `OptimisticLock Exception`

```
//Read and lock:

Account acct = em.find(Account.class,
acctId,PESSIMISTIC);

// Decide to withdraw $100 (already locked)

int balance = acct.getBalance();

acct.setBalance(balance - 100);
```

Locks **longer**, could cause bottlenecks, deadlock

JPA 2.0 Locking

Trade-offs:

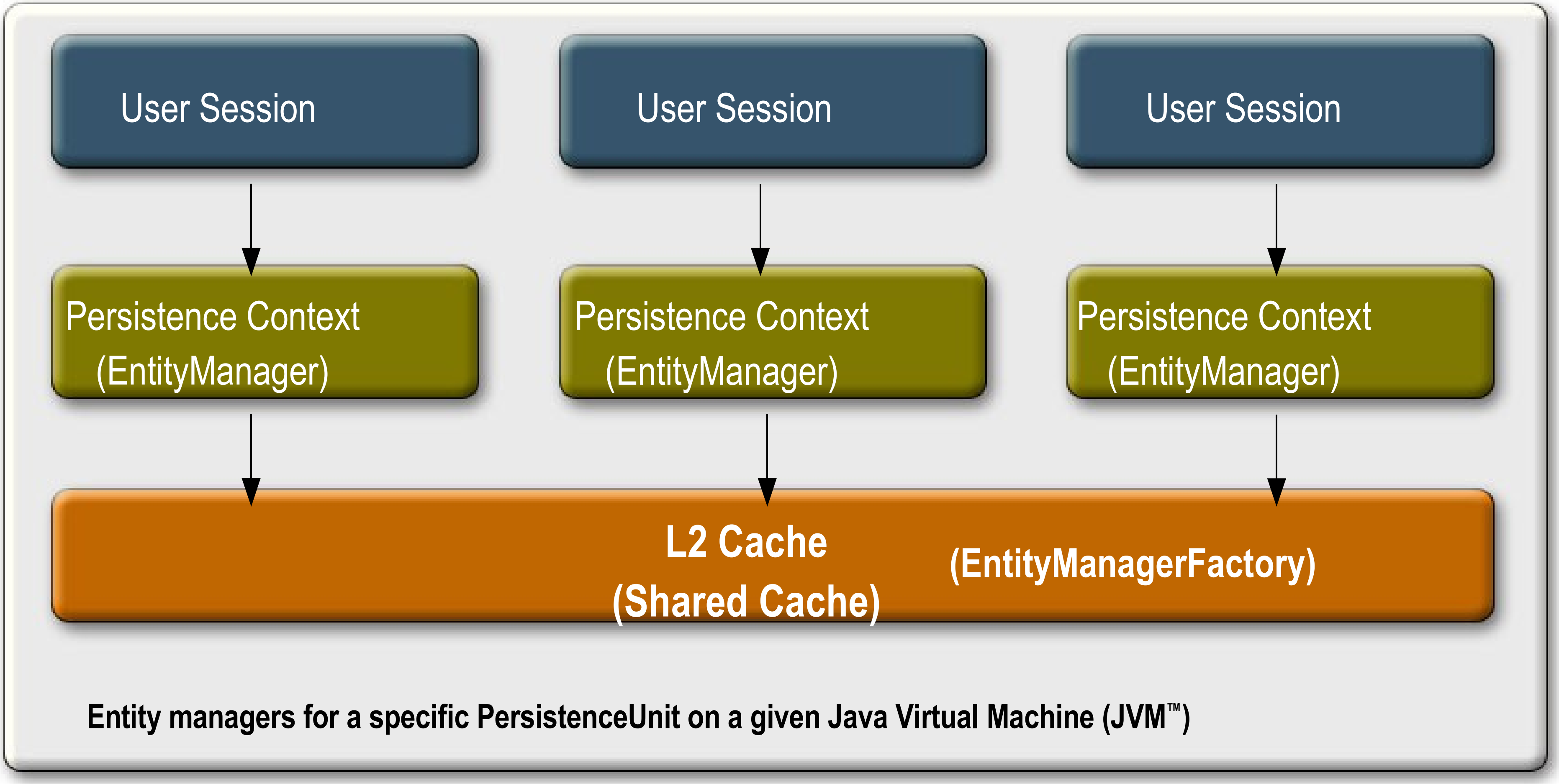
- lock earlier : **risk bad scalability, deadlock**
- Lock later : risk **stale** data for update, get optimistic lock exception

```
// read then lock and refresh
Account acct = em.find(Account.class, acctId);
// Decide to withdraw $100 - lock and refresh
em.refresh(acct, PESSIMISTIC);
int balance = acct.getBalance();
acct.setBalance(balance - 100);
```

“right” approach depends on requirements

L2 cache shared across transactions and users

Putting it all together



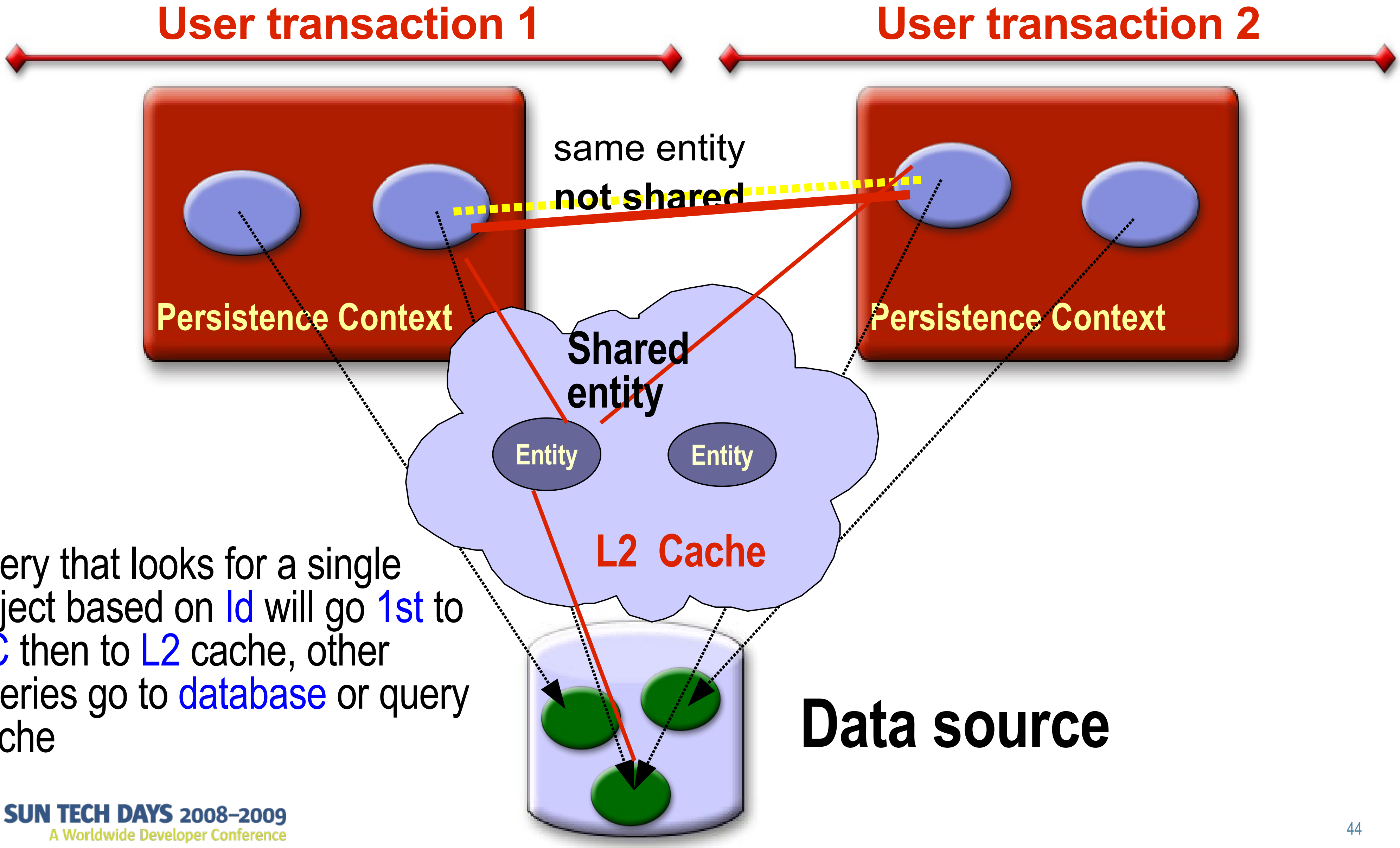
User Session
User Session
User Session
Persistence Context (EntityManager)
Persistence Context (EntityManager)
Persistence Context (EntityManager)
L2 Cache (Shared Cache)
(EntityManagerFactory)

Entity managers for a specific PersistenceUnit on a given Java Virtual Machine (JVM™)

Second-level Cache

- L2 Cache shares entity state across various persistence contexts
  - If caching is enabled, entities not found in persistence context, will be loaded from L2 cache, if found
- Best for read-mostly classes
- L2 Cache is Vendor specific
  - Java Persistence API 1.0 does not specify L2 support
  - Java Persistence API 2.0 has basic cache operations
  - Most persistence providers-- Hibernate, EclipseLink, OpenJPA ... provide second-level cache(s)

L2 Cache



query that looks for a single object based on Id will go 1st to PC then to L2 cache, other queries go to database or query cache

L2 Caching

- Pros:
  - > avoids database access for already loaded entities
    - faster for reading frequently accessed unmodified entities
- Cons
  - > memory consumption for large amount of objects
  - > Stale data for updated objects
  - > Concurrency for write (optimistic lock exception, or pessimistic lock)
    - Bad scalability for frequent or concurrently updated entities

L2 Caching

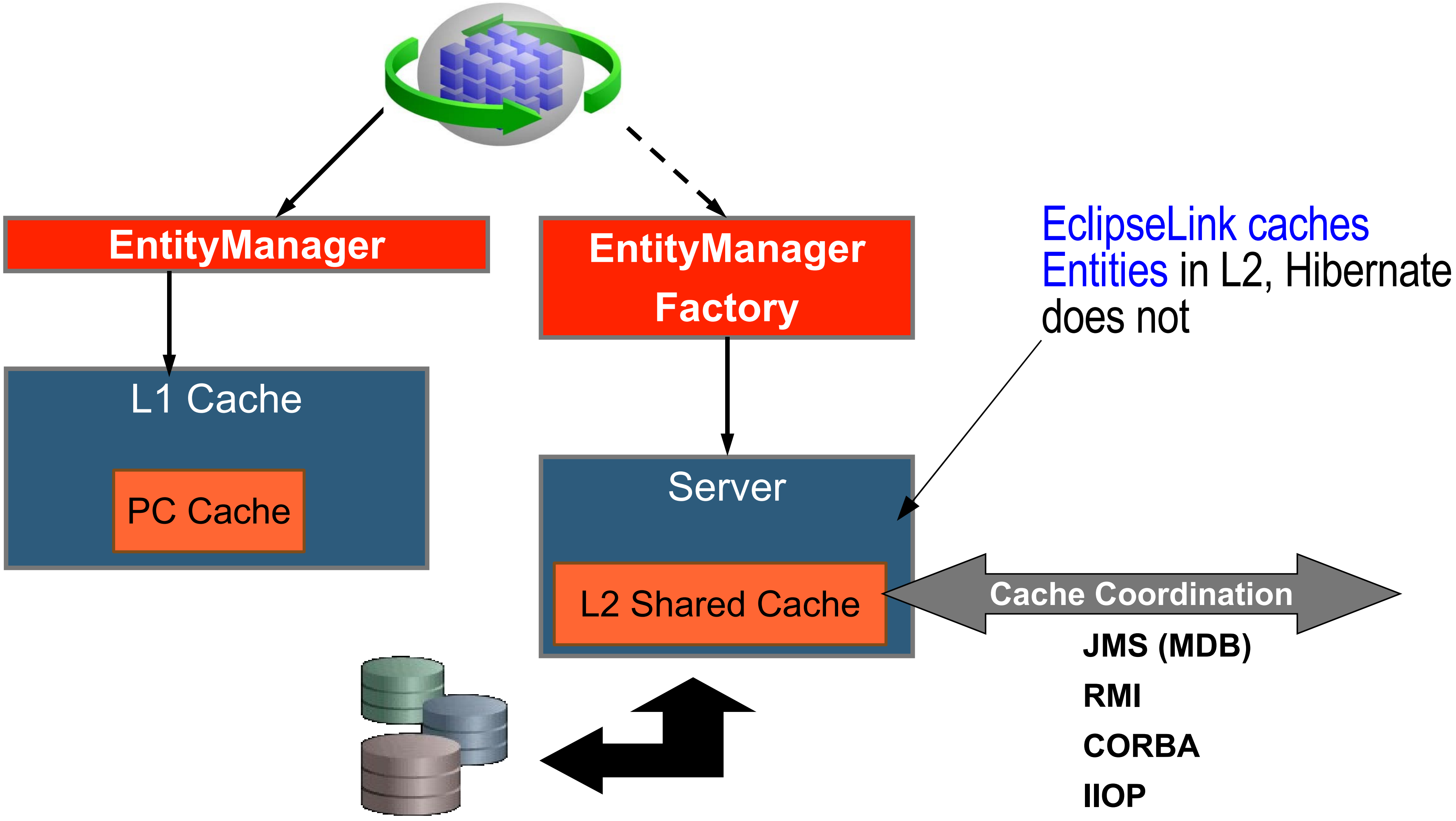
- Configure L2 caching for entities that are
  - read often
  - modified infrequently
  - Not critical if stale
- protect any data that can be concurrently modified with a locking strategy
  - Must handle optimistic lock failures on flush/commit
  - configure expiration, refresh policy to minimize lock failures
- Configure Query cache
  - Useful for queries that are run frequently with the same parameters, for not modified tables

JPA 2.0 Shared Cache API

- entity cache shared across persistence unit
  - Accessible from EntityManagerFactory
- Supports only very basic cache operations
  - Can be extended by vendors

```
public class Cache {
    //checks if object is in IdentityMap
    public boolean contains(Class class, Object pk);
    // invalidates object in the IdentityMap
    public void evict(Class class, Object pk);
    public void evict(Class class); // invalidates the class in the IdentityMap.
    public void evictAll(); // Invalidates all classes in the IdentityMap
}
```

EclipseLink Caching Architecture



EntityManager

EntityManager Factory

L1 Cache

PC Cache

Server

L2 Shared Cache

Cache Coordination

JMS (MDB)

RMI

CORBA

IIOP

EclipseLink caches Entities in L2, Hibernate does not

EclipseLink Extensions - L2 Caching

- Default: Entities read are L2 cached
- Cache Configuration by Entity type or Persistence Unit
  - > **You can disable L2 cache**
- Configuration Parameters
  - > Cache isolation, type, size, expiration, coordination, invalidation,refreshing
  - > Coordination (cluster-messaging)
    - Messaging: JMS, RMI, RMI-IIOP, …
    - Mode: SYNC, SYNC+NEW, INVALIDATE, NONE

EclipseLink Mapping Extensions

```
@Entity
@Table(name="EMPLOYEE")
@Cache (
    type=CacheType.WEAK,
    isolated=false,
    expiry=600000,
    alwaysRefresh=true,
    disableHits=true,
    coordinationType=INVALIDATE_CHANGED_OBJECTS
    )
public class Employee implements Serializable {
    ...
}
```

Type=

Full: objects never flushed unless deleted or evicted

weak: object will be garbage collected if not referenced

=true

disables L2 cache

@Cache

- type, size,isolated, expiry, refresh, cache usage, coordination
- Cache usage and refresh query hints

Hibernate L2 Cache

- Hibernate L2 cache is **not configured by default**
- Hibernate L2 does not cache Entities. Hibernate caches Id and state
- Hibernate L2 cache is pluggable
  - EHCache, OSCache, SwarmCacheProvider **(JVM)**
  - JBoss TreeCache **Cluster**
  - Can plug in others like Terracotta

		Cache Concurrency Strategy		
Cache	**Type**	**Read-only**	**Read-write**	**Transactional**
EHCache	memory, disk	Yes	Yes	
OSCache	memory, disk	Yes	Yes	
SwarmCache	clustered	Yes		
JBoss Cache	clustered	Yes		Yes

Hibernate L2 Cache

not configured by default

```
<!-- optional configuration file parameter -->
net.sf.ehcache.configurationResourceName=/name_of_configuration_resource
```

```
@Entity
@Cache(usage =
   CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
 public Class Country {
    private String name;
    ...
}
```

Cache Concurrency Strategy
must be supported by cache provider

OpenJPA L2 Caching

- OpenJPA L2 caches object data and JPQL query results
- Updated when data is loaded from database and after changes are successfully committed
- For cluster caches are notified when changes are made
- Can plug in implementations, such as Tangosol’s Coherence product
- several cache eviction strategies:
  - Time-to-live settings for cache instances

```
@Entity @DataCache(timeout=5000)
public class Person { ... }
```

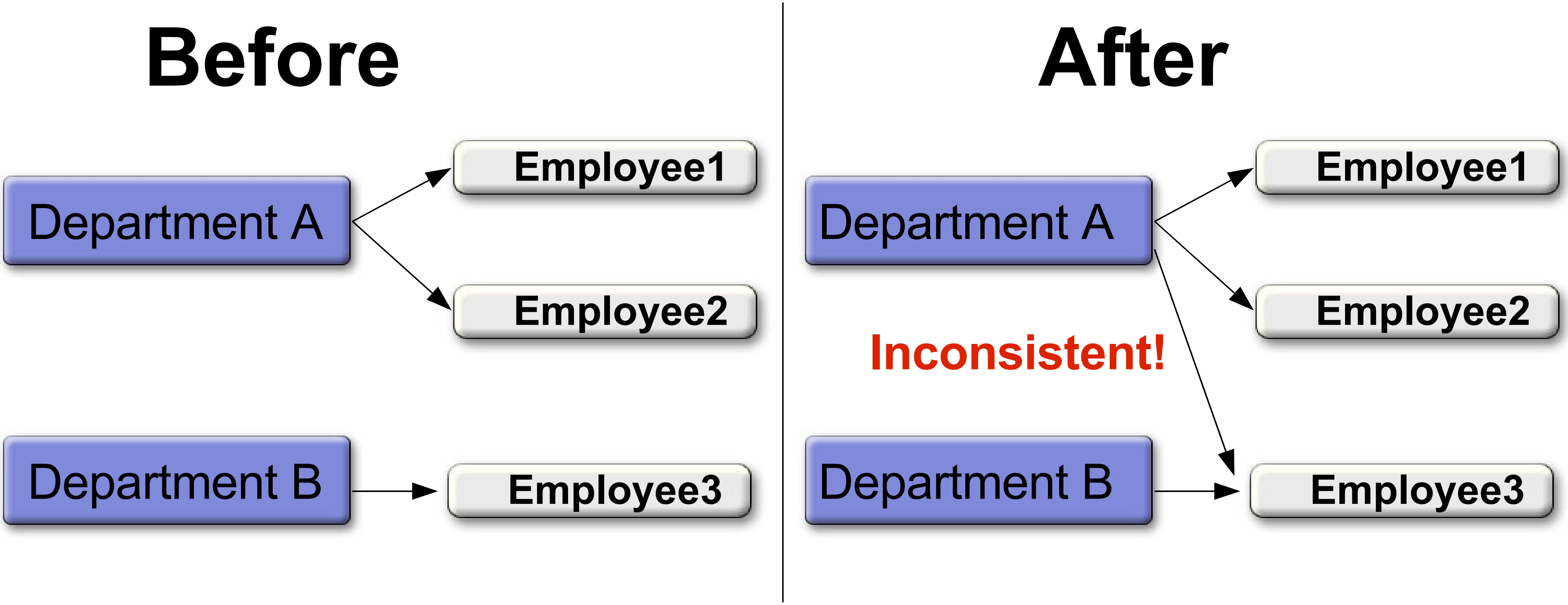
Agenda

- >Entity Manager
- >Persistence Context
- >**Entities**
- >Schema and Queries
- >Transaction

Maintaining Relationship

- Application bears the responsibility of maintaining relationship between objects

```
deptA.getEmployees().add(e3);
```



Example – Domain Model

```
@Entity public class Employee {
   @Id private int id;
   private String firstName;
   private String lastName;
   @ManyToOne(fetch=LAZY)
   private Department dept;
      ...
}
@Entity public class Department {
   @Id private int id;
   private String name;
   @OneToMany(mappedBy = "dept", fetch=LAZY)
   private Collection<Employee> emps = new ...;
         ...
 }
```

Example – Managing Relationship

INCORRECT

```
public int addNewEmployee(...) {
   Employee e = new Employee(...);
   Department d = new Department(1, ...);

   e.setDepartment(d);
   //Reverse relationship is not set
   em.persist(e);
   em.persist(d);

   return d.getEmployees().size();
}
```

Example – Managing Relationship

CORRECT

```
public int addNewEmployee(...) {
   Employee e = new Employee(...);
   Department d = new Department(1, ...);

   e.setDepartment(d);
   d.getEmployees().add(e);
   em.persist(e);
   em.persist(d);

   return d.getEmployees().size();
}
```

Navigating Relationships

Data fetching strategy

- EAGER – immediate
- LAZY – loaded only when needed
- LAZY is good for large objects and/or with relationships with deep hierarchies

Lazy loading and JPA

```
@Entity public class Department {
   @Id private int id;
   @OneToMany(mappedBy = "dept")
   private Collection<Employee> emps ;
         ...
 }
```

- Default FetchType is LAZY for 1:m and m:n relationships
  - benefits large objects and relationships with deep hierarchies
- However for use cases where data is needed can cause n+1 selects
- **LAZY – N + 1 problem:**

```
SELECT d.id, ... FROM Department d // 1 time
SELECT e.id, ... FROM Employee e
   WHERE e.deptId = ? // N times
```

Lazy loading and JPA

```
@Entity public class Department {
   @Id private int id;
   @OneToMany(mappedBy = "dept", fetch=EAGER)
   private Collection<Employee> employees ;
          ...
 }
```

Can cause Cartesian product

- Relationship can be Loaded Eagerly
  - > But if you have several related relationships, could load too much !

OR

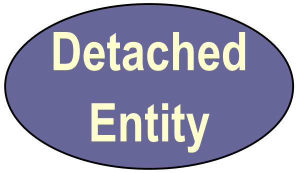
- Temporarily override the LAZY fetch type, use Join Fetch in a query:

```
@NamedQueries({ @NamedQuery(name="getItEarly",
       query="SELECT d FROM Department d JOIN FETCH d.employees")})

public class Department{
.....
}
```

Lazy loading and JPA

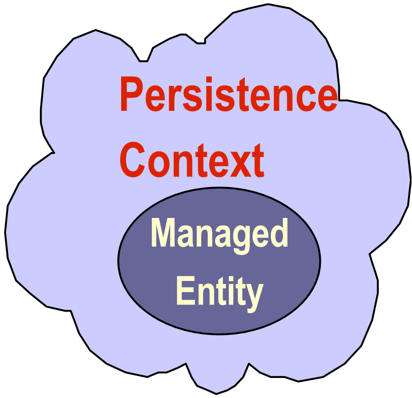
- Capture generated SQL
  - persistence.xml file:<property name="toplink.logging.level" value="FINE">
- Test run use cases and examine the SQL statements
  - optimise the number of SQL statements executed!
  - only retrieve the data your application needs!
- Lazy load large (eg BLOB) attributes and relationships that are not used often
- Override to Eagerly load in use cases where needed

Navigating Relationships



- Accessing a LAZY relationship from a detached entity
  - If not loaded , Causes an exception
- Solutions:
  - Use JOIN FETCH
  - Or Set Fetch type to EAGER
  - Or Access the collection before entity is detached:

```
d.getEmployees().size();
```



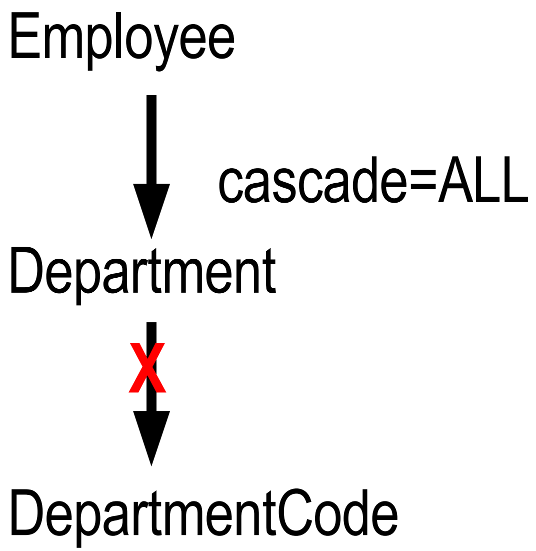
Navigating Relationships

Data fetching strategy

- Cascade specifies operations on relationships
  - ALL, PERSIST, MERGE, REMOVE, REFRESH
  - The default is do **nothing**
- Avoid MERGE, ALL with deep hierarchies
  - If want to do it, limit the scope

Using Cascade

```
@Entity public class Employee {
   @Id private int id;
   private String firstName;
   private String lastName;
   @ManyToOne(cascade=ALL, fetch=LAZY)
   private Department dept;
      ...
}
@Entity public class Department {
   @Id private int id;
   private String name;
   @OneToMany(mappedBy = "dept"
           cascade=ALL, fetch=LAZY)
   private Collection<Employee> emps = new ...;
   @OneToMany
   private Collection<DepartmentCode> codes;
         ...
 }
```



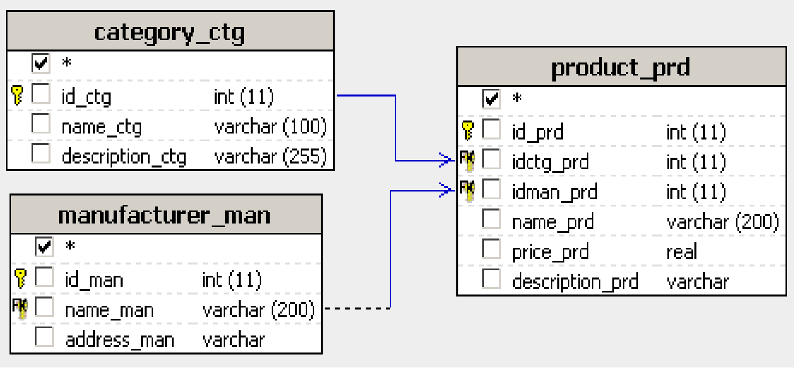
Agenda

- >Entity Manager
- >Persistence Context
- >Entities
- >**Schema and Queries**
- >Transaction

Database design Basic foundation of performance

- Smaller tables use less disk, less memory, can give better performance
  - > Use as small data types as possible
  - > use as small primary key as possible
  - > Vertical Partition:
    - ●split large, infrequently used columns into a separate one-to-one table
- Use good indexing
  - > Indexes Speed up Querys
  - > Indexes slow down Updates
  - > Index columns frequently used in Query Where clause

Normalization



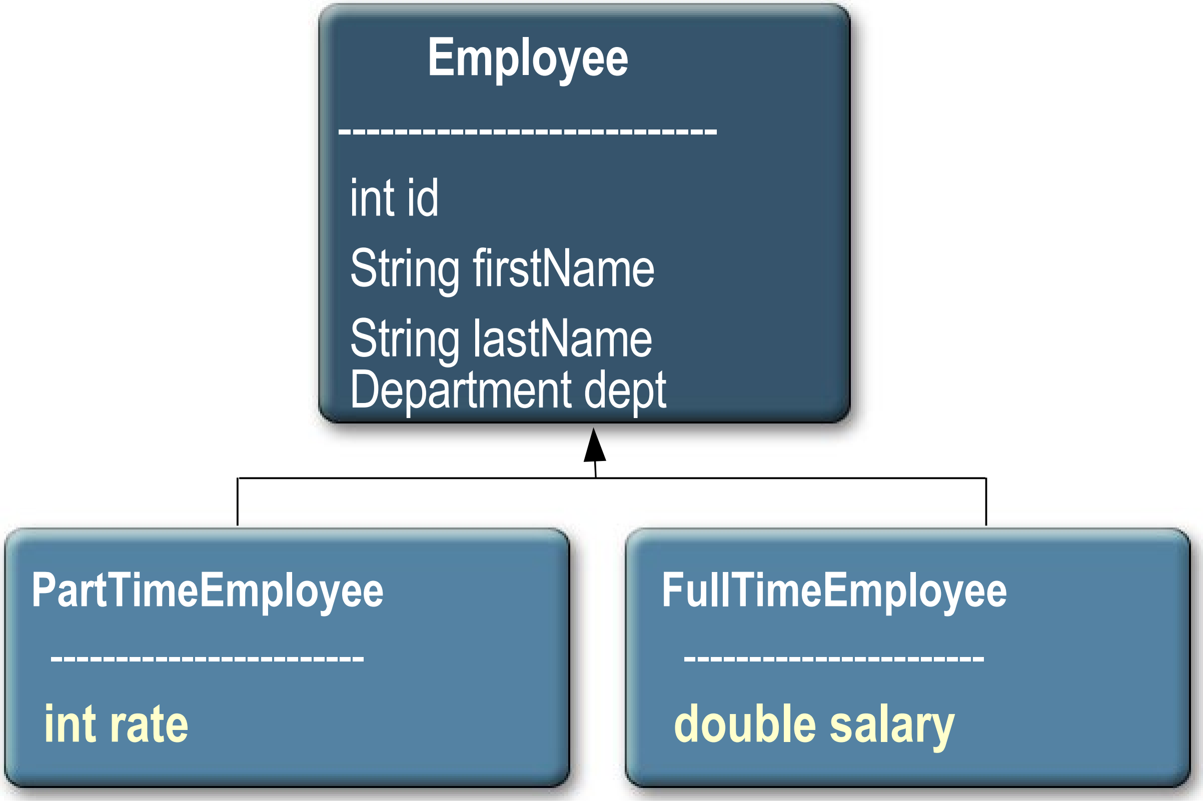
category_ctg			
	☑	*	
🔑	□	id_ctg	int (11)
	□	name_ctg	varchar (100)
	□	description_ctg	varchar (255)

manufacturer_man			
	☑	*	
🔑	□	id_man	int (11)
FK	□	name_man	varchar (200)
	□	address_man	varchar

product_prd			
	☑	*	
🔑	□	id_prd	int (11)
FK	□	idctg_prd	int (11)
FK	□	idman_prd	int (11)
	□	name_prd	varchar (200)
	□	price_prd	real
	□	description_prd	varchar

- Normalization Eliminates redundant data
  - > updates are usually faster.
    - there's less data to change.
- However Normalized database causes joins for queries
  - > Queries maybe slower
  - > Normalize then maybe De-normalize frequently read columns and cache them

Mapping Inheritance Hierarchies



Employee

----

int id

String firstName

String lastName
Department dept

PartTimeEmployee

----

int rate

FullTimeEmployee

----

double salary

Single Table Per Class

Benefits `@Inheritance(strategy=SINGLE_TABLE)`

- Simple
- **No joins** required
  - can be **fast** for Queries

Drawbacks

- Not normalized
  - Wastes space
- Requires columns corresponding to subclasses' state to be null
- Table can have too many columns
  - Larger tables= more data, can have negative affects on performance

–

EMPLOYEE	

ID	Int PK,
FIRSTNAME	varchar(255),
LASTNAME	varchar(255),
DEPT_ID	int FK,
RATE	int NULL,
SALARY	double NULL,
DISCRIM	varchar(30)

Joined Subclass

`@Inheritance(strategy=JOINED)`

Benefits

- Normalized database
  - > Better for storage
- Database view same as domain model
- Easy to evolve domain model

Drawbacks

- Queries cause joins
  - > Slower queries
  - > Poor performance for deep hierarchies, polymorphic queries and relationships

EMPLOYEE	
ID	int PK,
FIRSTNAME	varchar(255),
LASTNAME	varchar(255),
DEPT_ID	int FK,

PARTTIMEEMPLOYEE	
ID	int PK FK,
RATE	int NULL

FULLTIMEEMPLOYEE	
ID	int PK FK,
SALARY	double NULL

Table Per Class

`@Inheritance(strategy=TABLE_PER_CLASS)`

Benefits

- No need for joins to read entities of same type
  - > Faster reads

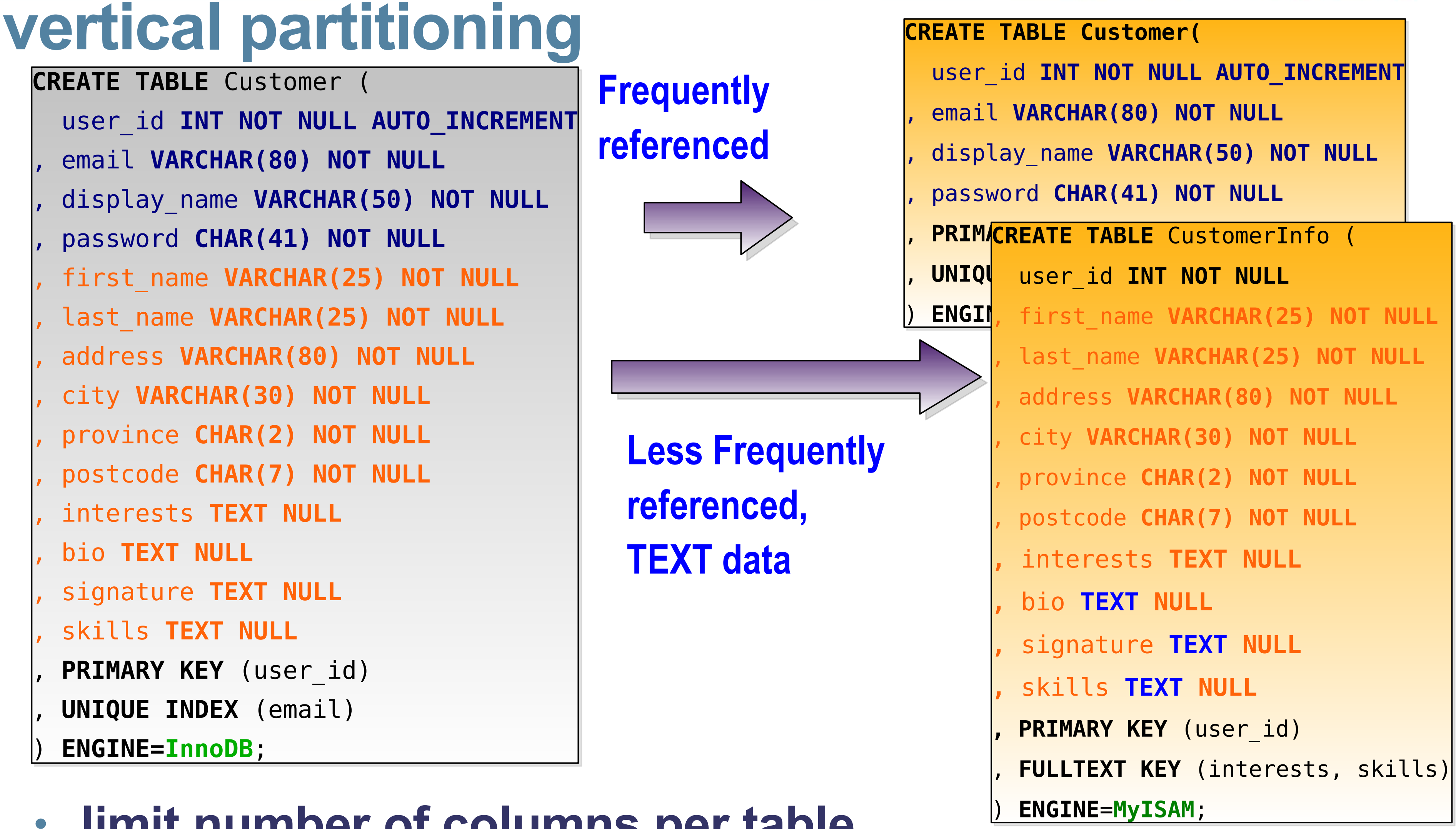
Drawbacks

- Not normalized
  - > Wastes space
- Polymorphic queries cause union (all employees)
  - > Poor performance
- This strategy is not mandatory

PARTTIMEEMPLOYEE	
ID	int PK,
FIRSTNAME	varchar(255),
LASTNAME	varchar(255),
DEPT_ID	int FK,
RATE	int NULL

FULLTIMEEMPLOYEE	
ID	int PK,
FIRSTNAME	varchar(255),
LASTNAME	varchar(255),
DEPT_ID	int FK,
SALARY	double NULL

vertical partitioning



```
CREATE TABLE Customer (
  user_id INT NOT NULL AUTO_INCREMENT
, email VARCHAR(80) NOT NULL
, display_name VARCHAR(50) NOT NULL
, password CHAR(41) NOT NULL
, first_name VARCHAR(25) NOT NULL
, last_name VARCHAR(25) NOT NULL
, address VARCHAR(80) NOT NULL
, city VARCHAR(30) NOT NULL
, province CHAR(2) NOT NULL
, postcode CHAR(7) NOT NULL
, interests TEXT NULL
, bio TEXT NULL
, signature TEXT NULL
, skills TEXT NULL
, PRIMARY KEY (user_id)
, UNIQUE INDEX (email)
) ENGINE=InnoDB;
```

Frequently referenced

```
CREATE TABLE Customer(
  user_id INT NOT NULL AUTO_INCREMENT
, email VARCHAR(80) NOT NULL
, display_name VARCHAR(50) NOT NULL
, password CHAR(41) NOT NULL
, PRIM
, UNIQU
) ENGI
```

Less Frequently referenced, TEXT data

```
CREATE TABLE CustomerInfo (
  user_id INT NOT NULL
, first_name VARCHAR(25) NOT NULL
, last_name VARCHAR(25) NOT NULL
, address VARCHAR(80) NOT NULL
, city VARCHAR(30) NOT NULL
, province CHAR(2) NOT NULL
, postcode CHAR(7) NOT NULL
, interests TEXT NULL
, bio TEXT NULL
, signature TEXT NULL
, skills TEXT NULL
, PRIMARY KEY (user_id)
, FULLTEXT KEY (interests, skills)
) ENGINE=MyISAM;
```

- **limit number of columns per table**
- **split large, infrequently used columns into a separate table**

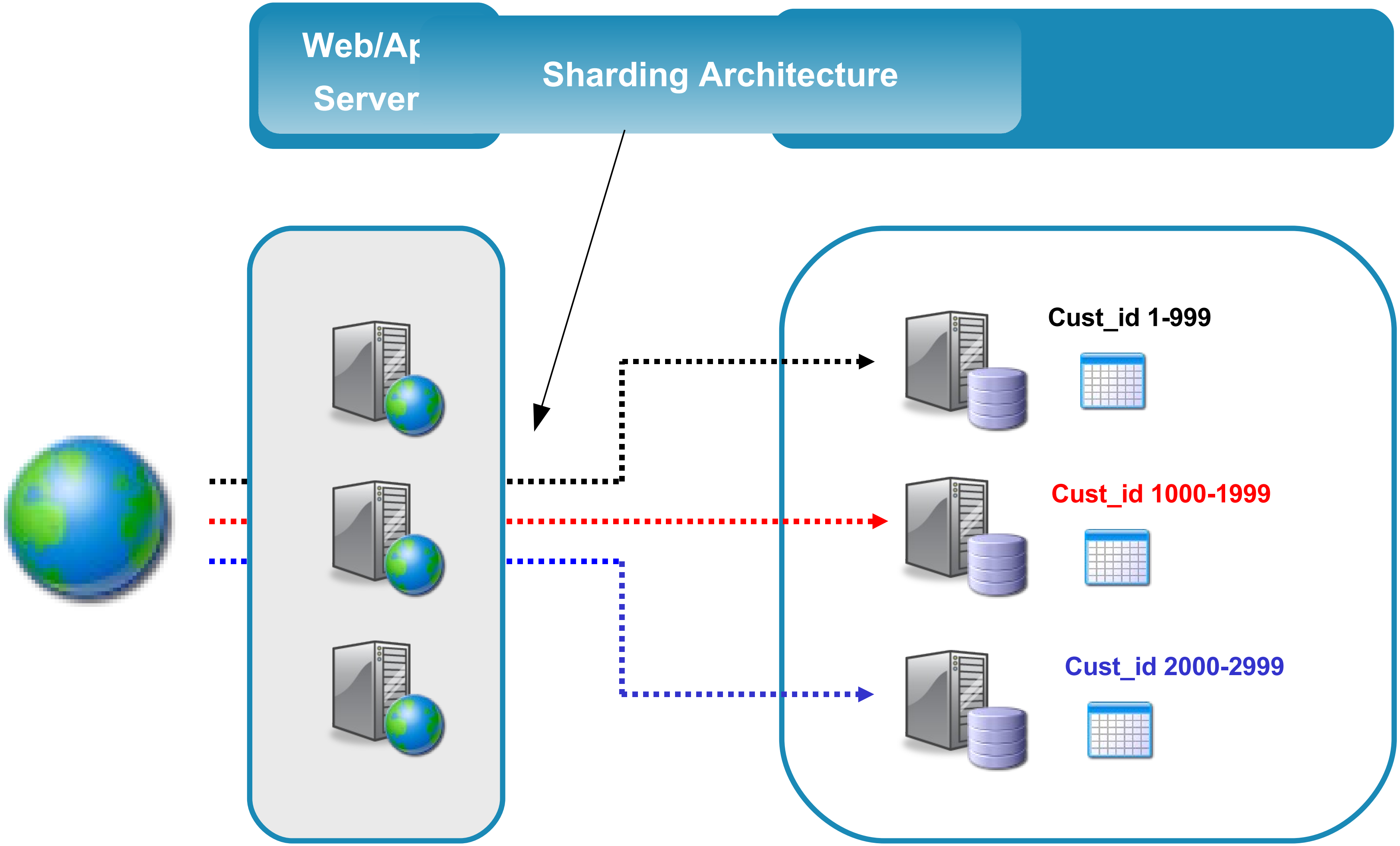
Vertical partitioning

```
@Entity
public class Customer {
   int userid;
   String email;
   String password;
   @OneToOne(fetch=LAZY)
   CustomerInfo info;
}
```

```
@Entity
public class CustomerInfo {
   int userid;
   String name;
   String interests;
   @OneToOne(mappedBy=
      "CustomerInfo")
   Customer customer;
}
```

- split large, infrequently used columns into a separate table

Scalability: Sharding - Application Partitioning



> Sharding =Horizontal partitioning

- Split table by rows into partitions

Know what SQL is executed

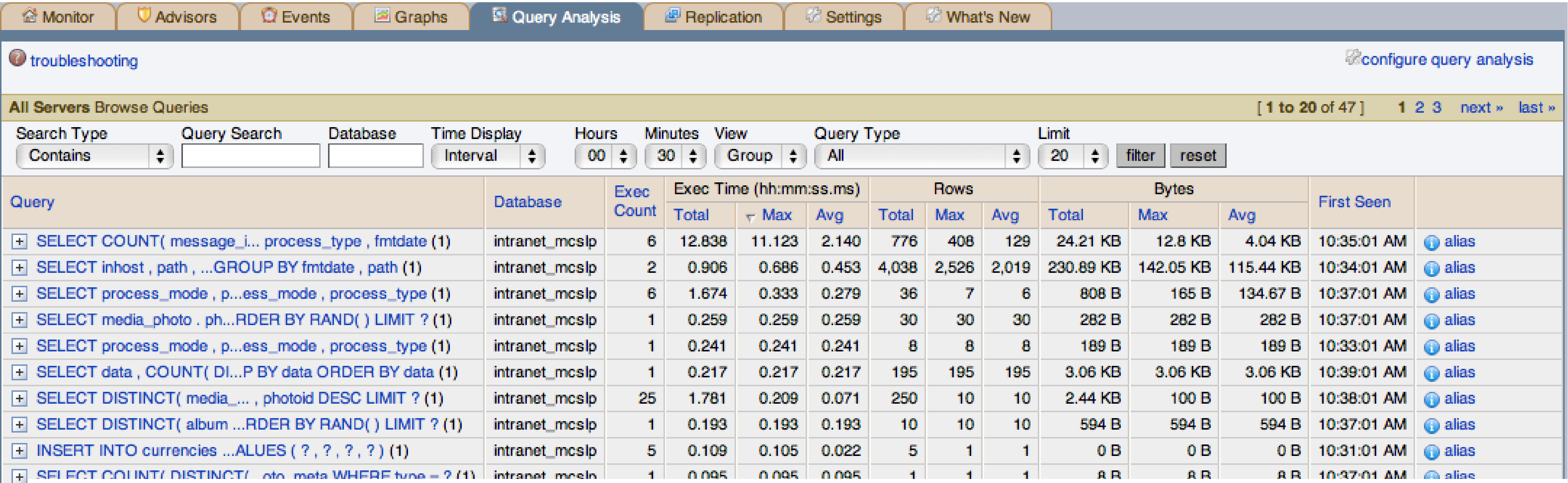
- Capture generated SQL:

  persistence.xml file:

  <property name="toplink.logging.level" value="FINE">

- Find and fix problem SQL:
  - > Watch for **slow** Queries
    - use the MySQL slow query log and use Explain
      - – Can reveal problems such as a **missing Indexes**
  - > Watch for Queries that execute **too often** to load needed data
  - > Watch for loading more data than needed

MySQL Query Analyser



All Servers Browse Queries   [**1 to 20** of 47]   1 2 3   next »   last »

Search Type: Contains   Query Search: Database: Time Display: Interval   Hours: 00   Minutes: 30   View: Group   Query Type: All   Limit: 20   filter   reset

Query	Database	Exec Count	Exec Time (hh:mm:ss.ms)			Rows			Bytes			First Seen	
			Total	Max	Avg	Total	Max	Avg	Total	Max	Avg		
+ SELECT COUNT(message_i... process_type , fmtdate (1)	intranet_mcslp	6	12.838	11.123	2.140	776	408	129	24.21 KB	12.8 KB	4.04 KB	10:35:01 AM	alias
+ SELECT inhost , path , ...GROUP BY fmtdate , path (1)	intranet_mcslp	2	0.906	0.686	0.453	4,038	2,526	2,019	230.89 KB	142.05 KB	115.44 KB	10:34:01 AM	alias
+ SELECT process_mode , p...ess_mode , process_type (1)	intranet_mcslp	6	1.674	0.333	0.279	36	7	6	808 B	165 B	134.67 B	10:37:01 AM	alias
+ SELECT media_photo . ph...RDER BY RAND( ) LIMIT ? (1)	intranet_mcslp	1	0.259	0.259	0.259	30	30	30	282 B	282 B	282 B	10:37:01 AM	alias
+ SELECT process_mode , p...ess_mode , process_type (1)	intranet_mcslp	1	0.241	0.241	0.241	8	8	8	189 B	189 B	189 B	10:33:01 AM	alias
+ SELECT data , COUNT(DI...P BY data ORDER BY data (1)	intranet_mcslp	1	0.217	0.217	0.217	195	195	195	3.06 KB	3.06 KB	3.06 KB	10:39:01 AM	alias
+ SELECT DISTINCT(media_... , photoid DESC LIMIT ? (1)	intranet_mcslp	25	1.781	0.209	0.071	250	10	10	2.44 KB	100 B	100 B	10:38:01 AM	alias
+ SELECT DISTINCT(album ...RDER BY RAND( ) LIMIT ? (1)	intranet_mcslp	1	0.193	0.193	0.193	10	10	10	594 B	594 B	594 B	10:37:01 AM	alias
+ INSERT INTO currencies ...ALUES (? , ? , ? , ?) (1)	intranet_mcslp	5	0.109	0.105	0.022	5	1	1	0 B	0 B	0 B	10:31:01 AM	alias
+ SELECT COUNT(DISTINCT(...oto_meta WHERE type = ? (1)	intranet_mcslp	1	0.095	0.095	0.095	1	1	1	8 B	8 B	8 B	10:37:01 AM	alias

Find and fix problem SQL:

- how long a query took
- results of EXPLAIN statements
- Historical and real-time analysis
  - > **query execution counts**, run time

Its not just slow running queries that are a problem, Sometimes its SQL that **executes a lot** that kills your system

Agenda

- >Entities
- >Entity Manager
- >Persistence Context
- >**Queries**
- >Transaction

Query Types – 1

- Named query
  - > Like `findByXXXX()` from `EntityBeans`
  - > Compiled by persistence engine
  - > Created either with `@NamedQuery` or externalized in `orm.xml`
- Dynamic query
  - > Created dynamically by passing a query string to EntityManager

```
Query query = em.createQuery(“select ...”);
```

  - > Beware of SQL injection, better to use with named parameters

NOT GOOD

```
q = em.createQuery(“select e from Employee e WHERE ”
        + “e.empId LIKE '” + id + “'”);
```

GOOD

```
q = em.createQuery(“select e from Employee e WHERE ”
        + “e.empId LIKE ':id'”);
q.setParameter(“id”, id);
```

Query Types – 2

- Native query
  - Leverage the native database querying facilities
  - Not portable – ties queries to database

Flush Mode

- Controls whether the state of managed entities are synchronized before a query
- Types of flush mode
  - AUTO – immediate, default
  - COMMIT – flush only when a transaction commits
  - NEVER – need to invoke `EntityManger.flush()` to flush

```
//Assume JTA transaction
Order order = em.find(Order.class, orderNumber);
em.persist(order);
Query q = em.createNamedQuery(“findAllOrders”);
q.setParameter(“id”, orderNumber);
q.setFlushMode(FlushModeType.COMMIT);
//Newly added order will NOT visible
List list = q.getResultList();
```

Agenda

- >Entities
- >Entity Manager
- >Persistence Context
- >Queries
- >**Transaction**

Transactions

- Do not perform expensive and unnecessary operations that are not part of a transaction
  > Hurt performance
  > Eg. logging – disk write are expensive, resource contention on log
- Do not use transaction when browsing data
  > `@TransactionAttribute(NOT_SUPPORTED)`

THANK YOU

Carol McDonald

Java Architect

SUN TECH DAYS 2008–2009

A Worldwide Developer Conference