

# Using Relational Databases With ScalaQuery



**Stefan Zeiger**

# Why Relational Databases?

- Well-Proven Model For Many Applications
- Prevent Data Silos
- Because That's Where Your Data Is (Duh!)



**But We Have JDBC!**

# But We Have JDBC!

```
def usersMatching(pattern: String)(conn: Connection) = {  
  val st = conn.prepareStatement("select id, name from users where name like ?")  
  try {  
    st.setString(1, pattern)  
    val rs = st.executeQuery()  
    try {  
      val b = new ListBuffer[(Int, String)]  
      while(rs.next)  
        b.append((rs.getInt(1), rs.getString(2)))  
      b.toList  
    } finally rs.close()  
  } finally st.close()  
}
```

```
Class.forName("org.h2.Driver")  
val conn = DriverManager.getConnection("jdbc:h2:test1")  
try {  
  println(usersMatching("%zeiger%")(conn))  
} finally conn.close()
```

# JDBC

- Good Basis For Frameworks
- Abstraction Level Too Low For Applications

# ScalaQuery: Simple Queries

Write your own SQL


```
val usersMatching = query[String, (Int, String)]  
  ("select id, name from users where name like ?")
```

```
Database.forURL("jdbc:h2:test1", driver = "org.h2.Driver") withSession {  
  println(usersMatching("%zeiger%").list)  
}
```

**But We Have ORMs!**

# But We Have ORMs!

- Object/Relational Mapping Tools
  - Hibernate, Toplink, JPA

- They solve  of the problem



**“Object/Relational Mapping is  
The Vietnam of Computer Science”**  
(Ted Neward)

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

# Relational Model

## Relational Model:

- Relation

- Attribute

- Tuple

- Relation Value

- Relation Variable

COF_NAME	SUP_ID	PRICE
Colombian	101	7.99
French_Roast	49	8.99
Espresso	150	9.99
Colombian_Decaf	101	8.99
French_Roast_Decaf	49	9.99

TABLE **COFFEES**

Examples from: <http://download.oracle.com/javase/tutorial/jdbc/basics/index.html>

# Impedance Mismatch: Concepts

## Object-Oriented:

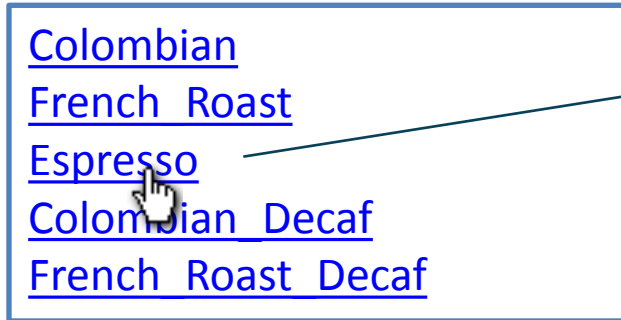
- Identity
- State
- Behaviour
- Encapsulation

## Relational:

- Identity
- State: Transactional
- Behaviour
- Encapsulation

# Impedance Mismatch: Retrieval

Colombian  
French Roast  
Espresso  
Colombian Decaf  
French Roast Decaf

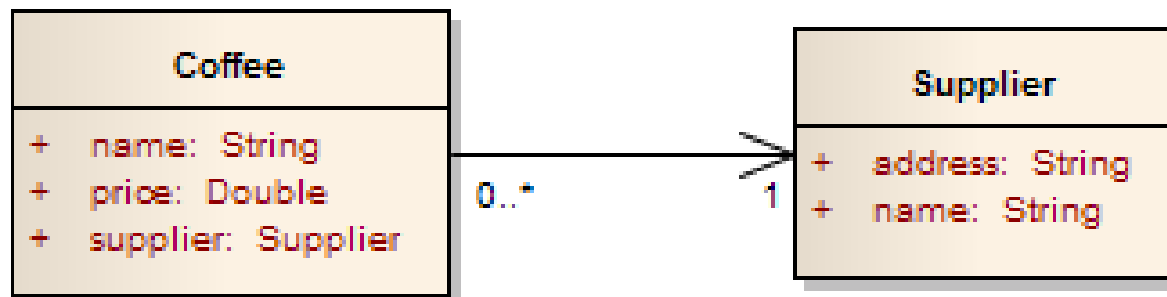
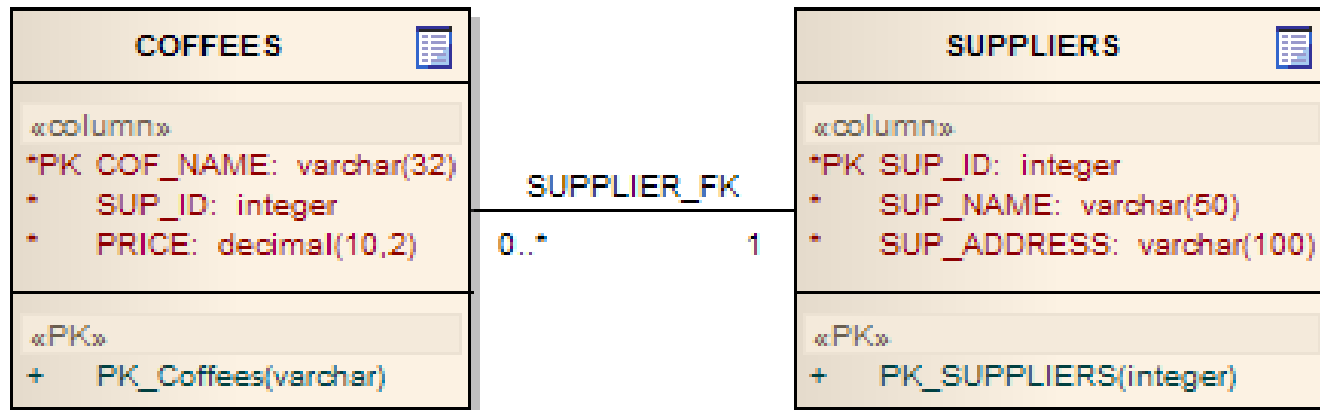


**Espresso**  
Price: 9.99  
Supplier: The High Ground

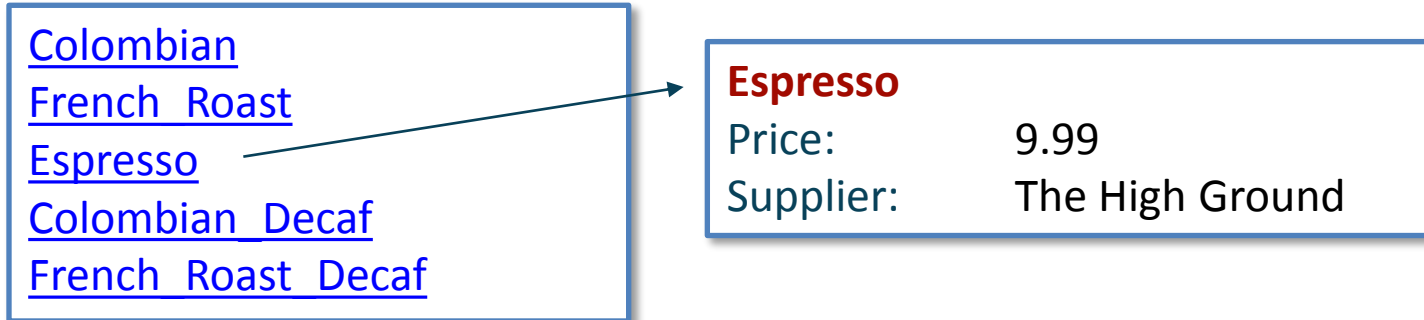
```
select COF_NAME  
from COFFEES
```

```
select c.*, s.SUP_NAME  
from COFFEES c, SUPPLIERS s  
where c.COF_NAME = ?  
and c.SUP_ID = s.SUP_ID
```

# Impedance Mismatch: Retrieval



# Impedance Mismatch: Retrieval



```
def getAllCoffees(): Seq[Coffee] = ...

def printLinks(s: Seq[Coffee]) {
  for(c <- s) println(c.name + " " + c.price)
}

def printDetails(c: Coffee) {
  println(c.name)
  println("Price: " + c.price)
  println("Supplier: " + c.supplier.name)
}
```

# O/R Mapper

- Mapping low-level programming (OOP) to high-level concepts (relational algebra)
- Not transparent

# Better Match: Functional Programming

- Relation
  - Attribute
  - Tuple
  - Relation Value
  - Relation Variable - mutable state in the DB
- ```
case class Coffee(name: String,  
supplierId: Int, price: Double)  
  
val coffees = Set(  
  Coffee("Colombian", 101, 7.99),  
  Coffee("French_Roast", 49, 8.99),  
  Coffee("Espresso", 150, 9.99)  
)
```
-

# ScalaQuery

# ScalaQuery

- **A Scala API for database access:** Remove all the boilerplate!
- **Compile-time checking and type-safety:** Write Scala code instead of SQL
- **Composable non-leaky abstractions:** Query comprehensions + explicit DB calls
- Natively supports PostgreSQL, MySQL, H2, HSQLDB/HyperSQL, Derby/JavaDB, MS SQL Server, MS Access, SQLite



# Scala

- A programming language running **on the JVM** (soon also on .NET)
- **Statically typed**, combines object-orientation and functional programming
- **Concise**
- **Fully interoperable** with Java
- **As fast** as Java
- Provides the necessary **abstractions** for a library like ScalaQuery (unlike e.g. Java)

# ScalaQuery

- **Type-Safe Queries in Scala** `org.scalaquery.q1`  
+ Insert, Update, Delete, DDL
- **Plain SQL Statements** `org.scalaquery.simple`
- **Session Management** `org.scalaquery.session`
- **Common API For Executing Statements** `org.scalaquery`

# Queries on Collections

```
case class Coffee(  
  name: String,  
  supID: Int,  
  price: Double  
)
```

```
val coffees = List(  
  Coffee("Colombian", 101, 7.99),  
  Coffee("Colombian_Decaf", 101, 8.99),  
  Coffee("French_Roast_Decaf", 49, 9.99)  
)
```

```
val l = for {  
  c <- coffees if c.supID == 101  
} yield (c.name, c.price)
```

```
l.foreach { case (n, p) => println(n + ": " + p) }
```

Scala Collections

# Queries on Database Tables

```
val Coffees = new Table[(String, Int, Double)]("COFFEES") {  
  def name = column[String]("COF_NAME")  
  def supID = column[Int]("SUP_ID")  
  def price = column[Double]("PRICE")  
  def * = name ~ supID ~ price  
}
```

```
Coffees.insertAll(  
  ("Colombian", 101, 7.99),  
  ("Colombian_Decaf", 101, 8.99),  
  ("French_Roast_Decaf", 49, 9.99)  
)
```

```
val q = for {  
  c <- Coffees if c.supID === 101  
} yield c.name ~ c.price
```

```
q.foreach { case (n, p) => println(n + ": " + p) }
```

ScalaQuery

# Table Definitions

```
val Suppliers = new Table[(Int, String, String, String, String, String)]("SUPPLIERS") {  
  
  def id      = column[Int]("SUP_ID", 0.PrimaryKey)  
  def name    = column[String]("SUP_NAME")  
  def street  = column[String]("STREET")  
  def city    = column[String]("CITY")  
  def state   = column[String]("STATE")  
  def zip     = column[String]("ZIP")  
  
  def * = id ~ name ~ street ~ city ~ state ~ zip  
  
  def nameConstraint = index("SUP_NAME_IDX", name, true)  
}
```

# Table Definitions

```
val Coffees = new Table[(String, Int, Double,  
    Int, Int)]("COFFEES") {
```

```
  def name = column[String]("COF_NAME")
```

```
  def supID = column[Int]("SUP_ID")
```

```
  def price = column[Double]("PRICE")
```

```
  def sales = column[Int]("SALES")
```

```
  def total = column[Int]("TOTAL")
```

```
  def * = name ~ supID ~ price ~ sales ~ total
```

```
  def supplier = foreignKey("SUP_FK", supID, Suppliers)(_id)
```

```
  def pk = primaryKey("COF_NAME_PK", name)
```

```
}
```

```
(Suppliers.ddl ++ Coffees.ddl).create
```

# Databases & Sessions

```
import org.scalaquery.session._  
import org.scalaquery.session.Database.threadLocalSession
```

```
val db = Database.forURL("jdbc:h2:mem:test1",  
                        driver = "org.h2.Driver")
```

- forName
- forDataSource

```
db withSession { s: Session =>  
  doSomethingWithSession(s)  
}
```

withTransaction

# A DAO Pattern

```
class DAO(driver: ExtendedProfile, db: Database) {  
  import driver.Implicit._  
  
  val Props = new Table[(String, String)]("properties") {  
    def key = column[String]("key", 0.PrimaryKey)  
    def value = column[String]("value")  
    def * = key ~ value  
  }  
  
  def insert(k: String, v: String) = db withSession  
    Props.insert(k, v)  
  
  def get(k: String) = db withSession  
    ( for(p <- Props if p.key === k)  
      yield p.value ).firstOption  
}
```

# Inner Joins & Abstractions

```
for {  
  c <- coffees if c.price < 9.0  
  s <- suppliers if s.id == c.supID  
} yield (c.name, s.name)
```

Scala Collections

# Inner Joins & Abstractions

```
for {  
  c <- Coffees if c.price < 9.0  
  s <- Suppliers if s.id === c.supID  
} yield c.name ~ s.name
```

ScalaQuery

```
for {  
  c <- Coffees.cheaperThan(9.0)  
  s <- c.supplier  
} yield c.name ~ s.name
```

```
val Coffees = new Table ... {  
  def supplier = Suppliers.where(_.id === supID)  
  def cheaperThan(d: Double) = this.where(_.price < d)  
}
```

# Data Types

- Basic Types

- Byte, Int, Long

0

- String

""

- Boolean

false

- Date, Time, Timestamp

1970-1-1 00:00:00

- Float, Double

0.0

- Blob, Clob, Array[Byte]

null, null, []

- Option[T] for all basic types T

None

- NULL from database mapped to a default value

# NULL

- Three-Valued Logic (3VL) in SQL

$a \oplus b \rightarrow \text{NULL}$

*if  $a = \text{NULL}$  or  $b = \text{NULL}$*

- Even for „=“

$a = \text{NULL} \rightarrow \text{NULL}$

$\text{NULL} = a \rightarrow \text{NULL}$

$a \text{ IS NULL} \rightarrow \text{TRUE or FALSE}$

# NULL

- In Scala, we prefer to make nullability explicit with **Option** types
- Computations on DB values support 3VL:

```
Column[      A ] ⊕ Column[      B ] → Column[      [C]]
Column[Option[A]] ⊕ Column[      B ] → Column[Option[C]]
Column[      A ] ⊕ Column[Option[B]] → Column[Option[C]]
Column[Option[A]] ⊕ Column[Option[B]] → Column[Option[C]]
```

# Using Custom Data Types


```
object Values extends Enumeration {  
  val a, b, c = Value  
}
```

```
implicit val valueTypeMapper =  
  MappedTypeMapper.base[Values.Value, Int](_.id, Values(_))
```

```
val MyTable = new Table[Values.Value]("MYTABLE") {  
  def a = column[Values.Value]("A")  
  def * = a  
}
```

```
MyTable.ddl.create  
MyTable.insertAll(Values.a, Values.c)
```

```
val q = MyTable.map(t => t.a ~ t.a.asColumnOf[Int])  
q.foreach(println)
```



(a, 0)  
(c, 2)

# Mapped Entities

```
case class Coffee(name: String, supID: Int, price: Double)
```

```
val Coffees = new Table[(String, Int, Double)]("COFFEES") {  
  def name = column[String]("COF_NAME", 0.PrimaryKey)  
  def supID = column[Int]("SUP_ID")  
  def price = column[Double]("PRICE")  
  def * = name ~ supID ~ price <> (Coffee, Coffee.unapply _)  
}
```

```
Coffees.insertAll(  
  Coffee("Colombian", 101, 7.99),  
  Coffee("French_Roast", 49, 8.99)  
)
```

```
val q = for(c <- Coffees if c.supID === 101) yield c  
q.foreach(println)
```



```
Coffee(Colombian,101,7.99)
```

# Aggregating, Sorting and Paging

```
val q = for {  
  c <- Coffees  
  s <- c.supplier  
  _ <- Query groupBy s.id  
  _ <- Query orderBy c.name.count  
} yield s.id ~ s.name.min.get ~ c.name.count
```

Aggregation Methods

.min, .max, .avg, .sum, .count

```
val q2 = q.drop(20).take(10)
```

# Running a Query

- **.to[C]()** – create a Collection C of all results

e.g. `myQuery.to[List]()`  
`myQuery.to[Array]()`

- **.list** – Shortcut for `.to[List]()`

- **.first, .firstOption** – get the first result

- **.foreach** – execute a function for each result

```
for(r <- myQuery) ...
```

# Debugging

```
val q = for {  
  c <- Coffees if c.supID === 101  
} yield c.name ~ c.price
```

```
q.dump("q: ")
```

```
SELECT "t1"."COF_NAME","t1"."PRICE"  
FROM "COFFEES" "t1"  
WHERE ("t1"."SUP_ID"=101)
```

```
println(q.selectStatement)
```

```
q: Query  
select: Projection2  
0: NamedColumn COF_NAME  
  table: <t1> AbstractTable.Alias  
    0: <t2> Table COFFEES  
    1: NamedColumn PRICE  
      table: <t1> ...  
where: Is(NamedColumn SUP_ID,ConstColumn[Int] 101)  
0: NamedColumn SUP_ID  
  table: <t1> ...  
1: ConstColumn[Int] 101
```

# Bind Variables

```
def coffeesForSupplier(supID: Int) = for {  
  c <- Coffees if c.supID == supID.bind  
} yield c.name
```

coffeesForSupplier(42).list

Query

```
select: NamedColumn COF_NAME  
table: <t1> AbstractTable.Alias  
0: <t2> Table COFFEES  
where: Is(NamedColumn SUP_ID, Bind Column[Int] 42)  
0: NamedColumn SUP_ID  
table: <t1> ...  
1: Bind Column[Int] 42
```

```
SELECT "t1"."COF_NAME" FROM "COFFEES" "t1"  
WHERE ("t1"."SUP_ID" =?)
```

# Query Templates

```
val coffeesForSupplier = for {  
  supID <- Parameters[Int]  
  c <- Coffees if c.supID === supID  
} yield c.name
```

```
coffeesForSupplier(42).list
```

Query

**select:** NamedColumn COF\_NAME

**table:** <t1> AbstractTable.Alias

0: <t2> Table COFFEES

**where:** Is(NamedColumn SUP\_ID,ParameterColumn[Int])

0: NamedColumn SUP\_ID

**table:** <t1> ...

1: ParameterColumn[Int]

```
SELECT "t1"."COF_NAME" FROM "COFFEES" "t1"  
WHERE ("t1"."SUP_ID"=?)
```

# Insert, Delete, Update

```
class Coffees(n: String)
  extends Table[(String, Int, Double)](n) {
  def name = column[String]("COF_NAME")
  def supID = column[Int]("SUP_ID")
  def price = column[Double]("PRICE")
  def * = name ~ supID ~ price
}
```

```
val Coffees1 = new Coffees("COFFEES_1")
```

```
val Coffees1.ddl ++ Coffees2.ddl).create
```

```
INSERT INTO "COFFEES1" ("COF_NAME","SUP_ID","PRICE") VALUES (?,?,?)
```

```
(Coffees1.ddl ++ Coffees2.ddl).create
```

```
Coffees1.insertAll(
  ("Colombian",      101, 7.99),
  ("French_Roast",   49, 8.99),
  ("Espresso",      150, 9.99)
)
```

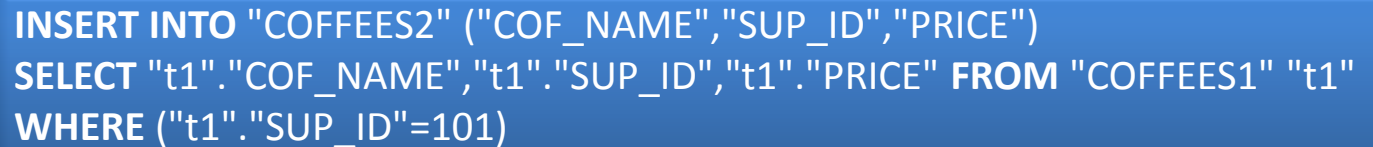
```
println(Coffees1.insertStatement)
```

# Insert, Delete, Update

```
val q = Coffees1.where(_.supID === 101)
```

```
Coffees2.insert(q)
```

```
println(Coffees2.insertStatementFor(q))
```



```
INSERT INTO "COFFEES2" ("COF_NAME","SUP_ID","PRICE")  
SELECT "t1"."COF_NAME","t1"."SUP_ID","t1"."PRICE" FROM "COFFEES1" "t1"  
WHERE ("t1"."SUP_ID"=101)
```

```
q.delete
```

```
println(q.deleteStatement)
```



```
DELETE FROM "COFFEES1" WHERE ("COFFEES1"."SUP_ID"=101)
```

# Insert, Delete, Update

```
val q2 = q.map(_.supID)  
q2.update(49)  
println(q2.updateStatement)
```



```
UPDATE "COFFEES1" SET "SUP_ID"=? WHERE ("COFFEES1"."SUP_ID"=101)
```

# More Query Language

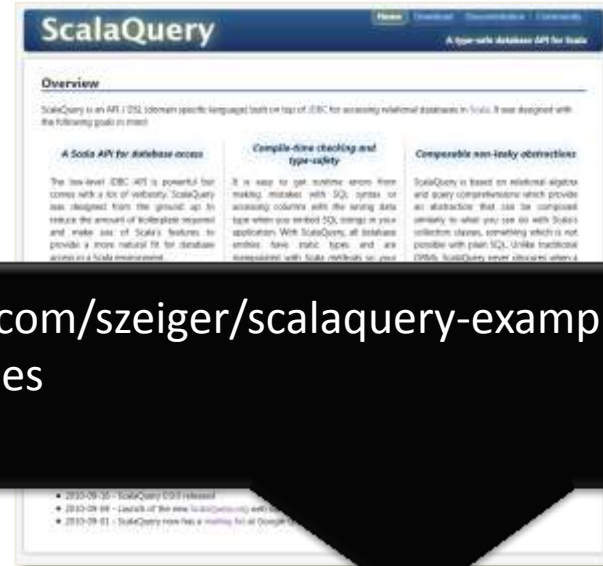
- Explicit Inner / Left / Right / Outer Joins
- Unions
- Conditionals (“CASE”)
- Sub-Queries
- Built-In Functions
- Custom Functions / Expressions

# More Features

- Mutating DB State On The Fly `MutatingInvoker.mutate`
- JDBC MetaData Abstractions `org.scalaquery.meta`
- Iteratees `org.scalaquery.iter`
- Sequences
- Dynamic Queries `org.scalaquery.simple`

# Getting Started

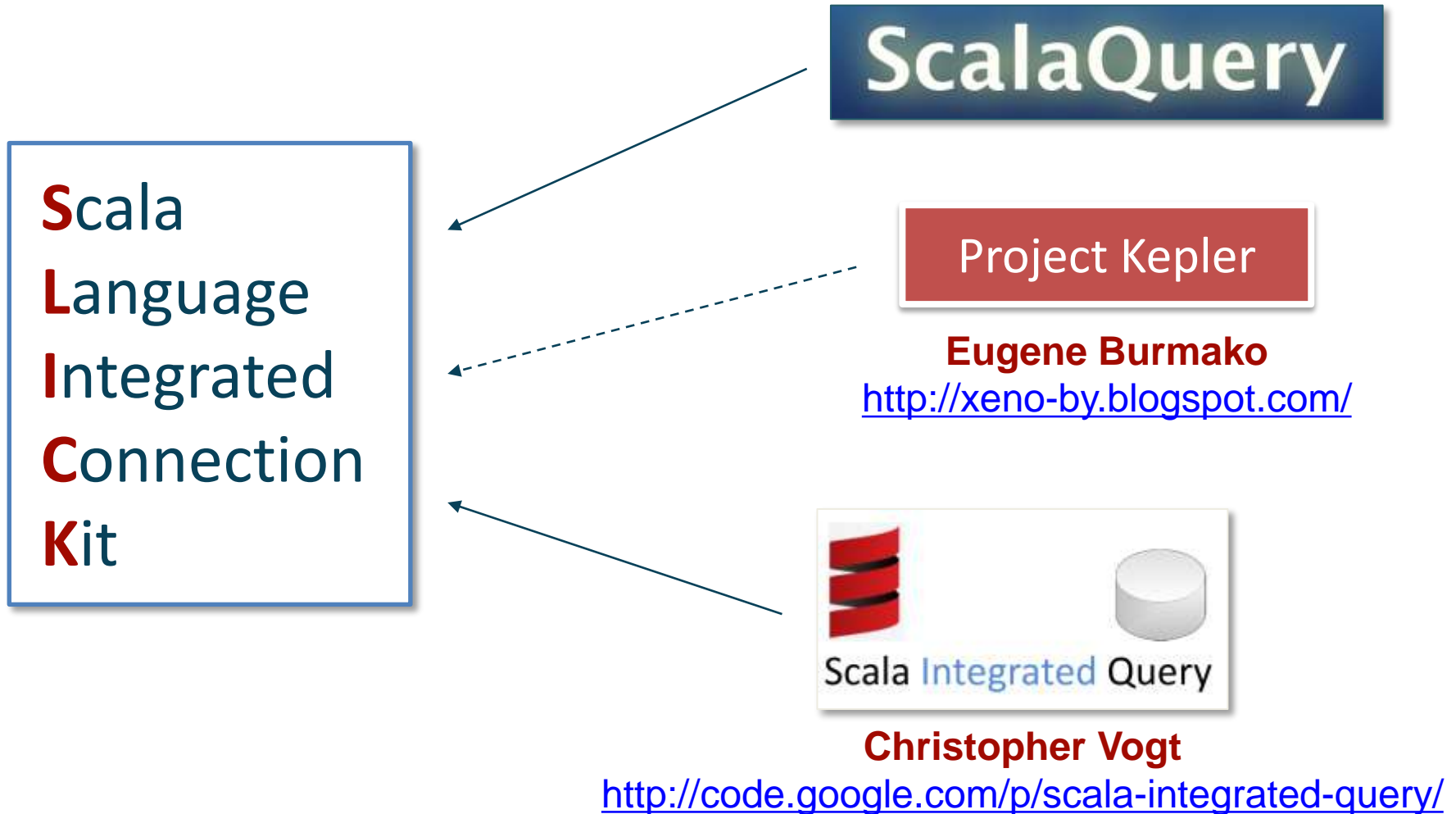
- <http://scalaquery.org>



- `git clone git://github.com/szeiger/scalaquery-examples.git`
- `cd scalaquery-examples`
- `sbt update run`

- Documented examples:  
<https://github.com/szeiger/scalaquery-examples>
- Test cases for all features:  
<https://github.com/szeiger/scala-query/tree/master/src/test/scala/org/scalaquery/test>

# The Future: SLICK



# Scala Language Integrated Connection Kit

- Support for **Relational + NoSQL** Databases
  - And other data sources
- Type Providers
- Optional LINQ-like API for **transparent** integration

# Scala Language Integrated Connection Kit

## Will be a part of the **Typesafe Stack**

- A 100% open source, integrated distribution offering Scala, Akka, sbt, and the Scala plugin for Eclipse
- Commercial support available via the Typesafe Subscription



# Outlook

## ScalaQuery 0.9

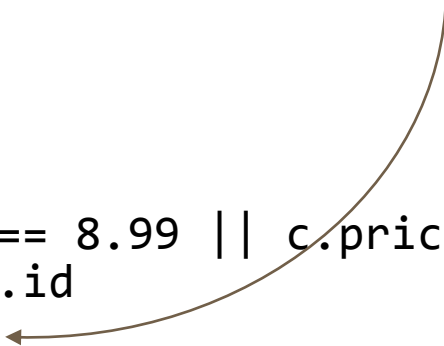
```
for {  
  c <- Coffees if c.price === 8.99 || c.price === 9.99  
  s <- c.supplier orderBy s.id  
} yield s.id ~ s.name ~ c.name ~ c.price ~ ...
```

# Outlook

ScalaQuery 0.9

ScalaQuery 0.10

```
for {  
  c <- Coffees if c.price === 8.99 || c.price === 9.99  
  s <- c.supplier orderBy s.id  
} yield ((s.id, s.name), c)
```



# Outlook

ScalaQuery 0.9

ScalaQuery 0.10

```
for {  
  s <- Suppliers  
  val cs = Coffees.filter(c => c.supID === s.id &&  
    (c.price === 8.99 || c.price === 9.99))  
} yield ((s.id, s.name), cs)
```

SIQ



# Outlook

ScalaQuery 0.9

ScalaQuery 0.10

```
for {  
  s <- Suppliers  
  val cs = Coffees.filter(c => c.supplier == s.id &&  
    (c.price == 8.99 || c.price == 9.99))  
} yield ((s.id, s.name), cs)
```

Plain Double instead of  
Column[Double]

LINQ-like API

SIQ

# Thank You!



<http://typesafe.com>

## Stefan Zeiger

Blog: <http://szeiger.de>

twitter



@StefanZeiger

@typesafe

# Bonus Slides

# Query Language Imports

```
import org.scalaquery.q1._
```

```
import org.scalaquery.q1.TypeMapper._
```

```
import org.scalaquery.q1.extended.H2Driver.Implicit._  
import org.scalaquery.q1.extended.{ExtendedTable => Table}
```

- basic.BasicDriver
- extended.AccessDriver
- extended.DerbyDriver
- **def column**[C : TypeMapper](n: String,  
options: ColumnOption[C, ProfileType]\*) = ...
- extended.MySQLDriver
- extended.PostgresDriver
- extended.SQLiteDriver
- extended.SQLServerDriver

# Column Operators

- **Common:** `.in(Query)`, `.notin(Query)`, `.count`, `.countDistinct`, `.isNull`, `.isNotNull`, `.asColumnOf`, `.asColumnType`
- **Comparison:** `=== (.is)`, `!== (.isNot)`, `<`, `<=`, `>`, `>=`, `.inSet`, `.inSetBind`, `.between`, `.ifNull`
- **Numeric:** `+`, `-`, `*`, `/`, `%`, `.abs`, `.ceil`, `.floor`, `.sign`, `.toDegrees`, `.toRadians`
- **Boolean:** `&&`, `||`, `.unary_!`
- **String:** `.length`, `.like`, `++`, `.startsWith`, `.endsWith`, `.toUpperCase`, `.toLowerCase`, `.ltrim`, `.rtrim`, `.trim`

# Unions

Scala Collections

```
val l1 = coffees.filter(_.supID == 101)
val l2 = coffees.filter(_.supID == 150)
val l3 = l1 ++ l2
```

ScalaQuery

```
val q1 = Coffees.filter(_.supID === 101)
val q2 = Coffees.filter(_.supID === 150)
val q3 = q1 unionAll q2
```

# Explicit Inner Joins

| <b>name</b>     | <b>supID</b> |
|-----------------|--------------|
| Colombian       | 101          |
| Espresso        | 150          |
| Colombian_Decaf | 42           |

Coffees

| <b>id</b> | <b>name</b>     |
|-----------|-----------------|
| 101       | Acme, Inc.      |
| 49        | Superior Coffee |
| 150       | The High Ground |

Suppliers

```
for (  
  Join(c, s) <- Coffees innerJoin Suppliers  
                                on (c.supID === s.id)  
) yield c.name ~ s.name
```

```
(Colombian, Acme, Inc.)  
(Espresso, The High Ground)
```

# Left Outer Joins

| name            | supID |
|-----------------|-------|
| Colombian       | 101   |
| Espresso        | 150   |
| Colombian_Decaf | 42    |

Coffees

| id  | name            |
|-----|-----------------|
| 101 | Acme, Inc.      |
| 49  | Superior Coffee |
| 150 | The High Ground |

Suppliers

```
for (  
  Join(c, s) <- Coffees leftJoin Suppliers  
                                on (c.supID == s.id)  
) yield c.name ~ s.name
```

```
(Colombian,Acme, Inc.)  
(Espresso,The High Ground)  
(Colombian_Decaf,)
```

# Option Lifting

| name            | supID |
|-----------------|-------|
| Colombian       | 101   |
| Espresso        | 150   |
| Colombian_Decaf | 42    |

Coffees

| id  | name            |
|-----|-----------------|
| 101 | Acme, Inc.      |
| 49  | Superior Coffee |
| 150 | The High Ground |

Suppliers

```
for (
  Join(c, s) <- Coffees leftJoin Suppliers
                                on (_.supID === _.id)
) yield c.name.~ ~ s.name.~
```

```
(Some(Colombian),Some(Acme, Inc.))
(Some(Espresso),Some(The High Ground))
(Some(Colombian_Decaf),None)
```

# Right Outer Joins

| name            | supID |
|-----------------|-------|
| Colombian       | 101   |
| Espresso        | 150   |
| Colombian_Decaf | 42    |

Coffees

| id  | name            |
|-----|-----------------|
| 101 | Acme, Inc.      |
| 49  | Superior Coffee |
| 150 | The High Ground |

Suppliers

```
for (  
  Join(c, s) <- Coffees rightJoin Suppliers  
                                on (c.supID === s.id)  
) yield c.name.? ~ s.name.?
```

```
(Some(Colombian),Some(Acme, Inc.))  
(None,Some(Superior Coffee))  
(Some(Espresso),Some(The High Ground))
```

# Full Outer Joins

| name            | supID |
|-----------------|-------|
| Colombian       | 101   |
| Espresso        | 150   |
| Colombian_Decaf | 42    |

Coffees

| id  | name            |
|-----|-----------------|
| 101 | Acme, Inc.      |
| 49  | Superior Coffee |
| 150 | The High Ground |

Suppliers

```
for (  
  Join(c, s) <- Coffees outerJoin Suppliers  
                                on (c.supID === s.id)  
) yield c.name.? ~ s.name.?
```

```
(Some(Colombian),Some(Acme, Inc.))  
(None,Some(Superior Coffee))  
(Some(Espresso),Some(The High Ground))  
(Some(Colombian_Decaf),None)
```

# Case

```
for {  
  c <- Coffees  
} yield (Case when c.price < 8.0 then "cheap"  
          when c.price < 9.0 then "medium"  
          otherwise "expensive") ~ c.name
```

- If...then...else for Queries
- Return type is automatically lifted to **Option** if there is no **otherwise** clause

# Sub-Queries

```
for {  
  c <- Coffees  
  s <- c.supplier  
  _ <- Query groupBy s.id orderBy s.id  
} yield s.name.min.get ~ c.price.min.get
```

# Sub-Queries

```
for {  
  c <- Coffees  
  s <- c.supplier  
  val lowestPriceForSupplier = (for {  
    c2 <- Coffees  
    s2 <- c2.supplier if s2.id === s.id  
  } yield c2.price.min).asColumn  
  _ <- Query if c.price === lowestPriceForSupplier  
  _ <- Query orderBy s.id  
} yield s.name ~ c.price
```

- Can also be used in **yield**
- Use directly (no `.asColumn`) with `.in` and `.notIn`
- `.exists`, `.count`

# Static Queries

```
import org.scalaquery.simple._
import org.scalaquery.simple.StaticQuery._

def allCoffees = queryNA[String](
  "select cof_name from coffees").list

def supplierNameForCoffee(name: String) =
  query[String, String]("""
    select s.sup_name from suppliers s, coffees c
    where c.cof_name = ? and c.sup_id = s.sup_id
  """).firstOption(name)

def coffeesInPriceRange(min: Double, max: Double) =
  query[(Double, Double), (String, Int, Double)]("""
    select cof_name, sup_id, price from coffees
    where price >= ? and price <= ?
  """).list(min, max)
```

# Static Queries

```
import org.scalaquery.simple._  
import org.scalaquery.simple.StaticQuery._
```

```
case class Coffee(  
  name: String, supID: Int, price: Double)
```

```
implicit val getCoffeeResult =  
  GetResult(r => Coffee(r<<, r<<, r<<))
```

[P : SetParameter,

R : GetResult]

```
def coffeesInPriceRange(min: Double, max: Double) =  
  query[(Double, Double), Coffee](  
    """  
    select cof_name, sup_id, price from coffees  
    where price >= ? and price <= ?  
    """).list(min, max)
```