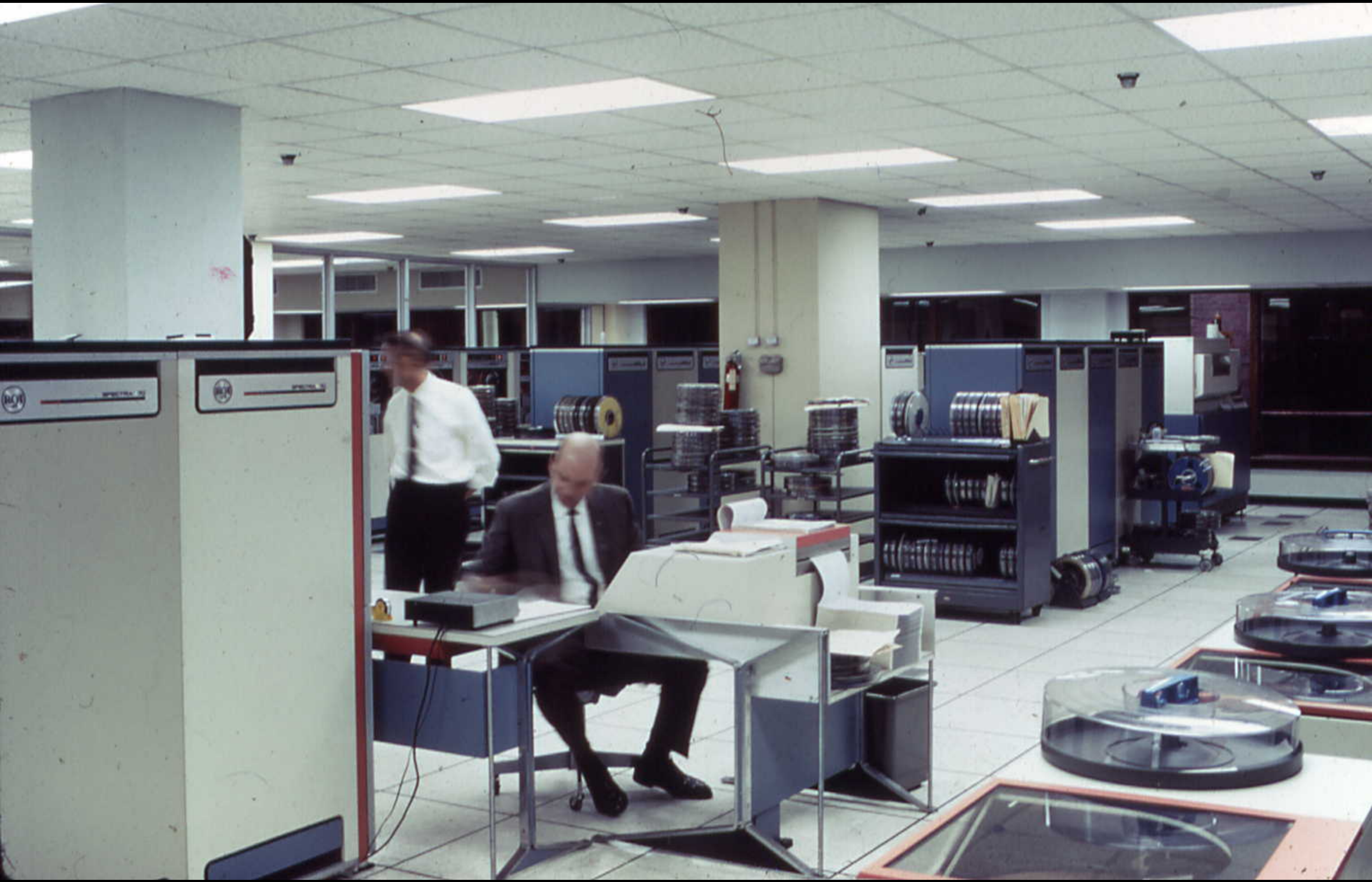
jax 2014

Business Technology Days

BIG DATA CON

Stefan Zeiger, Typesafe

Datenbankzugriff mit Slick



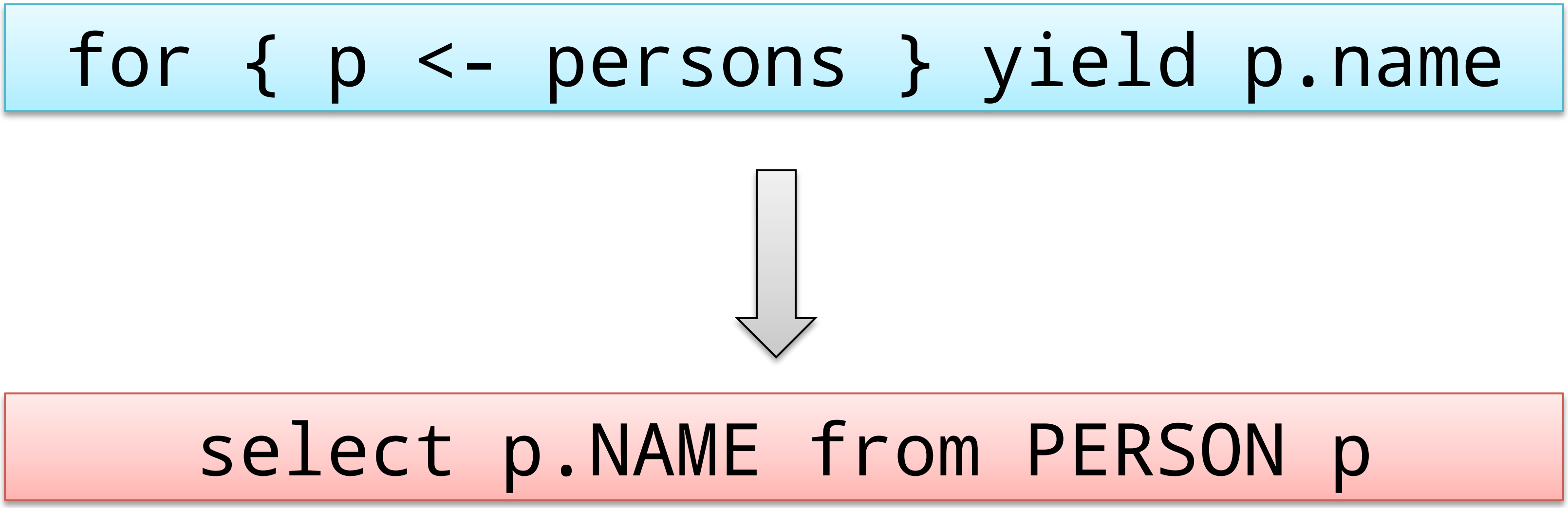
http://toto.lib.unca.edu/findingaids/photo/national_climatic_data_center/NCDC_interior.htm

NOAA's National Climatic Data Center is the source of this image and it is used by permission

e1r4p14
e1r3p9
e1r3p11
e1r2p10
e1r2p12

Datenbank-Queries in Scala

- Statt SQL, JPQL, Criteria API, etc.



```
for { p <- persons } yield p.name
```

```
select p.NAME from PERSON p
```

```
(for {
      p <- persons.filter(_.age < 20) ++
           persons.filter(_.age >= 50)
           if p.name.startsWith("A")
} yield p).groupBy(_.age).map { case (age, ps) =>
  (age, ps.length)
}
```

```
select x2.x3, count(1) from (
  select * from (
    select x4."NAME" as x5, x4."AGE" as x3
      from "PERSON" x4 where x4."AGE" < 20
    union all select x6."NAME" as x5, x6."AGE" as x3
      from "PERSON" x6 where x6."AGE" >= 50
    ) x7 where x7.x5 like 'A%' escape '^'
  ) x2
group by x2.x3
```

Slick

Scala Language Integrated Connection Kit

- Scala-Bibliothek für Datenbankzugriff
- Nachfolger von ScalaQuery
- Von Typesafe und der EPFL entwickelt
- Open Source

Funktional-Relationales Mapping

- Statt ORM (Objektrelationales Mapping)
- Baut auf dem relationalen Modell auf
- Vermeidet "Impedance Mismatch"

```
class Suppliers ... extends
    Table[(Int, String, String)](... "SUPPLIERS")
```

```
sup.filter(_.id < 2) ++ sup.filter(_.id > 5)
```

Funktional-Relationales Mapping

- Über Queries abstrahieren – wie bei Scala-Collections

```
def f(id1: Int, id2: Int) =
  sup.filter(_.id < id1) ++ sup.filter(_.id > id2)

val q = f(2, 5).map(_.name)
```

Funktional-Relationales Mapping

- Volle Kontrolle über Statement-Ausführung
- Zustandslos

```
val result = q.run(session)
```

Unterstützte Datenbanken

- PostgreSQL
- MySQL
- H2
- Hsqldb
- Derby / JavaDB
- SQLite
- Access

Kommerzielles Zusatzpaket *Slick Extensions* (mit Support durch Typesafe):

- Oracle
- DB/2
- SQL Server

Aufbau

Komponenten

- **Lifted Embedding** (Query-API)
- Direct Embedding (Query-API)
- **Plain SQL** (Query-API)
- **Session-Management**
- Schema-Modell
- **Code-Generator**

Session-Management

Einheitliches Session-Management

```
import scala.slick.driver.H2Driver.simple._

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

- forName
- forDataSource

```
db withSession { implicit session =>
  doSomethingWithSession
}
```

withTransaction

Treiber-unabhängiger Code

```
class MyDAO(driver: JdbcProfile) {
  import driver.simple._

  ...
}
```

BasicProfile

⮤ RelationalProfile

⮤ SqlProfile

⮤ JdbcProfile

⇨ MultiDBExample and MultiDBCakeExample in https://github.com/slick/slick-examples

Code-Generator

Beispiel: sbt-Task

```
lazy val slick = TaskKey[Seq[File]]("gen-tables")
lazy val slickCodeGenTask =
    (sourceManaged, dependencyClasspath in Compile,
    runner in Compile, streams) map { (dir, cp, r, s) =>
  val outputDir = (dir / "slick").getPath
  val url = "jdbc:h2:~/test"
  val jdbcDriver = "org.h2.Driver"
  val slickDriver = "scala.slick.driver.H2Driver"
  val pkg = "demo"
  toError(r.run(
    "scala.slick.model.codegen.SourceCodeGenerator",
    cp.files, Array(slickDriver, jdbcDriver, url, outputDir,
    pkg), s.log))
    Seq(file(outputDir + "/demo/Tables.scala"))
}
```

"Lifted Embedding"-API

Tabellen-Definition

```
class Suppliers(tag: Tag) extends
    Table[(Int, String, String)](tag, "SUPPLIERS") {
  def id = column[Int]("SUP_ID",
                       O.PrimaryKey, O.AutoInc)
  def name = column[String]("SUP_NAME")
  def city = column[String]("CITY")
  def * = (id, name, city)
}

val suppliers = TableQuery[Suppliers]
```

Eigene Typen für Zeilen

```
case class Supplier(id: Int, name: String,
  city: String)

class Suppliers(tag: Tag) extends
    Table[ Supplier            ](tag, "SUPPLIERS") {
  def id = column[Int]("SUP_ID",
                       O.PrimaryKey, O.AutoInc)
  def name = column[String]("SUP_NAME")
  def city = column[String]("CITY")
  def * = (id, name, city) <>
    (Supplier.tupled, Supplier.unapply)
}

val suppliers = TableQuery[Suppliers]
```

Eigene Typen für Spalten

```
class SupplierId(val value: Int) extends AnyVal

case class Supplier(id: SupplierId, name: String,
  city: String)

implicit val supplierIdType = MappedColumnType.base
  [SupplierId, Int](_.value, new SupplierId(_))

class Suppliers(tag: Tag) extends
    Table[Supplier](tag, "SUPPLIERS") {
  def id = column[SupplierId]("SUP_ID", ...)
  ...
}
```

Eigene Typen für Spalten

```
class SupplierId(val value: Int) extends MappedTo[Int]

case class Supplier(id: SupplierId, name: String,
  city: String)
```

```
class Suppliers(tag: Tag) extends
    Table[Supplier](tag, "SUPPLIERS") {
  def id = column[SupplierId]("SUP_ID", ...)
  ...
}
```

Fremdschlüssel

```
class Coffees(tag: Tag) extends Table[
    (String, SupplierId, Double)](tag, "COFFEES") {
  def name = column[String]("NAME", O.PrimaryKey)
  def supID = column[SupplierId]("SUP_ID")
  def price = column[Double]("PRICE")
  def * = (name, supID, price)
  def supplier =
    foreignKey("SUP_FK", supID, suppliers)(_.id)
}

val coffees = TableQuery[Coffees]
```

Tabellen erzeugen und befüllen

```
val suppliers = new ArrayBuffer[Supplier]
val coffees = new ArrayBuffer[(String, SupplierId, Double)]
```

```
suppliers += Supplier(si1, "Acme, Inc.", "Groundsville")
suppliers += Supplier(si2, "Superior Coffee", "Mendocino")
suppliers += Supplier(si3, "The High Ground", "Meadows")

coffees ++= Seq(
  ("Colombian",          si1, 7.99),
  ("French_Roast",       si2, 8.99),
  ("Espresso",           si3, 9.99),
  ("Colombian_Decaf",    si1, 8.99),
  ("French_Roast_Decaf", si2, 9.99)
)
```

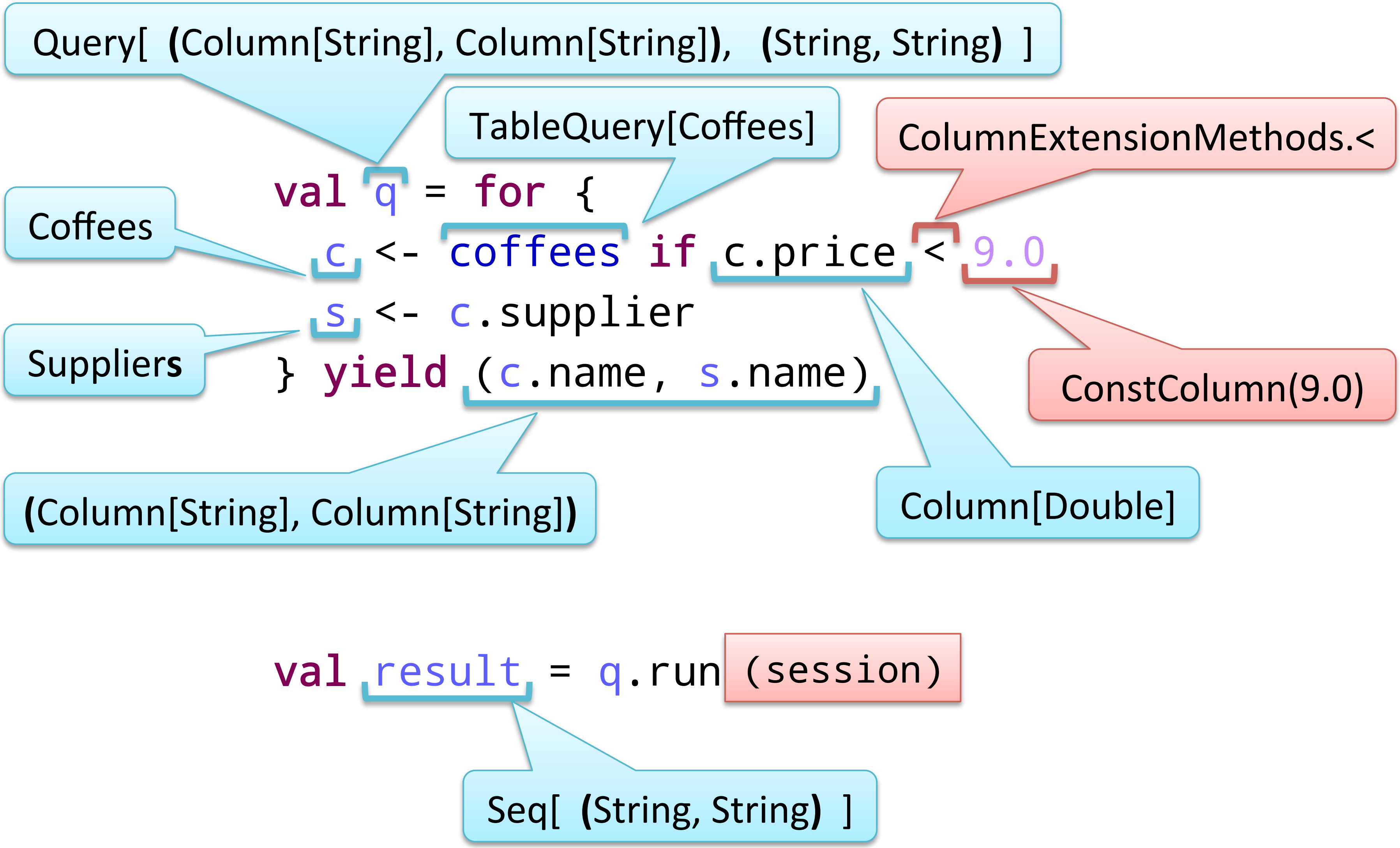
Auto-Generated Keys

```
val ins = suppliers.map(s => (s.name, s.city))
  returning suppliers.map(_.id)

val si1 = ins += ("Acme, Inc.", "Groundsville")
val si2 = ins += ("Superior Coffee", "Mendocino")
val si3 = ins += ("The High Ground", "Meadows")

coffees ++= Seq(
  ("Colombian",          si1, 7.99),
  ("French_Roast",       si2, 8.99),
  ("Espresso",           si3, 9.99),
  ("Colombian_Decaf",    si1, 8.99),
  ("French_Roast_Decaf", si2, 9.99)
)
```

Queries



```
val q = for {
  c <- coffees if c.price < 9.0
  s <- c.supplier
} yield (c.name, s.name)

val result = q.run (session)
```

Query[**(**Column[String], Column[String]**)**, **(**String, String**)**]

TableQuery[Coffees]

ColumnExtensionMethods.<

Coffees

Suppliers

ConstColumn(9.0)

Column[Double]

(Column[String], Column[String]**)**

Seq[**(**String, String**)**]

Mehr Queries

```
val q1 = suppliers.filter(_.id === 42)
val q2 = suppliers.filter(_.id =!= 42)

val q4 = (for {
  c <- coffees
  s <- c.supplier
} yield (c, s)).groupBy(_._2.id).map { case (_, q) =>
  (q.map(_._2.name).min.get, q.length)
}
```

Column[**Option[String]**]

"Plain SQL"-API

JDBC

```scala
def personsMatching(pattern: String)(conn: Connection) = {
  val st = conn.prepareStatement(
    "select id, name from person 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()
}
```

Slick

```
def personsMatching(pattern: String)(implicit s: Session) =
  sql"select id, name from person where name like $pattern"
    .as[(Int, String)].list
```

Ausblick

Neu in Slick 2.0 (Januar 2014)

- Verbessertes API
- Code-Generator
- Query-Scheduling (experimental)
- Neue Treiber- und Backend-Architektur

Slick 2.1: Juni 2014 (RC)

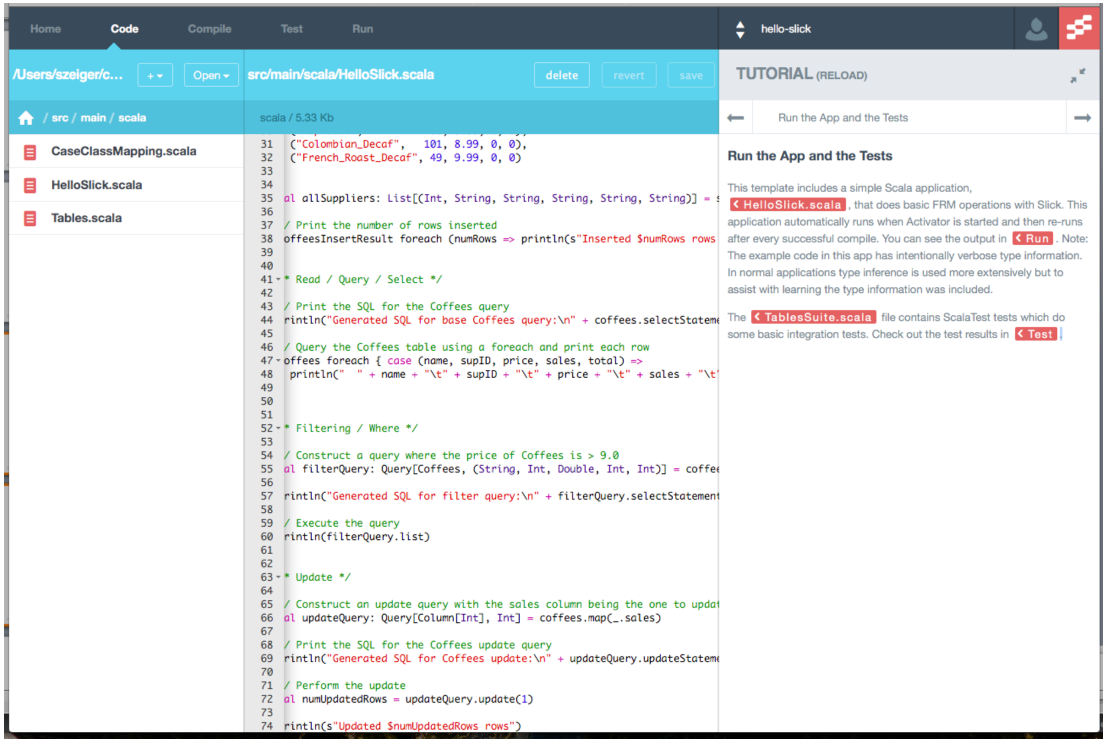
- User Experience (API, Dokumentation)
- Insert-or-Update
- Verbesserter Code-Generator
- Vorcompilierte Queries mit *.take* und *.drop*
- Effizienteres Lesen aus ResultSets
- OSGi-Unterstützung

Slick 2.2: Dezember 2014 (RC)

- Non-blocking API
  - Futures
  - Reactive Streams
- Erweiterte Unterstützung für *Option*-Typen
  - Einfacheres Handling von Outer Joins
- Statische Validierung und Typ-Inferenz von "Plain SQL"-Queries (GSoC-Projekt)

Einfach loslegen mit Activator:

http://typesafe.com/activator



CaseClassMapping.scala
HelloSlick.scala
Tables.scala

```
 ("Colombian_Decaf",   101, 8.99, 0, 0),
 ("French_Roast_Decaf", 49, 9.99, 0, 0)


al allSuppliers: List[(Int, String, String, String, String, String)] = s

/ Print the number of rows inserted
offeesInsertResult foreach (numRows => println(s"Inserted $numRows rows


* Read / Query / Select */

/ Print the SQL for the Coffees query
rintln("Generated SQL for base Coffees query:\n" + coffees.selectStateme

/ Query the Coffees table using a foreach and print each row
offees foreach { case (name, supID, price, sales, total) =>
 println("  " + name + "\t" + supID + "\t" + price + "\t" + sales + "\t'



* Filtering / Where */

/ Construct a query where the price of Coffees is > 9.0
al filterQuery: Query[Coffees, (String, Int, Double, Int, Int)] = coffee

rintln("Generated SQL for filter query:\n" + filterQuery.selectStatement

/ Execute the query
rintln(filterQuery.list)


* Update */

/ Construct an update query with the sales column being the one to updat
al updateQuery: Query[Column[Int], Int] = coffees.map(_.sales)

/ Print the SQL for the Coffees update query
rintln("Generated SQL for Coffees update:\n" + updateQuery.updateStateme

/ Perform the update
al numUpdatedRows = updateQuery.update(1)

rintln(s"Updated $numUpdatedRows rows")
```

Run the App and the Tests

This template includes a simple Scala application, HelloSlick.scala, that does basic FRM operations with Slick. This application automatically runs when Activator is started and then re-runs after every successful compile. You can see the output in Run. Note: The example code in this app has intentionally verbose type information. In normal applications type inference is used more extensively but to assist with learning the type information was included.

The TablesSuite.scala file contains ScalaTest tests which do some basic integration tests. Check out the test results in Test.

slick.typesafe.com

@StefanZeiger

Typesafe