Java & SQL
Stronger Together

@MarkusWinand   @ModernSQL
“We can solve any problem by introducing an extra level of abstraction”

(Based on the “fundamental theorem of software engineering”)
JDBC
Database
Provides unique API for different SQL Databases
JPA
JDBC
Database

Maps objects from/to tables

Provides unique API for different SQL Databases
Spring Data JPA

JPA

JDBC

Database

Provides unique API for different SQL Databases

Maps objects from/to tables

Makes common tasks very simple
Your Code
Spring Data JPA
JPA
JDBC
Database

Makes common tasks very simple
Maps objects from/to tables
Provides unique API for different SQL-Databases
Your Code
Spring Data JPA
JPA
JDBC
Database

90%?

Doesn't provide 100% of underlying functionality

Maps Objects from/to tables

Makes common tasks very simple

Provides unique API for different SQL Databases

90%?
Your Code
Spring Data JPA
JPA
JDBC
Database

Proprietary data types?
\rightarrow (VendorCls)jdbcCls

90%?

Makes common tasks very simple

Maps Objects from/to tables

Provides unique API for different SQL Databases
The processing is not about objects?
-> QueryBuilder, JPQL or native queries

Proprietary data types?
-> (VendorCls)jdbcCls

Your Code
Spring Data JPA
JPA
JDBC
Database

90%?

Makes common tasks very simple
Maps Objects from/to tables
Provides unique API for different SQL Databases
“Hibernate’s design goal is to relieve the developer from 95% of common data persistence-related programming tasks[...] Hibernate does not hide the power of SQL from you[...]”

— Hibernate ORM User Guide (second paragraph)
Database
JDBC
JPA
Spring Data JPA
Your Code

- Provides unique API for different SQL Databases
- Maps Objects from/to tables
- Makes common tasks very simple
- The processing is not about objects? → QueryBuilder, JPQL or native queries
- Proprietary data types? → (VendorCls)jdbcCls

90%?
Your Code

Spring Data JPA

JPA

JDBC

Database

Proprietary data types?
→ (VendorCls)jdbcCls

The processing is not about objects?
→ QueryBuilder, JPQL or native queries

findBy...
not powerful enough?
→ QueryBuilder, JPQL or native queries

90%?

Makes common tasks very simple

Maps Objects from/to tables

Provides unique API for different SQL-Databases

not powerful enough?
→ QueryBuilder, JPQL or native queries
What is the right level of abstraction for any given problem?
Pattern #1: Lists

Spring BUCKET LIST

1. Go for a walk in the woods
2. Picnic
3. Improve SQL-foo
List: All Attributes

Your Code

Spring Data JPA

JPA

JDBC

Database
List: All Attributes

Your Code
Spring Data JPA
JPA
JDBC
Database

SELECT * FROM tbl
Your Code

```java
ResultSet rs = c.prepareStatement("SELECT *
+ " FROM tbl")
.executeQuery();

SELECT * FROM tbl
```
Your Code

Spring Data JPA

JPA

JDBC

Database

Collection<Entity> c = em.createQuery("FROM Entity")
.getResultList();

ResultSet rs = c.prepareStatement("SELECT *
+ " FROM tbl")
.executeQuery();

SELECT * FROM tbl
Collection\<Entity\> c = r.findAll();

Collection\<Entity\> c = em.createQuery("FROM Entity")
    .getResultList();

ResultSet rs = c.prepareStatement("SELECT *
                                   + " FROM tbl")
    .executeQuery();

SELECT * FROM tbl
List: Some Attributes

Your Code
Spring Data JPA
JPA
JDBC
Database
List: Some Attributes

Your Code
Spring Data JPA
JPA
JDBC
Database

SELECT a1, ... FROM tbl
List: Some Attributes

Your Code
Spring Data JPA
JPA
JDBC
Database

SELECT a1, ... FROM tbl

Runtime improvement: ~0% - 10000%
List: Some Attributes

Your Code
Spring Data JPA
JPA
JDBC
Database

Affects:

Runtime improvement:
~0% - 10000%

SELECT a1, ... FROM tbl
List: Some Attributes

- JDBC
- JPA
- Spring Data JPA
- Database

Your Code

SELECT a1, ... FROM tbl

Runtime improvement:
~0% - 10000%

Affects:
- Java: GC
List: Some Attributes

Your Code
Spring Data JPA
JPA
JDBC
Database

Affects:
- Java: GC
- Network

Runtime improvement:
\(~0\% - 10000\%

```
SELECT a1, ... FROM tbl
```
SELECT a1, ... FROM tbl
List: Some Attributes

- Your Code
- Spring Data JPA
- JPA
- JDBC
- Database

SELECT a1, ... FROM tbl

Affects:
- Java: GC
- Network
- Database:
  - Index Only Scan

Runtime improvement:
~0% - 10000%
SELECT a1, ... FROM tbl
ResultSet rs =
c.prepareStatement("SELECT a1, ...
+ " FROM tbl")
.executeQuery();
List: Some Attributes

Your Code

Spring Data JPA

JPA

JDBC

Database

Collection<DTO> c =
em.createQuery("SELECT new DTO(a1, ...) + " FROM Ent")
.getResultList();

ResultSet rs =
c.prepareStatement("SELECT a1, ... + " FROM tbl")
.executeQuery();

SELECT a1, ... FROM tbl
List: Some Attributes

Your Code
Spring Data JPA
JPA
JDBC
Database

Collection<DTO> c =
em.createQuery("SELECT new DTO(a1, ...)"
 + " FROM Ent")
.getResultList();

ResultSet rs =
c.prepareStatement("SELECT a1, ...
 + " FROM tbl")
.executeQuery();

SELECT a1, ... FROM tbl
Collection<DTO> c = em.createQuery("SELECT new DTO(a1, ...)" + " FROM Ent")
  .getResultList();

ResultSet rs = c.prepareStatement("SELECT a1, ..." + " FROM tbl")
  .executeQuery();

SELECT a1, ... FROM tbl
public interface Proj {
    ... getA1();
    ...
}

Collection<P> findAllProjectedBy(Class<P> p);

Collection<DTO> c = em.createQuery("SELECT new DTO(a1, ...)" + " FROM Ent")
    .getResultList();

ResultSet rs = c.prepareStatement("SELECT a1, ...
    + " FROM tbl")
    .executeQuery();

SELECT a1, ... FROM tbl
List: Some Attributes

Your Code

Spring Data JPA
JPA
JDBC
Database

public interface Proj {
    ... getA1();
    ...
}

Collection<Proj> c = r.findAllProjectedBy(Proj.class);

Collection<DTO> c = em.createQuery("SELECT new DTO(a1, ...)"
+ " FROM Ent")
    .getResultList();

Or:
LAZY attributes

ResultSet rs = c.prepareStatement("SELECT a1, ...
+ " FROM tbl")
    .executeQuery();
List: Attributes of Multiple Objects

Your Code
Spring Data JPA
JPA
JDBC
Database
SELECT a.a1, b.a1, ... FROM tbl a
JOIN tbl2 b ON ...

List: Attributes of Multiple Objects
List: Attributes of Multiple Objects

Your Code

```java
Collection<DTO> c = em.createQuery("SELECT 
+ "new DTO(a.a1, b.a1, ...)"
+ " FROM Entity a JOIN a.child b")
.getResultList();
```

### SQL Query

```sql
SELECT a.a1, b.a1, ... FROM tbl a
JOIN tbl2 b ON ...
```
Spring Data JPA

JPA

Database

List: Attributes of Multiple Objects

Your Code

Collection<P> findAllProjectedBy(Class<P> p);

Collection<Proj> c = r.findAllProjectedBy(Proj.class);

Collection<DTO> c = em.createQuery("SELECT a.a1, b.a1, ...
+ " FROM Entity a JOIN a.child b")
.getResultList();

SELECT a.a1, b.a1, ...
 FROM tbl a
 JOIN tbl2 b ON ...
List: Attributes of Multiple Objects

```java
public interface Proj {
    ... getA1();
    Proj2 getB();
}
```

```sql
Collection<DTO> c =
em.createQuery("SELECT 
    new DTO(
        a.a1,
        b.a1, ...
    )
    FROM Entity a
    JOIN a.child b")
.getResultList();
```

```java
Collection<Proj> c =
r.findAllProjectedBy(Proj.class);
```

Collection<DTO> c =
em.createQuery("SELECT 
    a.a1, b.a1, ...
    FROM tbl a
    JOIN tbl2 b ON ..."
).getResultList();
```
List: Attributes of Multiple Objects

Still selects all columns DATAJPA-1218

public interface Proj {
    ... getA1();
    Proj2 getB();
}

Collection<P> findAllProjectedBy(Class<P> p);

Collection<Proj> c =
    r.findAllProjectedBy(Proj.class);

Collection<DTO> c =
    em.createQuery("SELECT 
        + "new DTO(a.a1, b.a1, ...)"
        + " FROM Entity a JOIN a.child b"
    )
    .getResultList();

SELECT a.a1, b.a1, ... FROM tbl a
    JOIN tbl2 b ON ...
List: Attributes of Multiple Objects

public interface Proj {
    ... getA1();
    Proj2 getB();
}

@Query("SELECT "+ "new DTO(a.a1, b.a1, ...)" + " FROM Entity a JOIN a.child b")
Collection<DTO> findAllDtoProj();

Collection<DTO> c = r.findAllDtoProj();
public interface Proj {
    ... getA1();
    Proj2 getB();
}

@Query("SELECT 
    " + "new DTO(a.a1, b.a1, ...)"
    + " FROM Entity a JOIN a.child b"
)
Collection<DTO> findAllDtoProj();

Collection<DTO> c = r.findAllDtoProj();

Collection<DTO> c = em.createQuery("SELECT 
    " + "new DTO(a.a1, b.a1, ...)"
    + " FROM Entity a JOIN a.child b"
).getResultList();

SELECT a.a1, b.a1, ... FROM tbl a 
    JOIN tbl2 b ON ...
List: Derived Data

Your Code
Spring Data JPA
JPA
JDBC
Database
SELECT a.a1, \textcolor{red}{\textit{SUM(b.a1)}} FROM tbl a
JOIN tbl2 b
\textcolor{green}{\textbf{GROUP BY} a.a1}
Your Code

Spring Data JPA

JPA

JDBC

Database

Collection<DTO> c =
em.createQuery("SELECT "+ "new DTO(a.a1, SUM(b.a1))"
+ " FROM Entity a JOIN a.child b"
+ " GROUP BY a"
).getResultList();

SELECT a.a1, SUM(b.a1) FROM tbl1 a 
JOIN tbl2 b 
GROUP BY a.a1
List: Derived Data

@Query("SELECT " + "new DTO(a.a1, \texttt{SUM(b.a1)})" + " FROM Entity a JOIN a.child b" + " \texttt{GROUP BY a}"")
Collection<DTO> findAllDtoProj();

Collection<DTO> c = r.findAllDtoProj();

SELECT a.a1, \texttt{SUM(b.a1)} FROM tbl a JOIN tbl2 b
\texttt{GROUP BY a.a1}
Your Code
Spring Data JPA
JPA
JDBC
Database

SELECT a.a1, SUM(b.a1), SUM(CASE WHEN b.a2=1 THEN b.a1 ELSE 0 END) FROM tbl a JOIN tbl2 b GROUP BY a.a1
List: Derived Data

Your Code

Spring Data JPA

JPA

JDBC

Database

More beautifully, but rarely supported:

```sql
SELECT a.a1, SUM(b.a1), SUM(CASE WHEN b.a2 = 1 THEN b.a1 ELSE 0 END) 
FROM tbl a JOIN tbl2 b 
GROUP BY a.a1
```

```sql
SELECT a.a1, SUM(b.a1), SUM(b.a1) FILTER (WHERE b.a2 = 1) 
FROM tbl a JOIN tbl2 b 
GROUP BY a.a1
```
List: Derived Data

More beautifully, but rarely supported:

```
SELECT a.a1, SUM(b.a1)
    , SUM(CASE WHEN b.a2=1 THEN b.a1 ELSE 0 END)
FROM tbl a JOIN tbl2 b
GROUP BY a.a1
```

```
SELECT a.a1, SUM(b.a1)
    , SUM(b.a1) FILTER (WHERE b.a2=1)
FROM tbl a JOIN tbl2 b
GROUP BY a.a1
```

### JDBC Database

<table>
<thead>
<tr>
<th>Year</th>
<th>MariaDB</th>
<th>MySQL</th>
<th>PostgreSQL</th>
<th>SQLite</th>
<th>DB2 LUW</th>
<th>Oracle</th>
<th>SQL Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999</td>
<td>5.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2007</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2015</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2019</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SELECT a.a1, SUM(b.a1),
    SUM(CASE WHEN b.a2=1 THEN b.a1 ELSE 0 END)
FROM tbl a JOIN tbl2 b
GROUP BY a.a1

More beautifully, but rarely supported:
SELECT a.a1, SUM(b.a1),
    SUM(b.a1) FILTER (WHERE b.a2=1)
FROM tbl a JOIN tbl2 b
GROUP BY a.a1

JPQL doesn't allow CASE in aggregates

SELECT a.a1, SUM(b.a1),
    SUM(CASE WHEN b.a2=1 THEN b.a1 ELSE 0 END)
FROM tbl a JOIN tbl2 b
GROUP BY a.a1
List: Derived Data

Your Code

Spring Data JPA

JPA

JDBC

Database

SELECT a.a1, SUM(b.a1), SUM(CASE WHEN b.a2=1 THEN b.a1 ELSE 0 END) FROM tbl a JOIN tbl2 b GROUP BY a.a1

JPQL doesn't allow CASE in aggregates
List: Derived Data

@NamedNativeQuery(
    name = "Entity.findAllXy",
    query = "<<SQL>>",
    resultSetMapping = "...")
@SqlResultSetMapping(...)
public class Entity {
...

JPQL doesn't allow CASE in aggregates

SELECT a.a1, SUM(b.a1), SUM(CASE WHEN b.a2=1 THEN b.a1 ELSE 0 END)
FROM tbl a JOIN tbl2 b
GROUP BY a.a1
Your Code

### Spring Data JPA

**JPA**

**JDBC**

**Database**

JPQL doesn't allow `CASE` in aggregates

```sql
SELECT a.a1, SUM(b.a1), SUM(CASE WHEN b.a2 = 1 THEN b.a1 ELSE 0 END) FROM tbl a JOIN tbl2 b GROUP BY a.a1
```

```java
@NamedNativeQuery(
    name = "Entity.findAllXy",
    query = "<<SQL>>",
    resultSetMapping = "..."
)
@SqlResultSetMapping(...)
public class Entity {...

Collection<DTO> c = em.createNamedQuery("Entity.findAllXy").getResultList();
```

List: Derived Data
JPQL doesn't allow `CASE` in aggregates.

```sql
SELECT a.a1, SUM(b.a1), SUM(CASE WHEN b.a2=1 THEN b.a1 ELSE 0 END)
FROM tbl a JOIN tbl2 b
GROUP BY a.a1
```
List: Derived Data

Your Code
Spring Data JPA
JPA
JDBC
Database

SELECT a.a1, SUM(b.a1)
, SUM(CASE WHEN b.a2=1 THEN b.a1 ELSE 0 END)
FROM tbl a
JOIN tbl2 b
GROUP BY a.a1

JPQL doesn't allow CASE in aggregates
Your Code

Spring Data JPA

JPA

JDBC

Database

List: Derived Data

Blaze Persistence
Entity Views & JPQL next

JPQL doesn't allow CASE in aggregates

SELECT a.a1, SUM(b.a1), SUM(CASE WHEN b.a2 = 1 THEN b.a1 ELSE 0 END) FROM tbl a JOIN tbl2 b GROUP BY a.a1

@EntityView(Entity.class)
interface Proj {
    Integer getA1();
    @Mapping("SUM(b.a1)")
    Integer getSum1();
    @Mapping("SUM(b.a1) FILTER (WHERE a2=1)")
    Integer getSum2();
}

SELECT a.a1, SUM(b.a1), SUM(CASE WHEN b.a2=1 THEN b.a1 ELSE 0 END) FROM tbl a JOIN tbl2 b GROUP BY a.a1
Application
Change
Load
Store
ORMs are super useful for the Load-Change-Store cycle on complex entity graphs.
ORMs are super useful for the Load-Change-Store cycle on complex entity graphs.

If the cycle is broken, entities might not be the right level of abstraction.
Pattern #2: Search
Pattern #2: Search
SELECT *
FROM tbl a
<<super_complex_query>>
Search: Load Entity from Native Query

Your Code

Spring Data JPA

JPA

JDBC

Database

@NamedNativeQuery(
    name = "Entity.findSpecialOnes",
    query = "<<SQL>>",
    resultSetMapping = "...")

@SqlResultSetMapping(
    @entities = @EntityResult(
        entityClass = Entity.class))

public class Entity {...}

SELECT *
FROM tbl a
<<super_complex_query>>
@Query(native=true)
Collection<Entity> findSpecialOnes();

@NamedNativeQuery(
    name = "Entity.findSpecialOnes",
    query = "<<SQL>>",
    resultSetMapping = "...")
@SqlResultSetMapping(
    @entities = @EntityResult(
        entityClass = Entity.class))
public class Entity {...}

SELECT *
FROM tbl a
<<super_complex_query>>
CREATE TABLE \( t \) (  
id INTEGER,  
parent INTEGER,
)
CREATE TABLE t (  
id INTEGER,  
parent INTEGER,  
)
CREATE TABLE t ( id INTEGER, parent INTEGER, )
CREATE TABLE t ( id INTEGER, parent INTEGER, )
public void descendTree(Entity start) {
    for (Entity e : start.getChildren()) {
        descendTree(e);
    }
}
public void descendTree(Entity start) {
    for (Entity e : start.getChildren()) {
        descendTree(e);
    }
}
public void descendTree(Entity start) {
    for (Entity e : start.getChildren()) {
        descendTree(e);
    }
}

SELECT ...
FROM Entity
WHERE parent = ?
public void descendTree(Entity start) {
    for (Entity e : start.getChildren()) {
        descendTree(e);
    }
}

SELECT ...
FROM Entity
WHERE parent = ?
public void descendTree(Entity start) {
    for (Entity e : start.getChildren()) {
        descendTree(e);
    }
}

SELECT ...
FROM Entity
WHERE parent = ?
public void descendTree(Entity start) {
    for (Entity e : start.getChildren()) {
        descendTree(e);
    }
}

SELECT ...
FROM Entity
WHERE parent = ?

public void ascendTree(Entity e) {
    while (e != null) {
        e = e.getParent();
    }
}
public void descendTree(Entity start) {
    for (Entity e : start.getChildren()) {
        descendTree(e);
    }
}

SELECT ... FROM Entity WHERE parent = ?

public void ascendTree(Entity e) {
    while (e != null) {
        e = e.getParent();
    }
}

SELECT ... FROM Entity WHERE id = ?
SELECT t.id, t.parent
FROM t
WHERE t.id = ?
SELECT t.id, t.parent
FROM t
WHERE t.id = ?
UNION ALL
SELECT t.id, t.parent
FROM t
WHERE t.parent = ?
SELECT t.id, t.parent
FROM t
WHERE t.id = ?
UNION ALL
SELECT t.id, t.parent
FROM t
WHERE t.parent = ?
SELECT t.id, t.parent
FROM t
WHERE t.id = ?
UNION ALL
SELECT t.id, t.parent
FROM t
WHERE t.parent = ?
WITH RECURSIVE prev (id, parent) AS ( 
    SELECT t.id, t.parent 
    FROM t 
    WHERE t.id = ? 
    UNION ALL 
    SELECT t.id, t.parent 
    FROM t 
    JOIN prev ON t.parent = prev.id 
) 
SELECT * FROM prev
WITH RECURSIVE prev (id, parent) AS (  
  SELECT t.id, t.parent  
  FROM t  
  WHERE t.id = ?  
  UNION ALL  
  SELECT t.id, t.parent  
  FROM t  
  JOIN prev ON t.parent = prev.id  
  )  
SELECT * FROM prev
WITH RECURSIVE prev (id, parent) AS ( 
  SELECT t.id, t.parent 
  FROM t 
  WHERE t.id = ? 
UNION ALL 
  SELECT t.id, t.parent 
  FROM t 
  JOIN prev ON prev.parent = t.id 
) 
SELECT * FROM prev
Standard SQL (ISO 9075)
Vendor Dialect
Vendor Dialect

Standard SQL (ISO 9075)

Vendor Dialect

HQL
Spring .findBy...
JPQL
HQL
Standard SQL (ISO 9075)
Vendor Dialect

* without keyword recursive
JVM
public void ascendTree(Entity e) {
    while (e != null) {
        e = e.getParent();
    }
}
public void ascendTree(Entity e) {
    while (e != null) {
        e = e.getParent();
    }
}
public void ascendTree(Entity e) {
    while (e != null) {
        e = e.getParent();
    }
}
public void ascendTree(Entity e) {
    while (e != null) {
        e = e.getParent();
    }
}
public void ascendTree(Entity e) {
    while (e != null) {
        e = e.getParent();
    }
}
public void ascendTree(Entity e) {
    while (e != null) {
        e = e.getParent();
    }
}
public void ascendTree(Entity e) {
  while (e != null) {
    e = e.getParent();
  }
}
What if I run the recursive query?
What if I run the recursive query

```sql
WITH RECURSIVE prev (id, parent) AS (
    SELECT t.id, t.parent
    FROM t
    WHERE t.id = ?
    UNION ALL
    SELECT t.id, t.parent
    FROM t
    JOIN prev ON prev.parent = t.id
) 
SELECT * FROM prev
```
What if I run the recursive query before my JPA logic?

WITH RECURSIVE prev (id, parent) AS (
    SELECT t.id, t.parent
    FROM t
    WHERE t.id = ?
    UNION ALL
    SELECT t.id, t.parent
    FROM t
    JOIN prev ON prev.parent = t.id
)
SELECT * FROM prev
JVM

PersistenceContext
```java
@NamedNativeQuery(
    name = "Entity.findSpecialOnes",
    query = "<<SQL>>",
    resultSetMapping = "..."
)
@SqlResultSetMapping(
    @entities = @EntityResult(
        entityClass = Entity.class)
)
public class Entity {...}
```
JVM

findSpecialOnes(); // Warm Up

PersistenceContext
JVM

findSpecialOnes(); // Warm Up

PersistenceContext
findSpecialOnes(); // Warm Up
public void ascendTree(Entity e) {
    while (e != null) {
        e = e.getParent();
    }
}

PersistenceContext

findSpecialOnes(); // Warm Up
JVM

PersistenceContext
@NamedNativeQuery(
    name = "Entity.findSpecialOnes",
    query = "<<SQL>>",
    resultSetMapping = "...")
@SqlResultSetMapping(
    @entities = @EntityResult(
        entityClass = Entity.class))
public class Entity {...}
JVM

findSpecialOnes(); // Warm Up

PersistenceContext
findSpecialOnes(); // Warm Up
findSpecialOnes(); // Warm Up
```java
public void descendTree(Entity s) {
    for (Entity e : s.getChildren()) {
        descendTree(e);
    }
}
```

```java
findSpecialOnes(); // Warm Up
```
findSpecialOnes(); // Warm Up

public void descendTree(Entity s) {
    for (Entity e : s.getChildren()) {
        descendTree(e);
    }
}
PersistenceContext cannot be searched by PARENT*

JVM

```java
findSpecialOnes(); // Warm Up
```

```java
defendTree(Entity s) {
    for (Entity e : s.getChildren()) {
        descendTree(e);
    }
}
```

* Possible Solution: EclipseLink @CacheIndex
PersistenceContext cannot be searched by PARENT

```java
public void descendTree(Entity s) {
    for (Entity e : s.getChildren()) {
        descendTree(e);
    }
}

SELECT ...
FROM Entity
WHERE parent = ?

* Possible Solution: EclipseLink @CacheIndex
# Pre-Load via Modern SQL

## Limitations

1. *Ineffective if Java code doesn’t access via @Id.*
   (EclipseLink @CacheIndex?)

2. `@NativeQueries` flush the PersistenceContext

## Helpful Tools

1. BlazePersistence
2. jOOQ.org
3. (myBatis, QueryDSL)
A lot has happened since SQL-92
A lot has happened since SQL-92.

SQL has evolved beyond the relational idea.
SQL has evolved beyond the relational idea.

A lot has happened since SQL-92.

If you use SQL for CRUD operations only, you are doing it wrong.
SQL has evolved beyond the relational idea.

A lot has happened since SQL-92.

If you use SQL for CRUD operations only, you are doing it wrong.

https://modern-sql.com
@ModernSQL by @MarkusWinand