ORACLE®

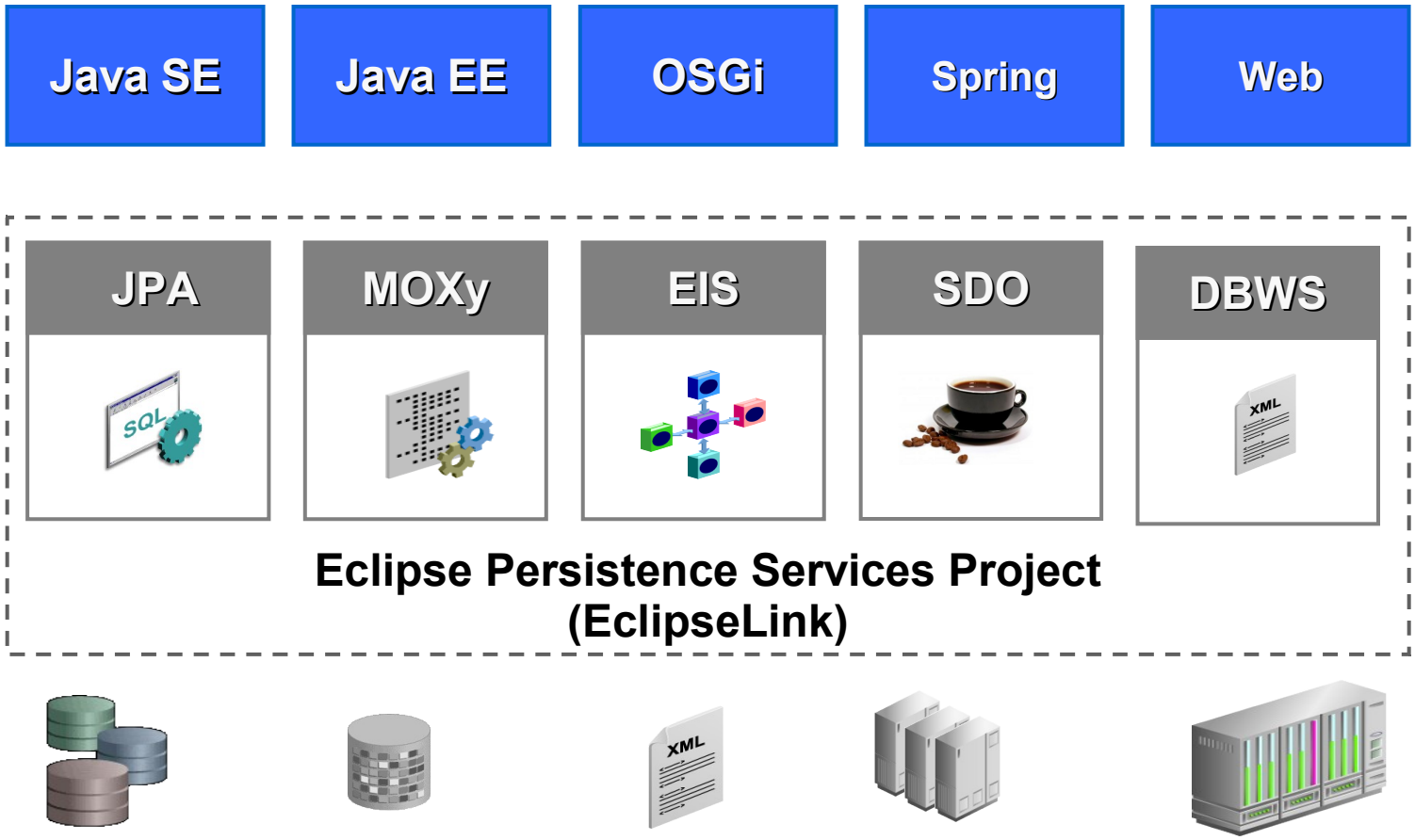
High Performance Persistence with EclipseLink

Shaun Smith—Principal Product Manager
shaun.smith@oracle.com

What is EclipseLink?

- Comprehensive Open Source Persistence solution
  - EclipseLink JPA — Object-Relational (JPA 2.0 RI)
  - EclipseLink MOXy — Object-XML (JAXB)
  - EclipseLink SDO — Service Data Objects (SDO 2.1.1 RI)
  - EclipseLink DBWS — Generated JAX-WS from DB
- Mature and full featured
  - Over 13 years of commercial usage
  - Initiated by the contribution of Oracle TopLink
- Target Platforms
  - Java EE, Web, Spring, Java SE, and OSGi
- Get involved
  - Open collaborative community
  - Contributions welcomed

EclipseLink Project



EclipseLink: Distributions

- Eclipse.org
  - www.eclipse.org/eclipselink/downloads
  - http://download.eclipse.org/rt/eclipselink/updates
- Oracle
  - TopLink 11g
  - WebLogic Server 10.3
- GlassFish v3
  - Replaces TopLink Essentials
  - JPA 2.0 Reference Implementation
- Spring Source
  - Spring Framework and Bundle Repository
- JOnAS
- Jetty
- JEUS TMaxSoft
- SAP NetWeaver coming soon

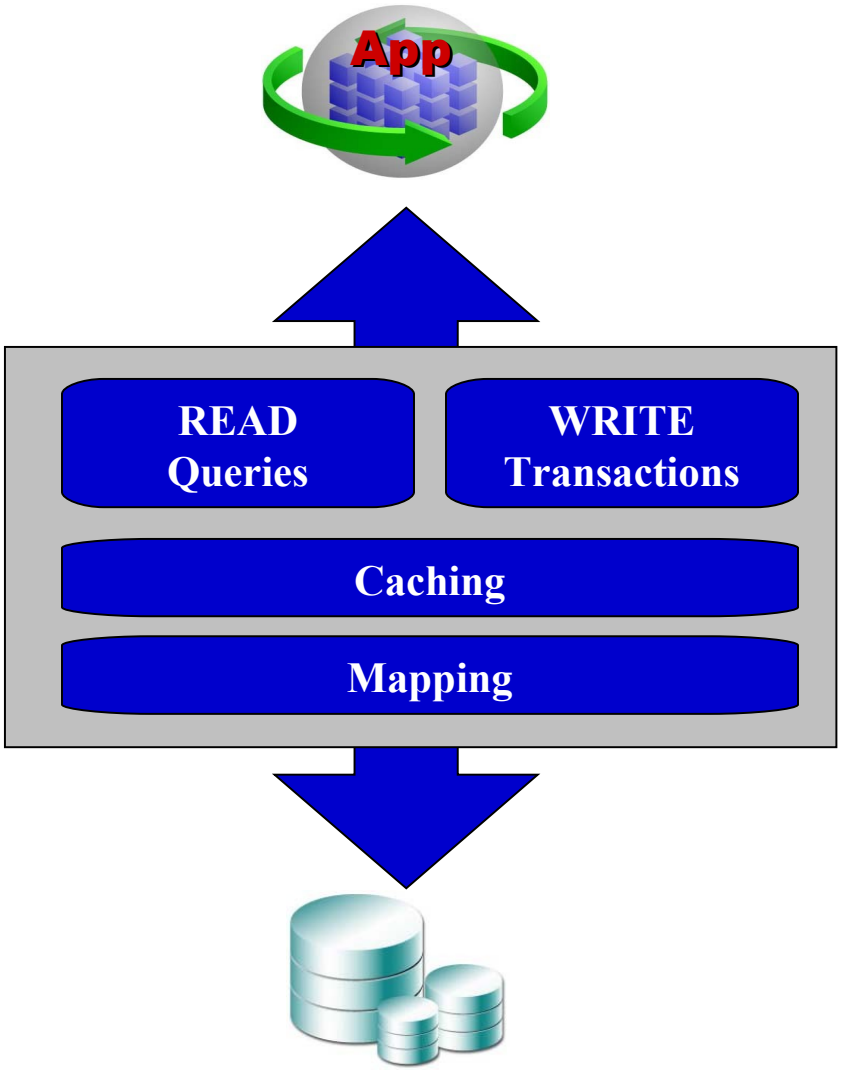
EclipseLink Developer Tool Support

- EclipseLink is a Runtime Project but supported by IDEs
- Eclipse IDE
  - EclipseLink support included by Dali in Eclipse 3.4 (Ganymede)
  - EclipseLink included in Eclipse 3.5 (Galileo) – JavaEE
    - Enhanced Dali support for use of EclipseLink
  - Oracle Enterprise Pack for Eclipse (OEPE)
  - MyEclipse
- JDeveloper 11g
  - JPA, Native ORM, OXM, and EIS mapping
- NetBeans
- Standalone Workbench
  - Native ORM, OXM, EIS

EclipseLink and JPA

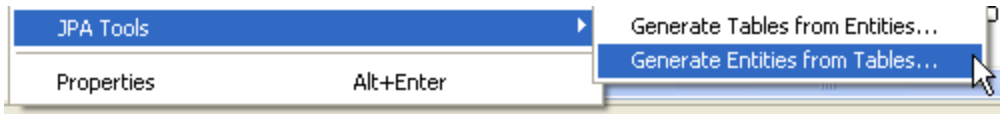
- EclipseLink 1.0 - July 2008
  - JPA 1.0
  - Simple upgrade from TopLink Essentials (JPA 1.0 RI)
- EclipseLink 1.1 - March 2009
  - JPA 1.0 with some JPA 2.0 capabilities
  - EclipseLink 1.1.2 included in Eclipse Galileo
- EclipseLink 2.0 - December 2009
  - JPA 2.0 reference Implementation
  - Maintain support for use as JPA 1.0 implementation
- EclipseLink 2.1 (Helios) – June 23rd 2010
  - Extended JPA 2.0 support

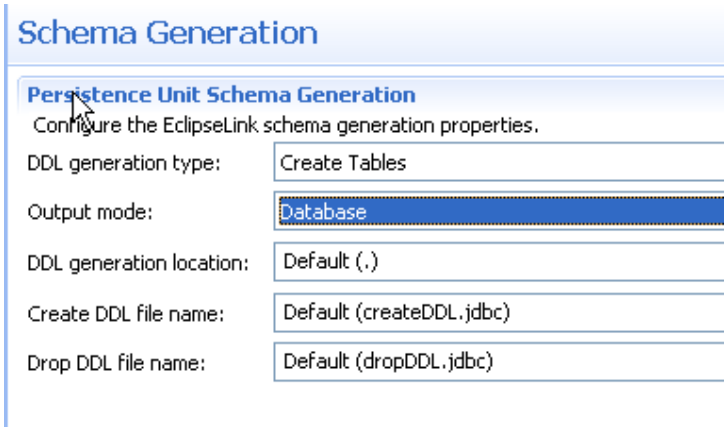
Object-Relational Performance



Modeling and Mapping Generation

- Options:
  - Generation of persistent entities from schema
  - Generation of relational schema from entities
  - Meet in the Middle mappings





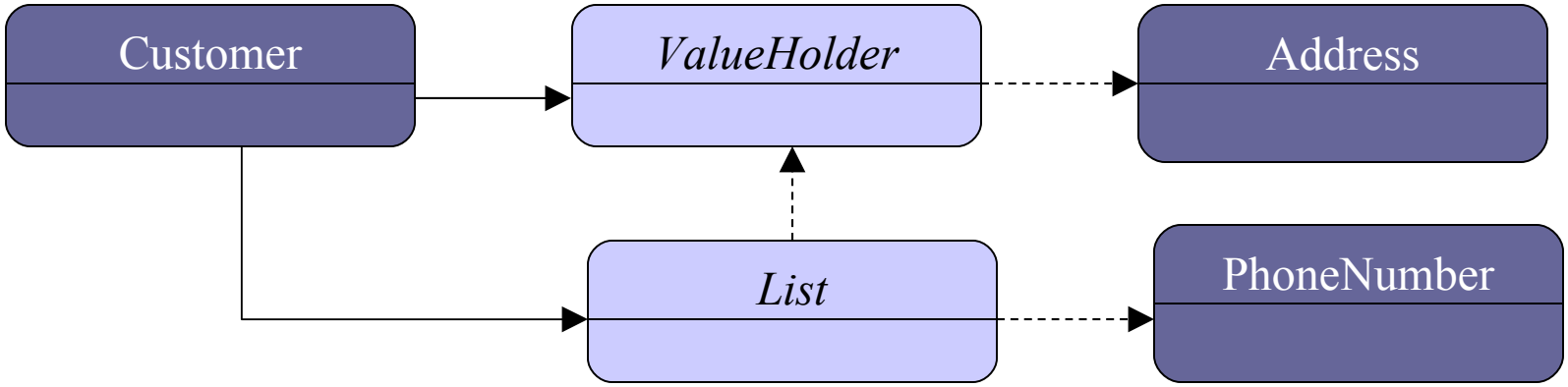
- Excellent productivity boost at start of project
- Not well suited for many legacy schemas
- Avoid being limited by a generated model/schema

Modeling Recommendations

- Leverage model generation to jump-start project
- Match the approach to current and long term project requirements
  - New/Flexible schema versus legacy static schema
  - Future enhancements
  - Maintenance
- Plan for model evolution

Lazy Loading – Just in Time Reading

- Use of proxy to defer reading until required
- Very valuable performance feature
- Several Implementation Options
  - Explicit proxy in domain classes (ValueHolderInterface)
  - Development time class enhancement (source or byte codes)
  - Dynamic weaving (AOP)

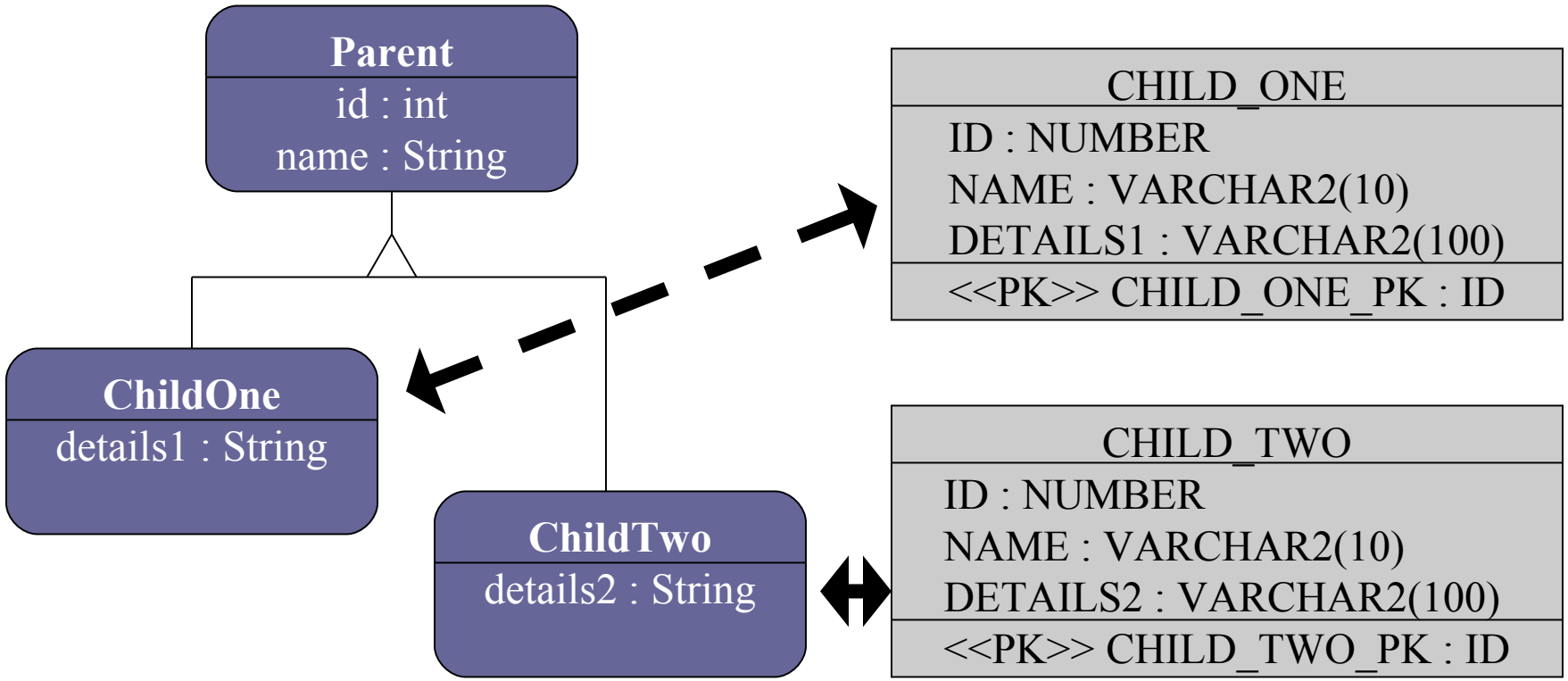


Object-Relational: Inheritance

- Java
  - Used to extend (share) common state & behavior
- Relational
  - Shared data can be normalized into common table
  - Adds additional unique constraint within common table

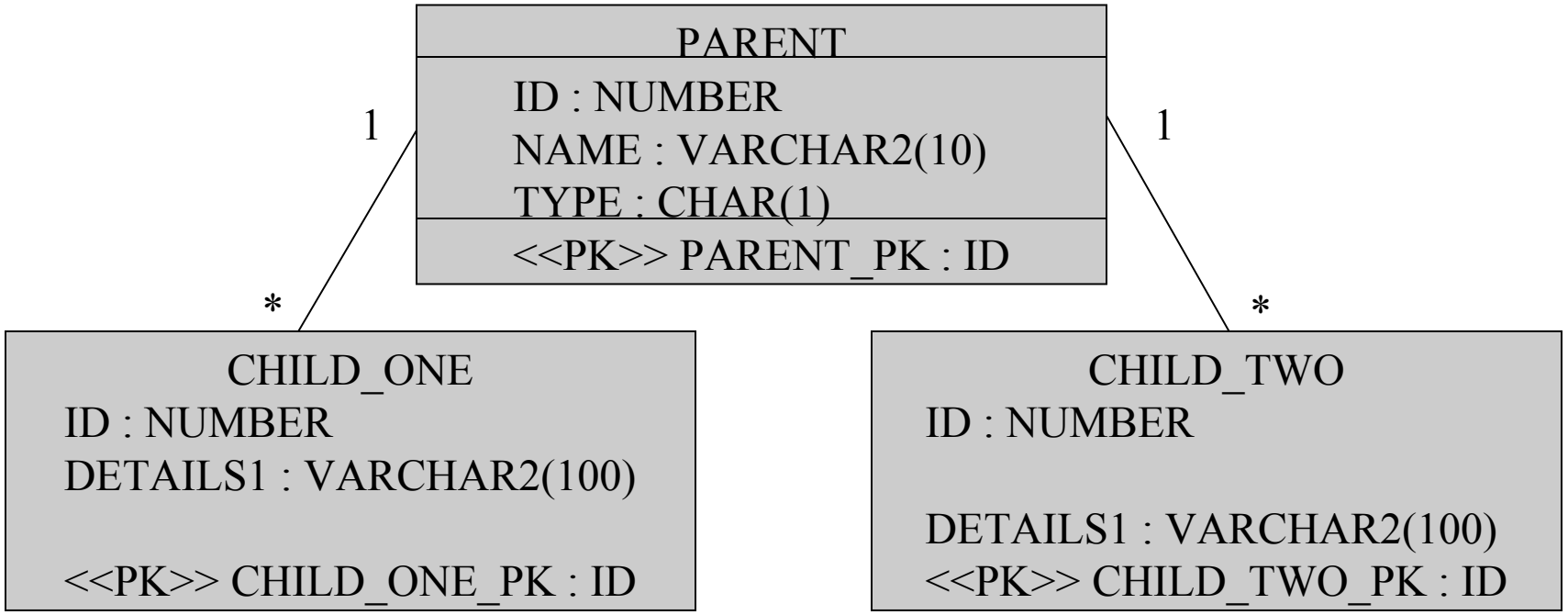
Inheritance: Object

- Only map concrete subclasses
  - PRO: No joins or table contention between types
  - CON: No heterogeneous query results



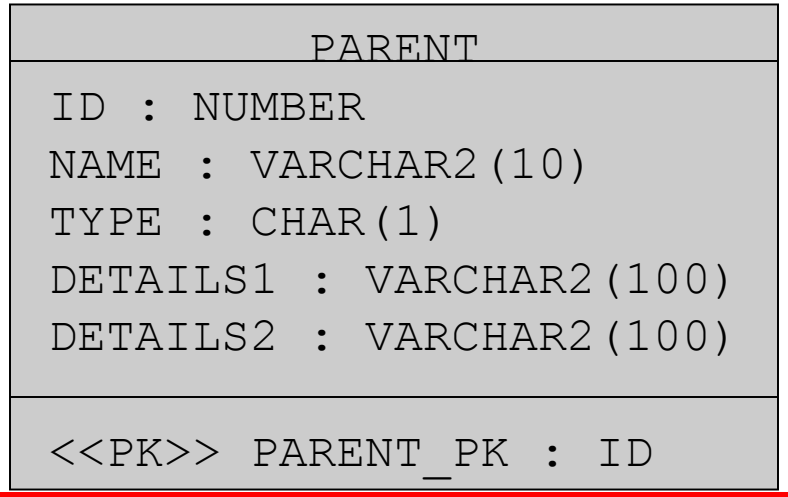
Inheritance: Object and Relational

- Root class has its own table
  - PRO: Heterogeneous query results possible
  - CON: Additional queries or joins required
- Example: Same object model, additional shared table



Inheritance: Optimized

- Root class has its own table + subclass data
  - PRO:
    - Heterogeneous query results possible
    - No extra queries or joins for subclasses
  - CON: Additional table size, unused columns
- Example: Same object mode, one shared table



Additional Mapping Tips

- Flexibility versus Performance
  - Only use Inheritance where required and optimize
    - Extra SQL and joining required
  - Use Table per class inheritance and Variable 1:1 cautiously
- Remove unnecessary mappings
  - Large Collection 1:M and M:M
  - Consider de-normalizing frequently joined tables (embeddable)
  - Consider splitting large tables
- Address sequence and locking columns early in project
- Project Management
  - Allow for change
  - Attack performance goals early to avoid schema/model lockdown

Expressions and JPA 2.0 Criteria

JP QL

```
SELECT e FROM Employee e WHERE e.name LIKE 'D%'
```

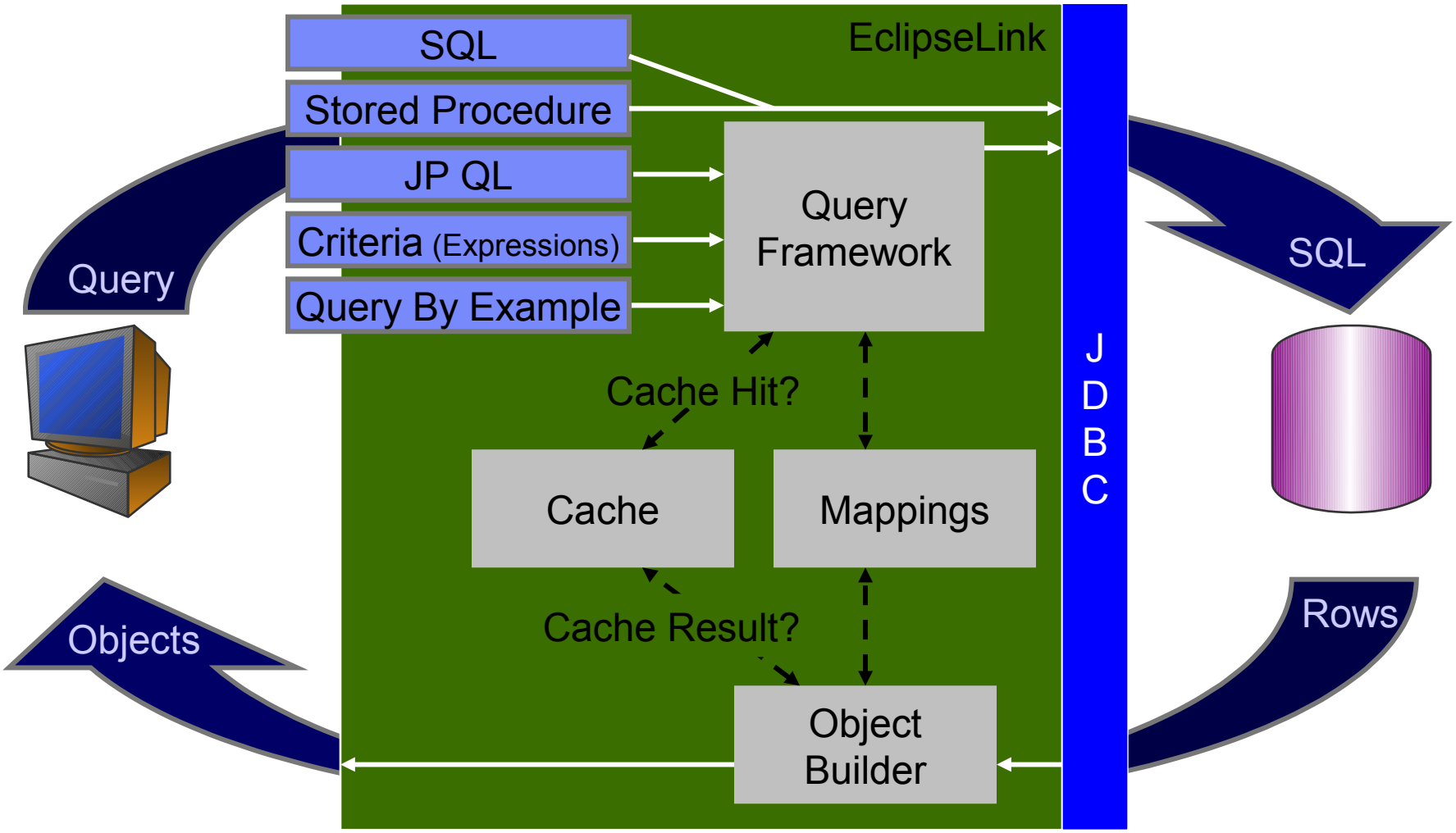
EclipseLink Expressions

```
ExpressionBuilder eb = new ExpressionBuilder();

ReadAllQuery q = new ReadAllQuery(Employee.class, eb);

q.setSelectionCriteria(eb.get("name").like("D%"));

List<Employee> results = JpaHelper.createQuery(em,
   q).getResultList();
```

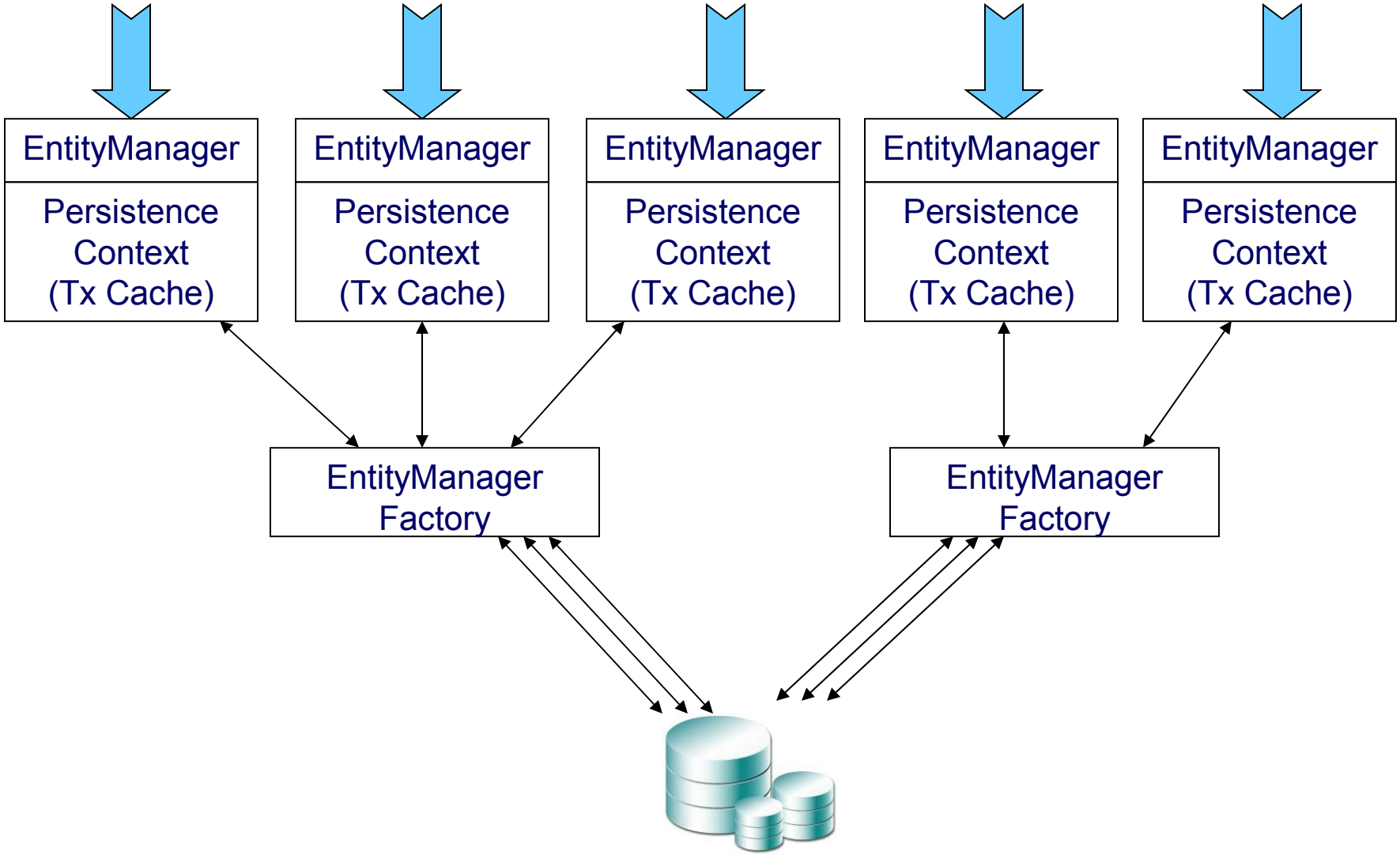
JPA 2.0 Criteria

```
QueryBuilder qb = em.getQueryBuilder();

CriteriaQuery<Employee> cq = qb.createQuery(Employee.class);

Root emp = cq.from(Employee.class);

cq.where(qb.like(emp.get("name"), "D%"));

List<Employee> results = em.createQuery(q).getResultList();
```

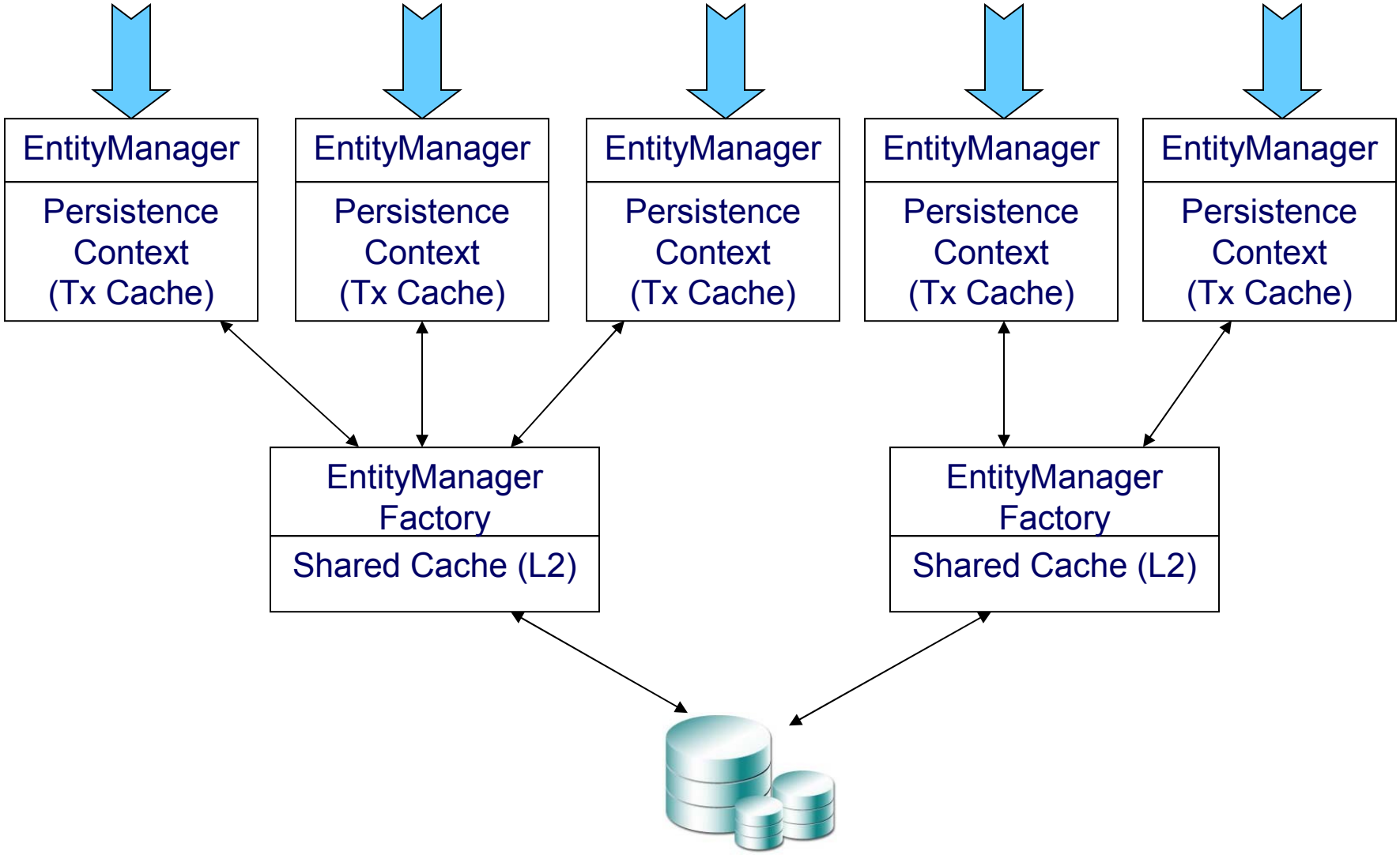
EclipseLink Query Execution



Transactional Caching Only



Shared Caching



EclipseLink's Caching Advantage

- Shared (L2) and Persistence Context caching
  - Entity cache—not data cache
- Cache Invalidation/Expiration
  - Time to live
  - Fixed Times
  - Programmable (external notification)
- Cache Coordination
  - Messaging
    - JMS, RMI, IIOP, CORBA
  - Type specific configuration
    - Modes: Invalidate, Sync, Sync+New, None
- All configurable on a **per Entity basis**

EclipseLink Cache Configuration

- Cache Shared (L2)/Isolated (L1 only)
  - Entity cache—not data cache

```
<property
  name="eclipselink.cache.shared.default"
  value="true"/>
```

- Cache Type & Size
  - Soft/Hard Weak
  - Weak
  - Full

```
@Cache(type = CacheType.HARD_WEAK,
       size = 500,
       isolated = true,
       coordinationType =
       INVALIDATE_CHANGED_OBJECTS,

expiryTimeOfDay=@TimeOfDay(hour=1)))
```

```
<property
       name="eclipselink.cache.type.default"
  value="Full"/>
```

JPA 2.0 Caching

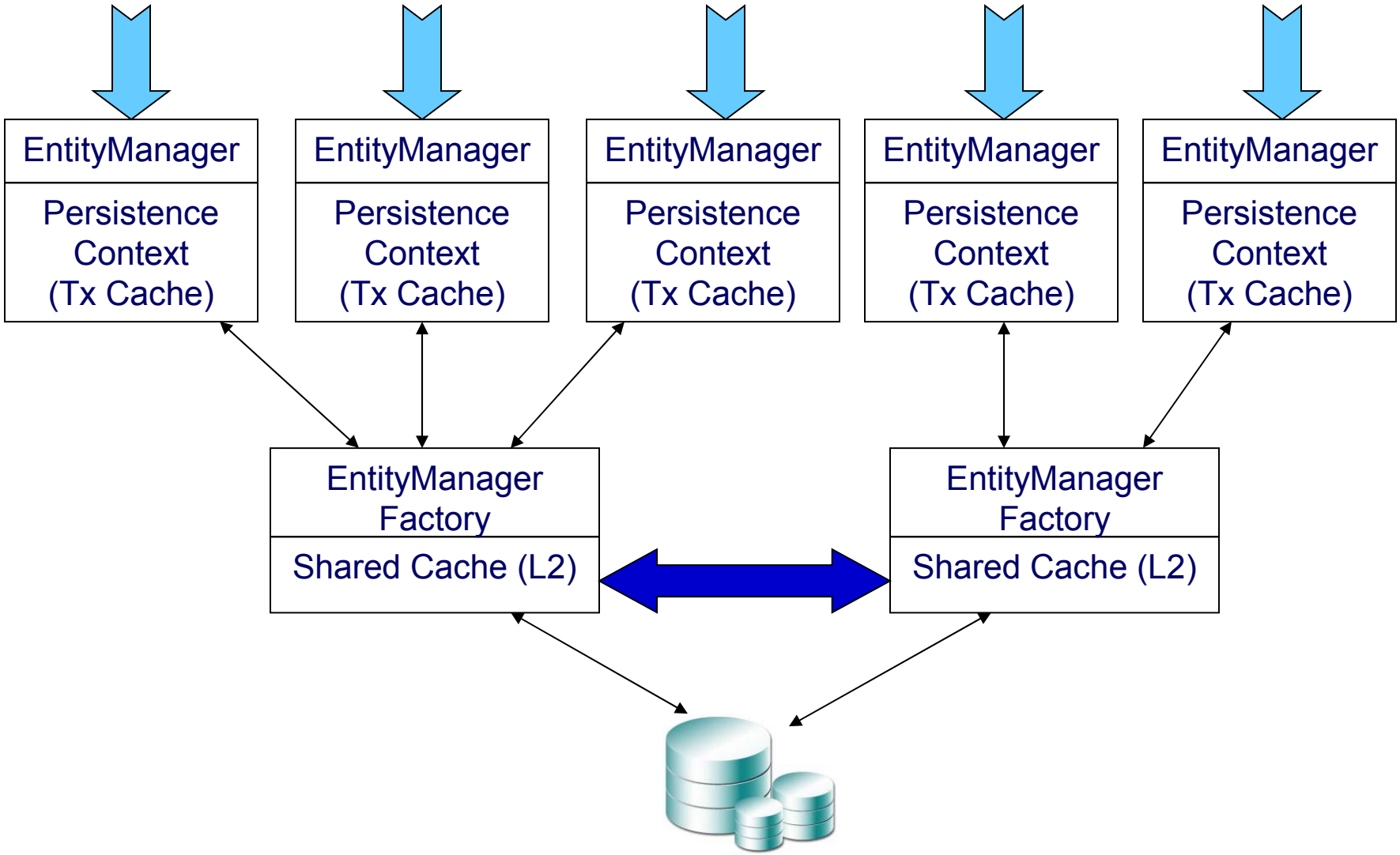
- API for operating on (shared) entity cache.
- Accessible from EntityManagerFactory
- Supports only very basic cache operations
- Can be extended by vendors

```
public interface Cache {
   public boolean contains(Class cls, Object pk);
   public void evict(Class cls, Object pk);
   public void evict(Class cls);
   public void evictAll();
}
```

Annotation **@Cacheable** (default true)

- Property CacheSetting to denote per PersistenceUnit whether to cache

Shared Coordinated Caching



Cache Configuration Best Practices

- Understand your application
  - Read-Only — FULL?
  - Read-Mostly — SOFT-WEAK
  - Volatile – frequent updates — WEAK or ISOLATED (PC)
  - Shared between users/requests — SHARED or ISOLATED?
  - Clustered Deployment — COORDINATION?
- Leverage the caching but protect the data
  - Don’t forget the Locking
- Features of Interest
  - Read-only query results
  - In-memory Querying (Cache Usage options)
  - Refreshing: query, expired, always

Locking – Concurrency Protection

- CRITICAL to avoid DB corruption in concurrent applications
- Java Developers want to think of locking at the object level
- Databases may need to manage locking across many applications
- EclipseLink is able to respect and participate in locks at database level
  - @OptimisticLocking : Numeric, Timestamp, All fields, Selected fields, Changed field

```
@Entity
@OptimisticLocking(type=CHANGED_COLUMNS)
public class Employee {
```

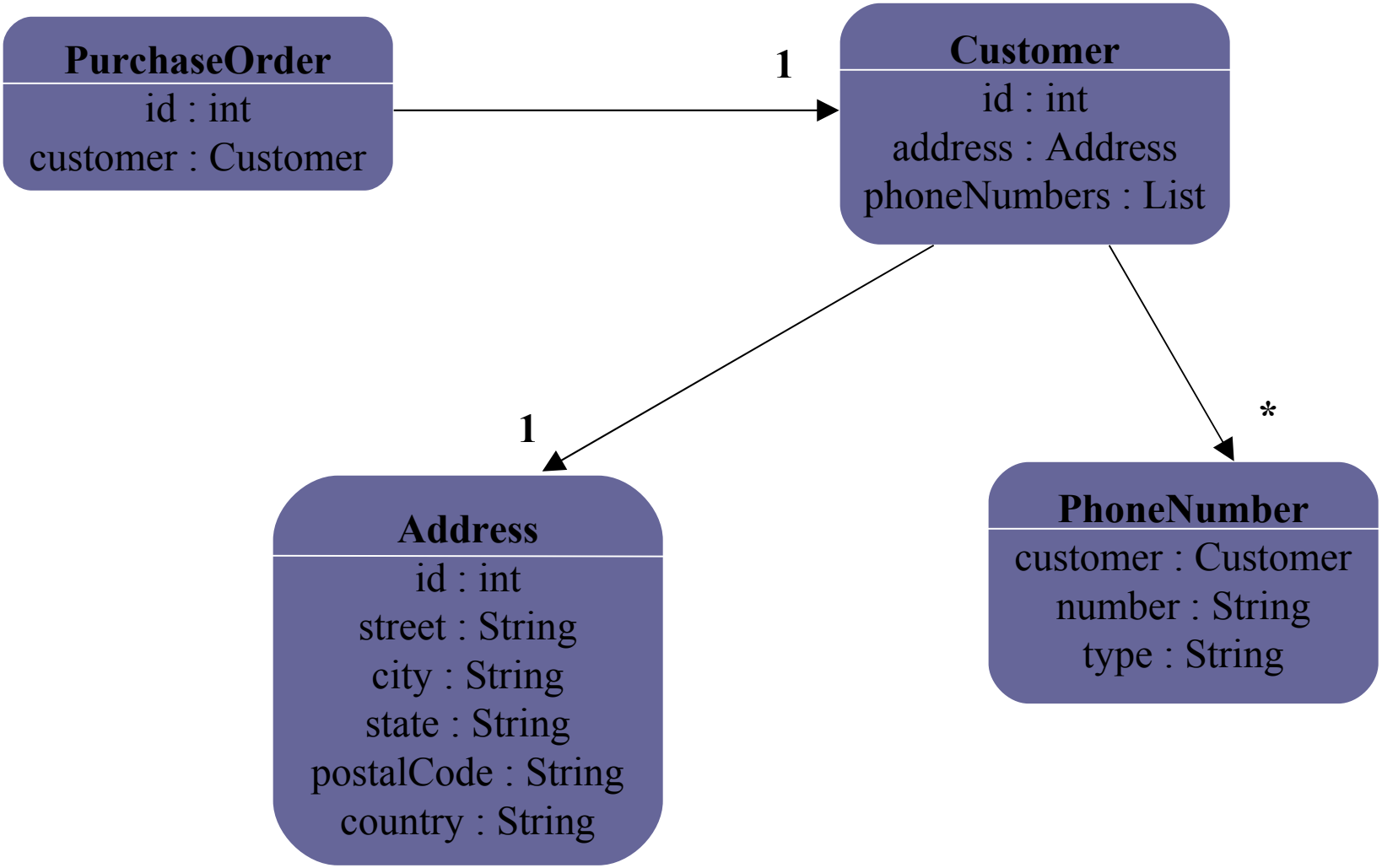
  - Pessimistic query hints: *eclipselink.pessimistic-lock*

    SELECT … FOR UPDATE

JPA 2.0: Advanced Locking

- New LockMode values introduced:
  - OPTIMISTIC (READ)
  - OPTIMISTIC_FORCE_INCREMENT (WRITE)
  - PESSIMISTIC_READ
  - PESSIMISTIC_WRITE
  - PESSIMISTIC_FORCE_INCREMENT
- Optimistic locking cooperates with pessimistic locking
- Multiple places to specify lock (depends upon need)
  - Query
  - EntityManager

Challenge: Graph Loading



Graph Load: Example

- Find all pending PO with a customer in San Francisco
- Populate a result page including information from the PO, Customer, Address, & Phone

```
Query query =
    em.createQuery("SELECT po FROM PurchaseOrder po " +
                   "WHERE po.status = 'ACTIVE' AND " +
                   "po.customer.address.city = 'SFO'");
List<PurchaseOrder> pos = query.getResultList();

// Force graph load simulating presentation tier logic
for (PurchaseOrder po: pos) {
    po.getCustomer().getAddress();
    po.getCustomer().getPhone().size();
}
```

Graph Load: No Optimizations

```
SELECT PO.* FROM PO, CUST, ADDR WHERE
PO.STATUS = 'ACTIVE' AND PO.CUST_ID = CUST.ID
AND CUST.ADDR_ID = ADDR.ID AND ADDR.CITY =
'SFO'

   {Returns N Purchase Orders}

SELECT * FROM CUST WHERE CUST.ID = 1 … {N}

SELECT * FROM ADDR WHERE ADDR.ID = 100 … {N}

SELECT * FROM PHONE WHERE PHONE.CUST_ID = 1 …
{N}
```

RESULT: 3N+1 queries (100 PO's = 301)

"N+1 Query – explosion of SQL"

Join Reading FK Relationships

- Join Reading of M:1 and 1:1

```
Query query = em.createQuery("…");
query.setHint(QueryHints.FETCH, "po.customer");
query.setHint(QueryHints.FETCH, "po.customer.address");
List<PurchaseOrder> pos = query.getResultList();
```

- Resulting SQL:

```
SELECT PO.*,CUST.*,ADDR.* FROM PO, CUST, ADDR WHERE
   PO.STATUS = 'ACTIVE' AND PO.CUST_ID = CUST.ID AND
   CUST.ADDR_ID = ADDR.ID AND ADDR.CITY = 'SFO'

   {Returns N Purchase Orders with Customers & Addresses}

SELECT * FROM PHONE WHERE PHONE.CUST_ID = 1

   …{N calls}
```

RESULT: N+1 queries (100 PO's = 101 SQL)

Join Reading All Relationships

- Join Reading

```
Query query = em.createQuery("SELECT po FROM PurchaseOrder po …");
query.setHint(QueryHints.FETCH, "po.customer");
query.setHint(QueryHints.FETCH, "po.customer.address");
query.setHint(QueryHints.FETCH, "po.customer.phones");
List<PurchaseOrder> pos = query.getResultList();
```

- Resulting SQL:

```
SELECT PO.*,CUST.*,ADDR.*, PHONE.*
FROM PO, CUST, ADDR, PHONE
WHERE PO.STATUS = 'ACTIVE' AND PO.CUST_ID = CUST.ID AND
   CUST.ADDR_ID = ADDR.ID AND ADDR.CITY = 'SFO' AND
   PHONE.CUST_ID = CUST.ID
```

{Returns N Purchase Orders with Customers, Addresses, and PhoneNumbers}

RESULT: 1 query (100 PO's = 1 SQL)

Batch and Join Reading

- Use SELECT per child with join to original clause

```
Query query = em.createQuery("SELECT po FROM PurchaseOrder po …");
query.setHint(QueryHints.FETCH, "po.customer");
query.setHint(QueryHints.FETCH, "po.customer.address");
query.setHint(QueryHints.BATCH, "po.customer.phones");
List<PurchaseOrder> pos = query.getResultList();
```

- Resulting SQL:

```
SELECT PO.*,CUST.*,ADDR.* FROM PO, CUST, ADDR WHERE
  PO.STATUS = 'ACTIVE' AND PO.CUST_ID = CUST.ID AND
  CUST.ADDR_ID = ADDR.ID AND ADDR.CITY = 'SFO'
```

{Returns N Purchase Orders with Customers & Addresses}

```
SELECT PHONE.* FROM PHONE, PO, CUST, ADDR WHERE
  PO.STATUS = 'ACTIVE' AND PO.CUST_ID = CUST.ID AND
  CUST.ADDR_ID = ADDR.ID AND ADDR.CITY = 'SFO' AND
  PHONE.CUST_ID = CUST.ID
```

RESULT: 2 queries (100 PO's = 2 SQL)

Batch All Reading

- Use SELECT per child with join to original clause

```
Query query = em.createQuery("SELECT po FROM PurchaseOrder po …");
query.setHint(QueryHints.BATCH, "po.customer");
query.setHint(QueryHints.BATCH, "po.customer.address");
query.setHint(QueryHints.BATCH, "po.customer.phones");
List<PurchaseOrder> pos = query.getResultList();
```

- Resulting SQL

```
SELECT PO.* FROM PO, CUST, ADDR WHERE PO.STATUS = 'ACTIVE' AND
   PO.CUST_ID = CUST.ID AND CUST.ADDR_ID = ADDR.ID AND ADDR.CITY
   = 'SFO'

SELECT CUST.* FROM PO, CUST, ADDR WHERE PO.STATUS = 'ACTIVE' AND
   PO.CUST_ID = CUST.ID AND CUST.ADDR_ID = ADDR.ID AND ADDR.CITY
   = 'SFO'

SELECT ADDR.* FROM PO, CUST, ADDR WHERE PO.STATUS = 'ACTIVE' AND
   PO.CUST_ID = CUST.ID AND CUST.ADDR_ID = ADDR.ID AND ADDR.CITY
   = 'SFO'

SELECT PHONE.* FROM PHONE, PO, CUST, ADDR WHERE PO.STATUS =
   'ACTIVE' AND PO.CUST_ID = CUST.ID AND CUST.ADDR_ID = ADDR.ID
   AND ADDR.CITY = 'SFO' AND PHONE.CUST_ID = CUST.ID
```

RESULT: 4 queries (100 PO's = 4 SQL)

Graph Loading Summary

	# SQL	Time
Default	301	200 ms
Join 1:1's	101	150 ms
Join All	1	60 ms
Batch All	4	40 ms
Batch & Join	2	20 ms

Other Query Optimizations

- Read-Only Requests: Leverage the shared cache

```
query.setHint(QueryHints.READ_ONLY, TRUE);
```

  - Returns instance from shared cache (no build overhead)

- Data Search
  - Query scenarios should only return what is needed
  - Retrieve entities (into cache) when needed

```
SELECT e.name, e.address.city FROM Employee e
SELECT new EmpView(e.name, e.salary) FROM Employee e
```

  - Consider Pagination

```
query.setFirstResult(0);
query.setMaxResults(5);
```

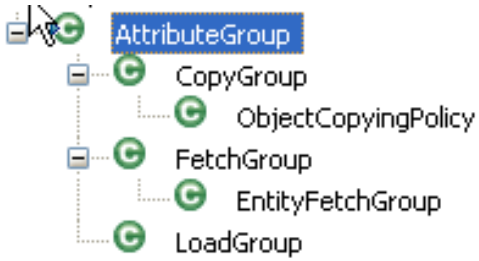
- Partial entities

FetchGroup in EclipseLink 2.0

- Supports lazy loading of relationship and @Basic mappings
  - Defines which columns to load
  - Fetches additional columns when needed
- Default Fetch Groups defined automatically: eagerly loaded fields and lazily loaded fields.
  - Initial Entity select fetches only eager fields.
  - Referencing *any* lazy field will fetch all lazy fields
  - Especially useful with CLOB/BLOB fields to avoid loading until necessary
- Enabled by byte code weaving
- Fetch Groups can be manually defined and added to a query as a hint—overrides default fetch groups.

AttributeGroup in EclipseLink 2.1

- Common group for defining attribute graph fetch, load, and copy
  - FETCH: Which columns are selected
  - LOAD: Relationships populated
  - COPY: Attributes copied
  - MERGE: Sparse merge




```
TypedQuery<Employee> query = em.createQuery("SELECT e FROM Employee e WHERE

// Load only its names and phone Numbers
FetchGroup group = new FetchGroup();
group.addAttribute("firstName");
group.addAttribute("lastName");
group.addAttribute("phoneNumbers");

// Use the FetchGroup for loading as well
group.setShouldLoadAll(true);

// Controls what is queried from the database initially
query.setHint(QueryHints.FETCH_GROUP, group);

Employee emp = query.getSingleResult();
```

AttributeGroup: Copy & Merge

- Create partial entity graph
- Merge only attributes defined in group
  - Still follows cascade-merge configuration on relationships

```
// Copy only its names and phone Numbers
AttributeGroup group = new CopyGroup();
group.addAttribute("firstName");
group.addAttribute("lastName");
group.addAttribute("address");

Employee empCopy = (Employee) em.unwrap(JpaEntityManager.class).copy(emp, group);

empCopy.setFirstName(emp.getLastName());
empCopy.setLastName(emp.getFirstName());

// Merge the detached employee copy
em.merge(empCopy);
```

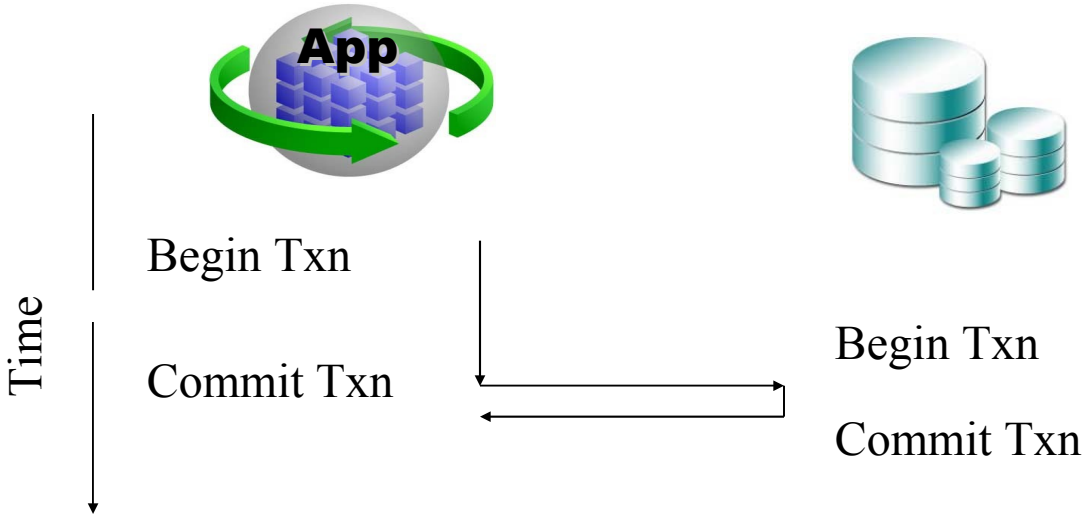
AttributeGroup: Serialize & Merge

- Only serialize what was fetched & loaded
- Allow entity to be extended

```
AttributeGroup group = new AttributeGroup();
group.addAttribute("firstName");
group.addAttribute("lastName");
group.addAttribute("phoneNumbers");

// Controls what is queried from the database initially
query.setHint(QueryHints.FETCH_GROUP, group);
query.setHint(QueryHints.LOAD_GROUP, group);

Employee emp = query.getSingleResult();

// Detach Employee through Serialization
Employee detachedEmp = (Employee) serialize(emp);

// Modify the detached Employee inverting the names, adding a phone number, and setting the salary
detachedEmp.setFirstName(emp.getLastName());
detachedEmp.setLastName(emp.getFirstName());
detachedEmp.addPhoneNumber("TEST", "999", "999999");
detachedEmp.setSalary(1);

// Merge the detached employee
em.merge(detachedEmp);
```

Transactions: Short & Sweet

- J2EE apps typically support many clients sharing small number of db connections
- Ideally would like to minimize length of transaction on database



Change Tracking

- Use the @ChangeTracking annotation to configure the way changes to Entities are computed. Four options are provided:
  - ATTRIBUTE – uses weaving to detect which fields or properties of the object change.
  - OBJECT – uses weaving to detect if the object has been changed, but uses a backup copy of the object to determine what fields or properties changed.
  - DEFERRED – defers all change detection to the underlying UnitOfWork's change detection process.
  - AUTO – If weaving is available then ATTRIBUTE, if weaving not available then DEFERRED
    - Default
- Optimize Transactions

EclipseLink Weaving

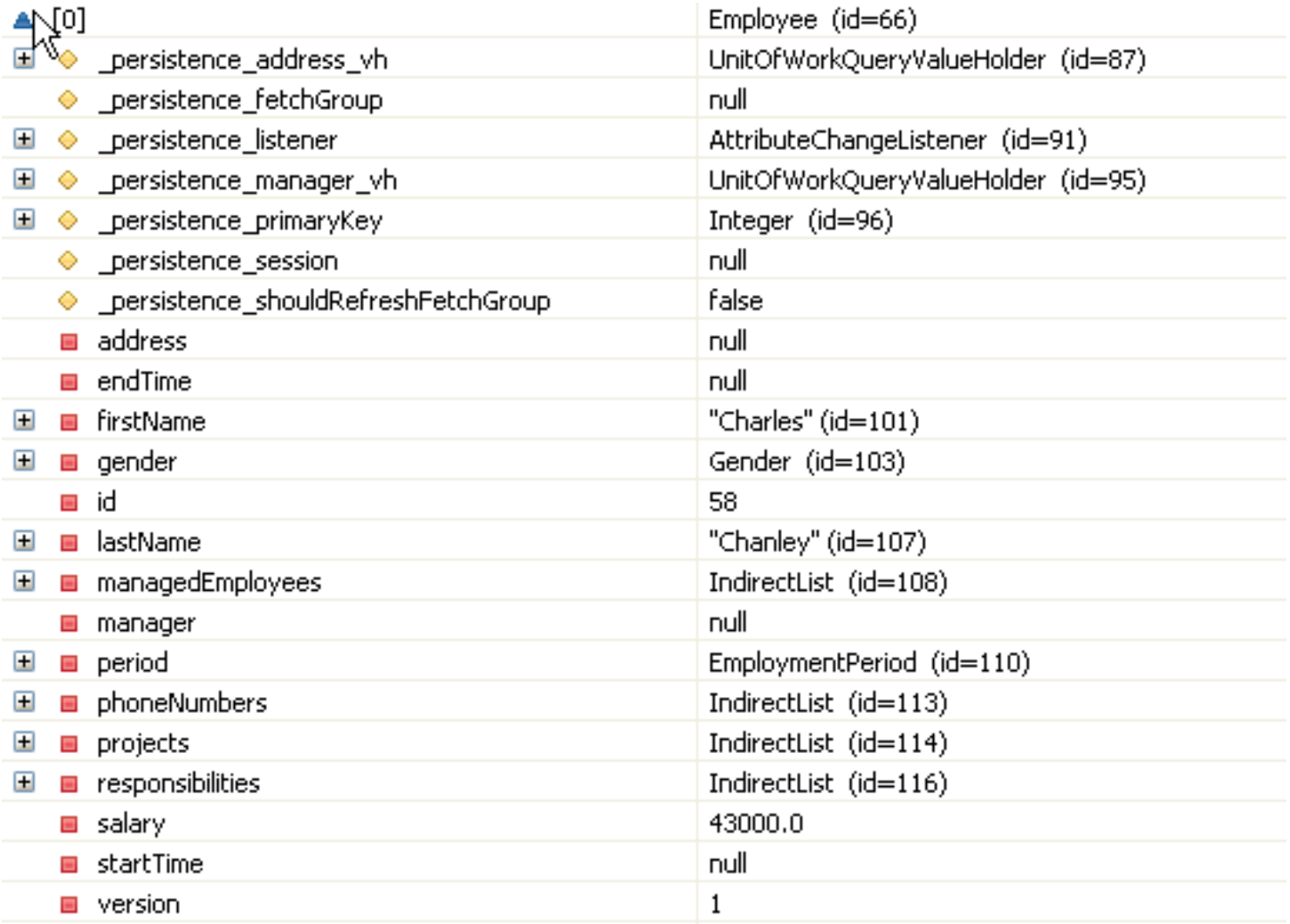
Usage is optional and configurable

<property name="eclipselink.weaving" value="true" />

- Default (**true**): Weave when possible and when needed
- **false**: Do not weave
- **static**: assumes static weaving has been performed

- Value-add
  - Lazy loading of M:1 and 1:1 relationships
  - State caching (identity, context)
  - Lazy loading of Basic attributes (Fetch Groups)
    - Configurable. Property: "eclipselink.weaving.fetchgroups"
  - Change Tracking
    - Configurable. Property: "eclipselink.weaving.changetracking"

What does Weaving Look Like?



ORM Performance in a Nutshell

- Performance must be planned for
  - Modeling and mapping decisions can effect/limit performance
    - Address measuring performance early and allow for change
    - Work with your DBA – EclipseLink mappings and queries are very flexible
- Learn what's in your toolbox
  - Caching
  - Locking
  - Queries
  - Transactions
  - Profilers (EclipseLink provided and external)
- Know you application