

# Object-Relational Mapping Tools

... let's talk to each other!



BASEL ■ BERN ■ BRUGG ■ DÜSSELDORF ■ FRANKFURT A.M. ■ FREIBURG I.BR. ■ GENEVA  
HAMBURG ■ COPENHAGEN ■ LAUSANNE ■ MUNICH ■ STUTTGART ■ VIENNA ■ ZURICH

**trivadis**  
makes IT easier. ■ ■ ■

# ■ Agenda

- O/R Mappers – what, why, how
- The “Object-Relational Impedance Mismatch”
- Fetching Data

# ■ Love - Hate: What People Say About O/R Mapping

“The Vietnam of Computer Science”

<http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx>

“ORM hate”

<http://java.dzone.com/articles/martin-fowler-orm-hate>

“No more need for ORMs”

<http://blog.jooq.org/2014/04/11/java-8-friday-no-more-need-for-orms/>

“ORM haters don’t get it”

<http://techblog.bozho.net/orm-haters-dont-get-it/>

# ■ What is an O/R Mapper?

- “Translation service” between data structures in application code (objects, in OOP) and tuples in a relational database
- Typically part of a persistence framework that offers additional functionality like lifecycle management, transaction handling, caching, connection pooling, validation, etc.
- At its simplest, an O/R Mapper might map a database table directly to an application object (Active Record)

# ■ Why Should You Care?

- Developer: Because you want good performance
- Database administrator: Because you want good performance
  - With or without an ORM, tuning application SQL is not just “SQL Tuning”
  - Application processing logic decides *what* is retrieved from the database, and *when*
  - As a DBA, you would normally just catch a glimpse of this logic, e.g. by tracing
  - Applications using documented ORMs may even be more accessible to external diagnosis and consulting (as opposed to in-house frameworks)

# ■ Scope and Purpose

- Focus on
  - Essential challenges of O/R mapping
  - Fetch / SELECT performance considerations
- Help understanding of what ORMs do, as a prerequisite to achieving optimal performance
- Using Java and Hibernate as an *example* of an O/R mapping framework

# ■ Do It Yourself – Plain JDBC

```
String getEmployeeByCity = "select employee_id, firstname, lastname from employee";
String getTaskByEmployeeId = "select name, description, status from task where employee_id = ?";
PreparedStatement employeeStmt = conn.prepareStatement(getEmployeeByCity);
PreparedStatement taskStmt = conn.prepareStatement(getTaskByEmployeeId);

List<Employee> employees = new ArrayList<>();
ResultSet rset1 = employeeStmt.executeQuery();

while (rset1.next()) {
    Employee employee = new Employee();
    employee.setFirstname(rset1.getString(2));
    employee.setLastname(rset1.getString(3));
    List<Task> tasks = new ArrayList<>();
    taskStmt.setInt(1, rset1.getInt(1));
    ResultSet rset2 = taskStmt.executeQuery();

    while (rset2.next()) {
        Task task = new Task();
        task.setName(rset2.getString(1));
        task.setDescription(rset2.getString(2));
        task.setStatus(rset2.getString(3));
        tasks.add(task);
    }
    rset2.close();
    employee.setTasks(tasks);
    employees.add(employee);
}
rset1.close();
taskStmt.close();
employeeStmt.close();
taskStmt.close();
```

# ■ ORM Example - Hibernate

- The same with Hibernate, using JPQL

```
List<Employee> employees = em.createQuery("select e from Employee e join fetch e.tasks").  
    getResultList();
```

- The same with Hibernate, using the Criteria API

```
final CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<Employee> criteria = cb.createQuery(Employee.class);  
Root<Employee> from = criteria.from(Employee.class);  
from.fetch("tasks");  
List<Employee> employees = em.createQuery(criteria).getResultList();
```

- If we were searching for a specific employee (and possibly her tasks)

```
Employee employee = em.find(Employee.class, employeeId);
```



# ■ ORM Example: Some Basic Mappings

- @Table (name = <...>)
- @OneToMany (<...>)
- @ManyToOne (<...>)

```
@Entity
@Table(name = "EMPLOYEE")
public class Employee implements Serializable {

    @Id
    @Basic(optional = false)
    @Column(name = "EMPLOYEE_ID")
    private BigDecimal employeeId;

    @OneToMany(mappedBy = "employee", fetch = FetchType.LAZY)
    private Collection<Task> tasks;
```

```
@Entity
@Table(name = "TASK")
public class Task implements Serializable {

    @Id
    @Basic(optional = false)
    @Column(name = "TASK_ID")
    private BigDecimal taskId;

    @JoinColumn(name = "EMPLOYEE_ID", referencedColumnName = "EMPLOYEE_ID", nullable=false)
    @ManyToOne(fetch = FetchType.LAZY)
    private Employee employee;
```

# ■ Agenda

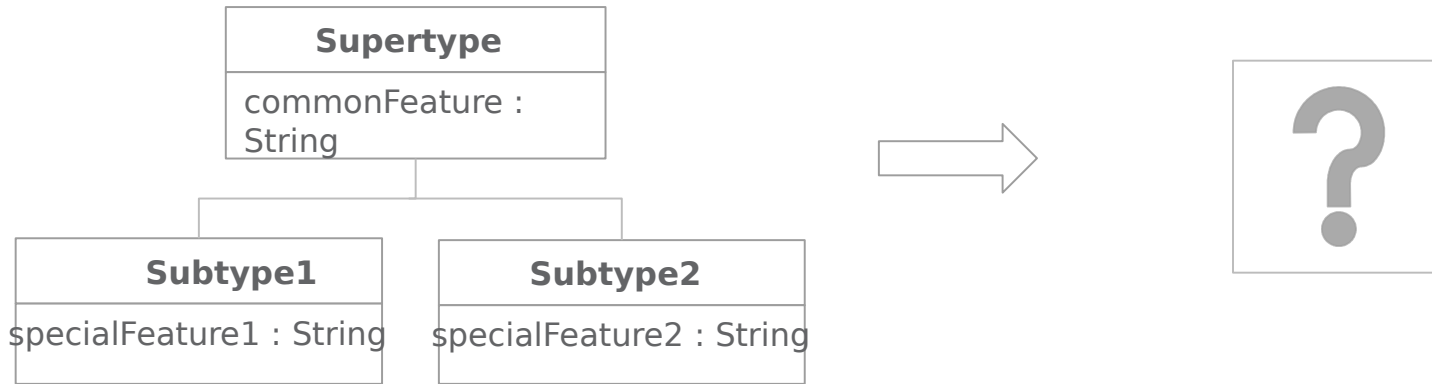
- O/R Mappers – what, why, how
- The “Object-Relational Impedance Mismatch”
- Fetching Data

# ■ The “Object-Relational Impedance Mismatch”

- In all but the simplest applications, 1 table <-> 1 class mappings don't necessarily fit all cases
- More importantly, the mechanics of data retrieval are fundamentally different in OOP vs. relational databases / SQL
- Conceptual / theoretical mismatch may easily transform into real world performance issues

# ■ The “Object-Relational Impedance Mismatch”

## Inheritance (IS-A relationships)



# ■ Mapping Inheritance

## ■ Strategy No. 1: table per concrete class

KEY.SUBTYPE1	
P * ID	NUMBER
COMMON_FEATURE	VARCHAR2 (30 BYTE)
SPECIAL_FEATURE1	VARCHAR2 (30 BYTE)
👉 SUBTYPE1_PK (ID)	

KEY.SUBTYPE2	
P * ID	NUMBER
COMMON_FEATURE	VARCHAR2 (30 BYTE)
SPECIAL_FEATURE2	VARCHAR2 (30 BYTE)
👉 SUBTYPE2_PK (ID)	

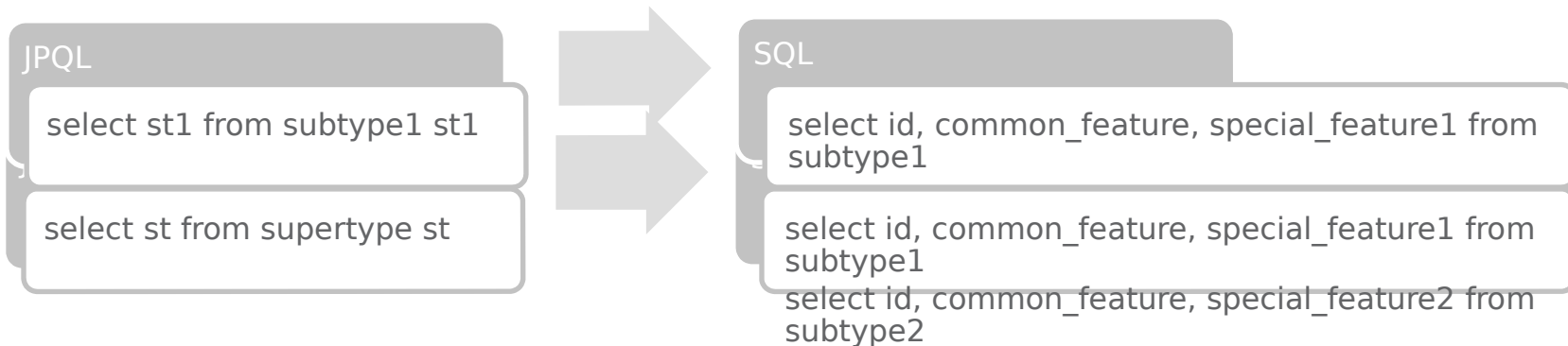
□ No table corresponding to the superclass

→ Cannot define foreign key constraint against supertype as a whole

→ Performance depends on what data are needed

# ■ Mapping Inheritance

## ■ Strategy No. 1: table per concrete class



□ Queries against a single subclass are unproblematic

→ Query against the superclass needs SELECT against both subclass tables

→ May be implemented using a UNION instead of several SELECTs

# ■ Mapping Inheritance

## ■ Strategy No. 2: table per class hierarchy

KEY.ALLTYPES	
ID	NUMBER
TYPE_TYPE	VARCHAR2 (30 BYTE)
COMMON_FEATURE	VARCHAR2 (30 BYTE)
SPECIAL_FEATURE1	VARCHAR2 (30 BYTE)
SPECIAL_FEATURE2	VARCHAR2 (30 BYTE)

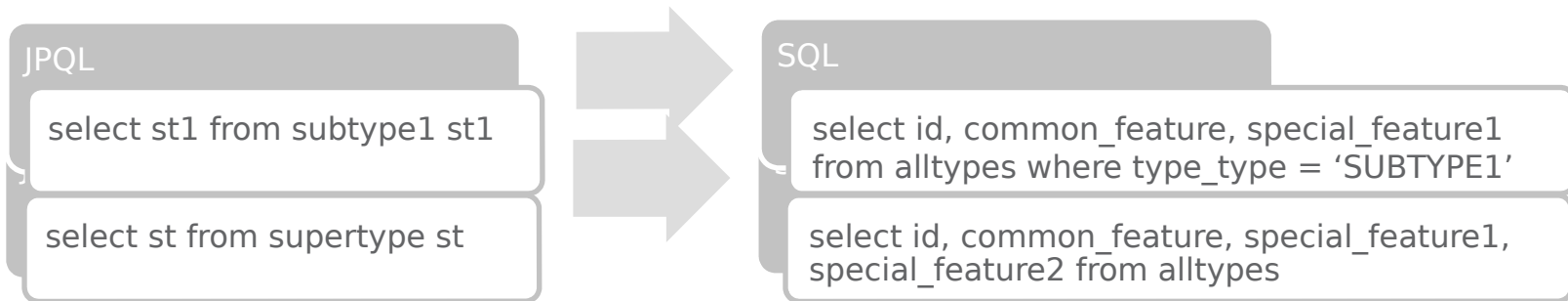
□ Discriminator column designates corresponding object type

→ Nightmare for data integrity (fields must be NULLABLE)

→ Performance-wise, probably best, most of the time

# ■ Mapping Inheritance

## ■ Strategy No. 2: table per class hierarchy



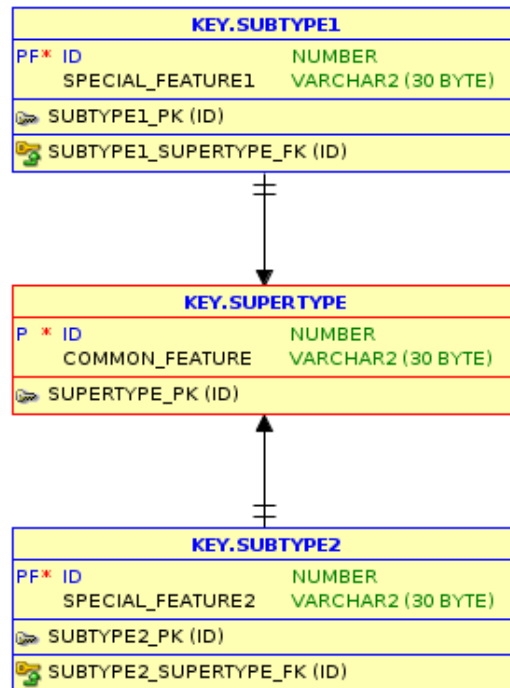
- Index on discriminator column may speed up queries against subtype
- Instead of an explicit discriminator column, some ORMs may allow using NOT NULL checks (CASE WHEN special\_feature IS NOT NULL THEN ...)
- Despite any performance gains, will probably be loathed by most DBAs for its denormalized design ;-)



# ■ Mapping Inheritance

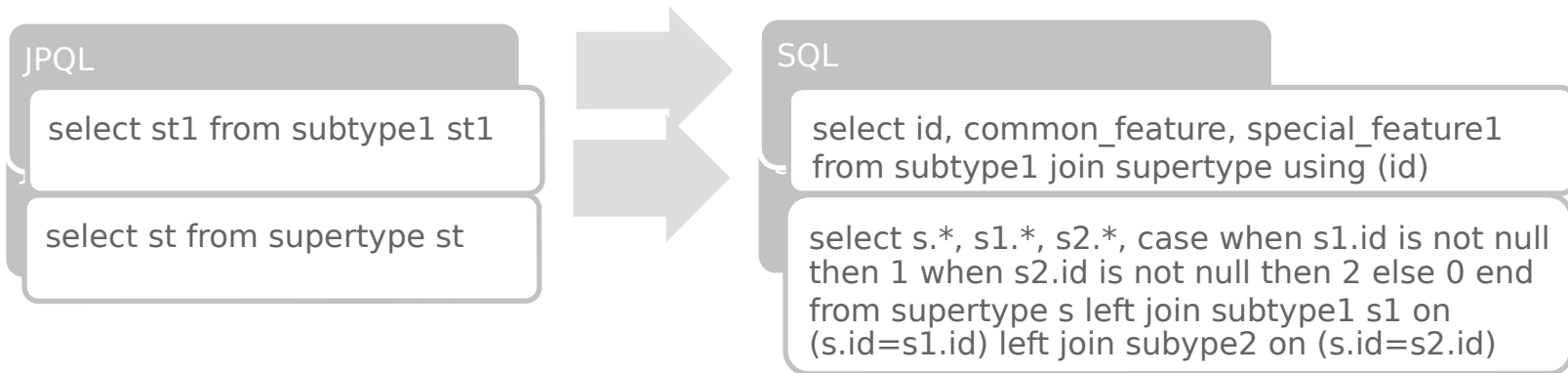
## ■ Strategy No. 3: table per sub- and superclass

- Sub- and superclass linked by foreign key
- Foreign key constraints against supertype are possible
- Creating new subtype takes two inserts



# ■ Mapping Inheritance

## ■ Strategy No. 3: table per sub- and superclass



→ Uses inner join for query against subtype, outer join for query against supertype

→ May quickly become catastrophic for performance

# ■ Mapping Inheritance

- No strategy is universally best
- The most adequate mapping will depend on the depth of the class hierarchy and actual data usage in the application
- E.g., if only queries against subtypes (like “select st1 from subtype1 st1”) are issued, the table per concrete class strategy is optimal

# ■ The “Object-Relational Impedance Mismatch”

## Granularity

KEY.EMPLOYEE	
P *	EMPLOYEE_ID NUMBER
	FIRSTNAME VARCHAR2 (30 BYTE)
	LASTNAME VARCHAR2 (30 BYTE)
	HIRE_DATE DATE
	STREET VARCHAR2 (100 BYTE)
	STREETNO NUMBER
	ZIP NUMBER
	CITY VARCHAR2 (100 BYTE)
	EMPLOYEE_ID_PK (EMPLOYEE_ID)
	EMPLOYEE_ID_PK (EMPLOYEE_ID)



# ■ Granularity

- In object-oriented programming, an Employee class does not contain fields like street or city
- Instead, an Employee *has* an Address (HAS-A relationship):



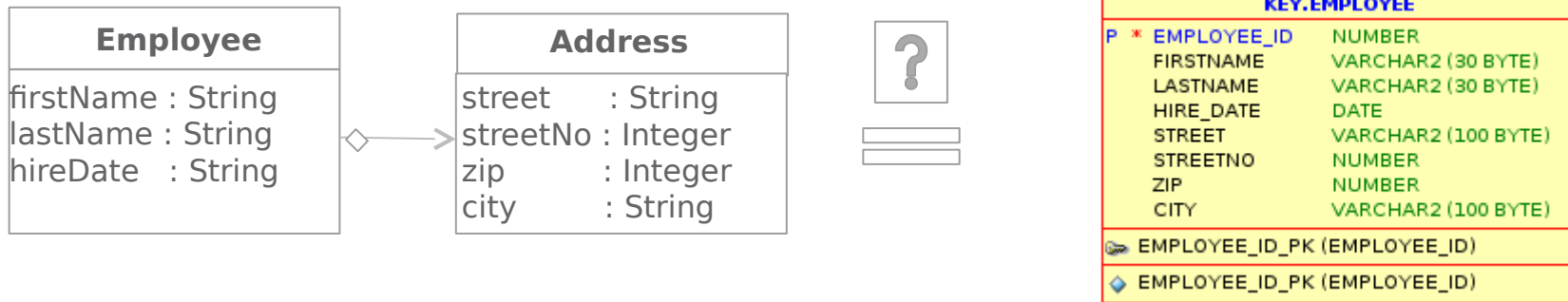
- What does this mean for the persistence framework?

# ■ Granularity

- There are two kinds of objects, entities and value types
- Value types have no independent lifecycle
- Instead, they are persisted when the owning class is persisted
- This equally applies to built-in language types like `java.lang.Integer`
- No need to have same granularity on the database side (thus avoiding performance impact of excessive joins)
- This is more of a thing to keep in mind when doing application design than an insurmountable problem

# ■ The “Object-Relational Impedance Mismatch”

## Object Identity



# ■ Object Identity

- In Java, object identity and object equality are distinct concepts
- If two non-identical objects refer to the same row in the database, data corruption may occur
- The persistence context has to make sure this does not happen
- Again, this is a manageable challenge



# ■ The “Object-Relational Impedance Mismatch”

## Directionality







# ■ Directionality

- In the database, associations may be freely created “on the fly” by joining arbitrary relations (independent of foreign key dependencies)
- In Java, associations are directed
- Associations may be
  - Unidirectional: need e.g. `item.getImages()`, but not `image.getItem()`
  - Bidirectional: need e.g. `project.getTasks()` as well as `task.getProject()`
- If a bidirectional association is many-to-many in both directions (an employee has many projects, a project is worked on by many employees), a mapping table is needed

# Directionality

- Table mapping projects and employees:

KEY.PROJECT_MEMBER	
PF*	PROJECT_ID NUMBER
PF*	EMPLOYEE_ID NUMBER
	BEGIN_DATE DATE
	END_DATE DATE
	PROJECT_ID_EMPLOYEE_ID_PK (PROJECT_ID, EMPLOYEE_ID)
	EMPLOYEE_ID_FK (EMPLOYEE_ID)
	PROJECT_ID_FK (PROJECT_ID)
	PROJECT_ID_EMPLOYEE_ID_PK (PROJECT_ID, EMPLOYEE_ID)

- If the mapping table does not contain any additional columns, this results in a nice and clean design on the Java side:

```
@Entity
@Table(name = "PROJECT")
public class Project implements Serializable {

    @ManyToMany(cascade = CascadeType.PERSIST)
    @JoinTable(name = "PROJECT_MEMBER",
        joinColumns = @JoinColumn(name = "PROJECT_ID"),
        inverseJoinColumns = @JoinColumn(name = "EMPLOYEE_ID")
    )
    private Set<Employee> employees = new HashSet<>();
}
```

```
@Entity
@Table(name = "EMPLOYEE")
@org.hibernate.annotations.BatchSize(size = 1)
public class Employee implements Serializable {

    @ManyToMany(mappedBy = "employees")
    private Set<Project> projects = new HashSet<>();
}
```

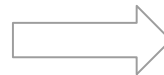
# ■ Directionality

- Often, mapping tables will contain additional information (like e.g., begin\_date and end\_date)
- In this case, an additional class (e.g., ProjectMember) will have to be created on the Java side, effectively messing up the design
- AFAIK, there is no aesthetically pleasing solution to this

# ■ The “Object-Relational Impedance Mismatch”

## Navigation

employee.getTasks().iterator().next().getName()



# ■ Navigation

- In Java, data is retrieved by “walking the object network”
- Naively following the same strategy in the database will lead to disastrous performance
- Extreme (but not unseen, esp. in handwritten frameworks) example:  
`employee.getTasks().size()`, if no care is taken, will fetch all the employee’s tasks from the database just to count them!
- In any case, **what** data you fetch from the database, and **how** you fetch it, is the all-important question when using an ORM

# ■ Agenda

- O/R Mappers – what, why, how
- The “Object-Relational Impedance Mismatch”
- Fetching Data

## ■ Fetch what? – The Fetch Plan

- When asked to retrieve a specific employee, the framework might
  - query just the employee table to retrieve first name, last name, etc.
  - additionally query the task table, in preparation for any upcoming (will it?) `employee.getTasks()`
  - additionally, retrieve the projects these tasks belong to, in preparation for any upcoming (will it?) `task.getProject()`
  - Additionally, query ... (And so forth, up to a configurable limit.)
- The decision what part of the object graph to retrieve is called the **fetch plan**.



# ■ Lazy Fetch

■ With lazy fetching, only the employee table is queried here:

■ Code:

```
Employee employee = em.find(Employee.class, employeeId);  
out.println("Employee " + employeeId + ": " + employee.getLastname());
```

■ SQL (Hibernate):

```
select employee0_.EMPLOYEE_ID as EMPLOYEE_ID1_0_0_, employee0_.CITY as CITY2_0_0_,  
employee0_.FIRSTNAME as FIRSTNAME3_0_0_, employee0_.HIRE_DATE as HIRE_DATE4_0_0_,  
employee0_.LASTNAME as LASTNAME5_0_0_, employee0_.STREET as STREET6_0_0_,  
employee0_.STREETNO as STREETNO7_0_0_, employee0_.ZIP as ZIP8_0_0_  
from EMPLOYEE employee0_ where employee0_.EMPLOYEE_ID=:1
```

# ■ Lazy Fetch

■ Let's assume we are going to process the employee's tasks next:

■ Code:

```
Employee employee = em.find(Employee.class, employeeId);
out.println("Employee " + employeeId + ": " + employee.getLastname());

Set<Task> tasks = employee.getTasks();
tasks.forEach(out::println);
```

■ SQL:

```
select employee0_.EMPLOYEE_ID as EMPLOYEE_ID1_0_0_, employee0_.CITY as CITY2_0_0_,
<...>
from EMPLOYEE employee0_ where employee0_.EMPLOYEE_ID=:1

select tasks0_.EMPLOYEE_ID as EMPLOYEE_ID5_0_0_, tasks0_.TASK_ID as TASK_ID1_3_0_,
tasks0_.TASK_ID as TASK_ID1_3_1_, tasks0_.DESCRIPTION as DESCRIPTION2_3_1_,
tasks0_.EMPLOYEE_ID as EMPLOYEE_ID5_3_1_, tasks0_.NAME as NAME3_3_1_,
tasks0_.PROJECT_ID as PROJECT_ID6_3_1_, tasks0_.STATUS as STATUS4_3_1_
from TASK tasks0_ where tasks0_.EMPLOYEE_ID=:1 ]
```

# ■ Lazy Fetch

- We were fetching just one employee here. What would happen had we asked for a set of employees?

PAR...	EXE...	SQL_TEXT
100	100	select tasks0.EMPLOYEE_ID as EMPLOYEE_ID5_0_0_, tasks0.TASK_ID as TASK_ID1_3_0_, tasks0.TASK_ID as TASK_ID1_3_1_, tasks0
1	1	select employee0.EMPLOYEE_ID as EMPLOYEE_ID1_0_, employee0.CITY as CITY2_0_, employee0.FIRSTNAME as FIRSTNAME3_0_, employ

- The query against task is executed once for every employee...

# ■ Lazy Fetch

- Let's assume we were not interested in just any tasks, but only those that belong to "CAT 1" projects:

```
for (Employee employee : employees) {  
    Set<Task> tasks = employee.getTasks();  
    Set<Task> cat1 = tasks.stream().filter(t -> t.getProject().getName().startsWith("CAT1")).collect(toSet());  
    cat1.forEach(out::println);  
}
```

- For every *distinct* project\_id obtained from the tasks query, we query the project table to find the names:

PARS...	EXE...	SQL_TEXT
100	100	select tasks0.EMPLOYEE_ID as EMPLOYEE_ID5_0_0, tasks0.TASK_ID as TASK_ID1_3_0, tasks0.TASK_ID as TASK_ID1_3_1, tasks0.D
38	38	select project0.PROJECT_ID as PROJECT_ID1_2_0, project0.CREATED as CREATED2_2_0, project0.DESCRPTION as DESCRIPTION3_2_0
1	1	select employee0.EMPLOYEE_ID as EMPLOYEE_ID1_0, employee0.CITY as CITY2_0, employee0.FIRSTNAME as FIRSTNAME3_0, employee

# ■ The $n + 1$ SELECTs Problem

- This is commonly called the “ $n + 1$  SELECTs” problem
- When navigating the object graph with lazy fetching the framework will issue
  - 1 query against the base object’s table,  $n$  being the resulting number of distinct rows, plus
  - $n$  queries against the associated object’s table
- May result in an enormous number of network roundtrips

# ■ Eager Fetch

- Assuming the Employee class was configured to fetch its tasks eagerly, for this ...

```
Employee employee = em.find(Employee.class, employeeId);
out.println("Employee " + employeeId + ": " + employee.getLastname());

Set<Task> tasks = employee.getTasks();
tasks.forEach(out::println);
```

- ... as well as this code ...

```
Employee employee = em.find(Employee.class, employeeId);
out.println("Employee " + employeeId + ": " + employee.getLastname());
```

- ... in the database, we see an outer join to the task table:

```
select employee0_.EMPLOYEE_ID as EMPLOYEE_ID1_0_0_, employee0_.CITY as CITY2_0_0_, <...>,
tasks1_.TASK_ID as TASK_ID1_3_1_, tasks1_.PROJECT_ID as PROJECT_ID6_3_2_, <...>
from EMPLOYEE employee0_ left outer join TASK tasks1_ on employee0_.EMPLOYEE_ID=tasks1_.EMPLOYEE_ID
where employee0_.EMPLOYEE_ID=:1
```

# ■ Eager Fetch

- Assuming that additionally the `Task.project` field was eager fetched:
- For both the above statements, we now have a three table outer join in the database:

```
select employee0_.EMPLOYEE_ID as EMPLOYEE_ID1_0_0_, employee0_.CITY as CITY2_0_0_, <...>,
tasks1_.TASK_ID as TASK_ID1_3_1_, tasks1_.PROJECT_ID as PROJECT_ID6_3_2_, <...>,
project2_.PROJECT_ID as PROJECT_ID1_2_3_, project2_.CREATED as CREATED2_2_3_, <...>|
from EMPLOYEE employee0_ left outer join TASK tasks1_ on employee0_.EMPLOYEE_ID=tasks1_.EMPLOYEE_ID
left outer join PROJECT project2_ on tasks1_.PROJECT_ID=project2_.PROJECT_ID
where employee0_.EMPLOYEE_ID=:1
```

- With eager fetching, as soon as an object is touched, the whole connected object graph is fetched
- Depending on how it is structured, the so called Cartesian Join Problem may appear

# ■ The Cartesian Join Problem

- This Project class has several one-to-many associations that are all eagerly fetched:

```
@OneToMany(mappedBy = "projectId", fetch = FetchType.EAGER)
private Set<Image> images;

@OneToMany(mappedBy = "project", fetch = FetchType.EAGER)
private Set<Task> tasks;
```

- As tasks and images are unrelated, for every project, we fetch all permutations of tasks and images:

PROJECT_ID	TA...	IMAGE_ID
11	99	79
11	99	158
11	100	19
11	100	79
11	100	158
11	101	19
11	101	79
11	101	158



# ■ The Cartesian Join Problem

- Not a problem with many-to-one associations
- With one-to-many associations, may result in enormous amounts of data transferred over the network
- All but a small portion of this data will have to be discarded by the framework
- There is nothing to be done about this in the database

# ■ Lazy vs. Eager Fetch: Questions to Ask

- Whenever I am doing something with object X, will I need X's Y(s), too?
- This associated object, is it actually a Y (many-to-one or one-to-one) or a collection of Ys (one-to-many)?
- How large is the connected portion of the object graph involved?
- With either fetch plan, can I make use of non-default fetch strategies?

# ■ Fetch how? – The Fetch Strategy

- In addition to *what* part of the object graph to fetch, the framework must decide on *how* to access these objects (fetch strategy)
- Available strategies (vendor-dependent) are, e.g.
  - Batch prefetching (with a lazy fetch plan)
  - subselect prefetching
  - breaking up large joins into single selects (with an eager fetch plan)

# ■ Batch Prefetching

- With a lazy fetch plan, batch prefetching may be used to avoid the n+1 SELECTs problem
- Instead of one select per employee to retrieve her tasks, one select is issued per accumulated list of employees (IN-LIST):

```
select tasks0_.EMPLOYEE_ID as EMPLOYEE_ID5_0_1_, tasks0_.TASK_ID as TASK_ID1_3_1_, <...>,
from TASK tasks0_ where tasks0_.EMPLOYEE_ID in (:1 , :2 , :3 , :4 , :5 , :6 , :7 , :8 , :9 , :10 )
```

- Batch size may be configurable (vendor-dependent)
- Turns n+1 SELECTs into n/<batch size>+1 SELECTs

# ■ Batch Prefetching: Pros and Cons

- Pro: Avoid excessive network roundtrips
- Con: With longer in-lists, an index on the filtering column is less likely to be used
- Net result will depend on various global (network latency ...) and use case specific (amount of data, goodness of index ...) factors
- Conclusion: test the concrete scenarios!

## ■ Subselect Prefetching

- Fetches the associated objects as a whole *as soon as* the first of them is accessed
- Instead of passing an evaluated in-list, the selection is restricted by the same query that was used to retrieve the base objects:

```
select tasks0_.EMPLOYEE_ID as EMPLOYEE_ID5_0_1_, tasks0_.TASK_ID as TASK_ID1_3_1_, <...>,
from TASK tasks0_ where tasks0_.EMPLOYEE_ID in
((select employee0_.EMPLOYEE_ID from EMPLOYEE employee0_ where employee0_.LASTNAME like '%')
```

- Availability varies (vendor-dependent)
- Turns n+1 SELECTs into 1+1 SELECTs

## ■ Subselect Prefetching: Pros and Cons

- PRO: Reduces network roundtrips to a minimum (with lazy fetch plan)
- PRO: Unlike with batch prefetching, no need to outsmart the system ;-)
- PRO: leaves optimization to the database
- PRO: in theory, possibly the optimal solution – fetch only when needed, and let the database decide how!
- CON: Is there? There could be - if the database is not able to transform the subselect into a join □ check!
- Conclusion: Check what is actually going on in the database!

# ■ Conclusion

- Know what is possible in your ORM
- Check out what is actually sent to the database, AND
- Check with your DBA how it performs!
- (DBAs: don't just curse that ORM ... but advise)
- In a nutshell: **let's talk to each other** □



# Thank you!

Sigrid Keydana

Tech Event, Sept. 11 2015

