

---

# *quill*, DOTTY, AND THE AWESOME POWER OF 'inline'

Alexander Ioffe

@deusaquilus   

[https://github.com/deusaquilus/dotty\\_test/tree/fs\\_2020\\_talk/](https://github.com/deusaquilus/dotty_test/tree/fs_2020_talk/)

# — What is Quill?

```
case class Person(name: String, age: Int)

val q = quote {
    query[Person].filter(p => p.name == "Joe")
}
```

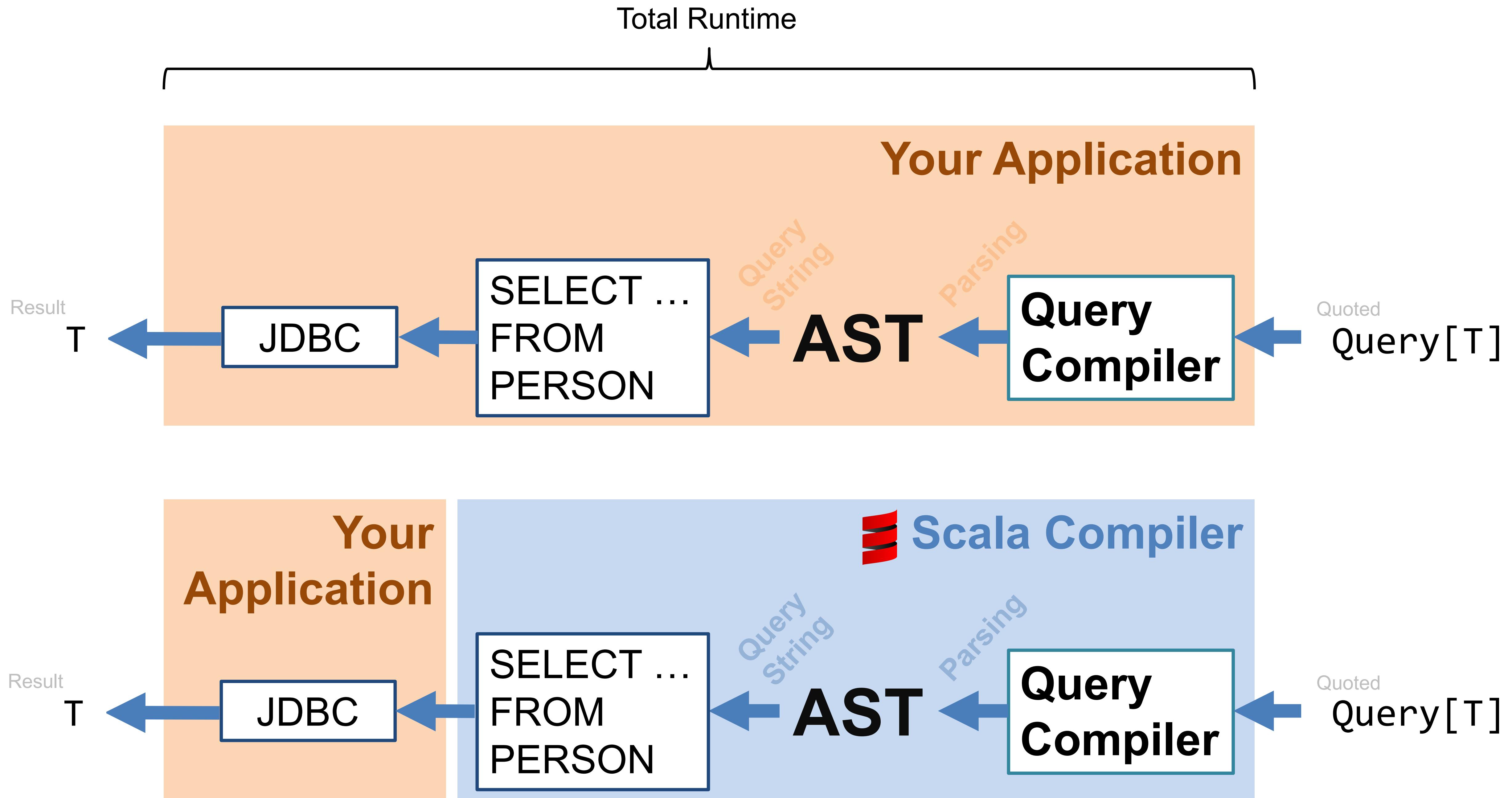
```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
SELECT p.name, p.age FROM Person p WHERE p.name = 'Joe'
```

# — What is Quill...

```
case class Person(name: String, age: Int)

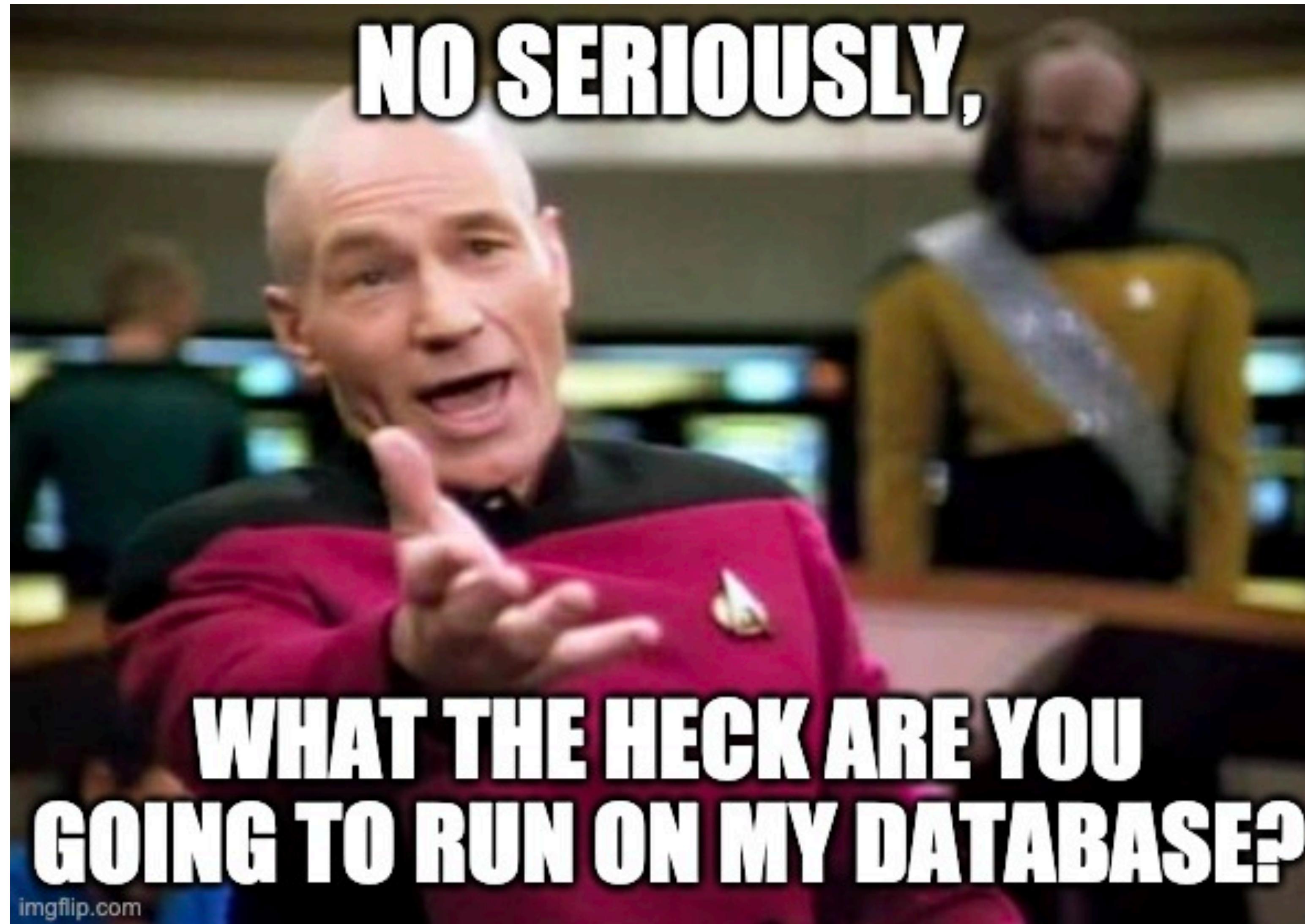
val q = quote {
  select[Person].filter(p => p.name == "Joe")
}
```

```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
SELECT p.name, p.age FROM Person p WHERE p.name = 'Joe'
```



# Then Your DBA Asks...

In a fit of Desperation at 3:00 AM



# Bring on the Complexity...

```
val q = quote {
  for {
    o1 <- query[Country]
    c1 <- query[Company].join(c1 => c1.countryId == o1.id)
    p1 <- query[Person].join(p1 => p1.companyId == c1.id)
    af <- query[ClientAffiliation].leftJoin(af => af.of == p1.id)
    p2 <- query[Person].leftJoin(p2 => af.map(_.host).exists(v => v == p2.id))
  } yield (o1, c1, p2, af, p2)
}
run(q)
```

```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
SELECT o1.id, o1.name, o1.population, c1.id, c1.name, c1.countryId,
p1.id, p1.name, p1.age, p1.companyId, af.host, af.of, p2.id, p2.name,
p2.age, p2.companyId FROM Country o1
INNER JOIN Company c1 ON c1.countryId = o1.id
INNER JOIN Person p1 ON p1.companyId = c1.id
LEFT JOIN ClientAffiliation af ON af.of = p1.id
LEFT JOIN Person p2 ON af.host = p2.id
```

# Limitation - Must Lift Runtime Values

```
case class Person(name: String, age: Int)

val q = quote {
    query[Person].filter(p => p.name == lift(runtimeVar))
}
```

```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
SELECT p.name, p.age FROM Person p WHERE p.name = ?
```

# Limitation - Runtime Switches

```
case class Person(name: String, age: Int)

val q =
  if (someRuntimeCondition)
    quote { query[Person].filter(p => p.name == "Joe") }
  else
    quote { query[Person] }
```

```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
Dynamic Query
```

# Limitation - Dynamic Filters

```
case class Person(name: String, age: Int)

val filters = Map("name" -> "Joe", "age" -> "22")
val q = quote {
    query[Person].filter(p =>
        col(p, "name") == lift(filters("name")) AND
        col(p, "age") == lift(filters("age")))
    )
}
```

```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
Dynamic Query
```

# Limitation - Encapsulating Methods

```
case class Person(name: String, age: Int)

val q = quote {
    joes(query[Person])
}

def joes(q: Query[People]) =
    q.filter(p => p.name == "Joe")
```

```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
Dynamic Query
```

---

# This...

```
case class Person(name: String, age: Int)

val q: Quoted[Query[Person]] =
    quote { query[Person].filter(p => p.name == "Joe") }
```

```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
Dynamic Query
```

# ... Is Really This

```
case class Person(name: String, age: Int)

val q: AnyRef with Quoted[Query[Person]] { def quoted... def lifted... } =
  quote { query[Person].filter(p => p.name == "Joe") }
```

```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
Dynamic Query
```

# I CAN (EASILY) GET COMPILE-TIME QUERIES

... BUT GIVE UP MOST LANGUAGE FEATURES

```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
Dynamic Query
```



## Before We Get Started...



# I was told there would be Macros

```
object Mac {  
    inline def enter(str: String): Unit = ${ enterImpl('str) }  
  
    def enterImpl(str: Expr[String])(using qctx: QuoteContext): Expr[Unit] = {  
        import qctx.tasty._  
        '{ () }  
    }  
}
```

# I was told there would be Macros

```
object Mac {  
    inline def enter(str: String): Unit = ${ enterImpl('str) }  
  
    def enterImpl(str: Expr[String])(using qctx: QuoteContext): Expr[Unit] = {  
        import qctx.tasty._  
        println(pprint(str.unseal))  
        '{ () }  
    }  
}
```

# I was told there would be Macros

```
object Mac {  
    inline def enter(str: String): Unit = ${ enterImpl('str) }  
  
    def enterImpl(str: Expr[String])(using qctx: QuoteContext): Expr[Unit] = {  
        import qctx.tasty._  
        println(pprint(str.unseal))  
        '{ () }  
    }  
}
```

# I was told there would be Macros

```
object Mac {  
    inline def enter(str: String): Unit = ${ enterImpl('str) }  
  
    def enterImpl(str: Expr[String])(using qctx: QuoteContext): Expr[Unit] = {  
        import qctx.tasty._  
        println(pprint(str.unseal))  
        '{ () }  
    }  
}
```

```
val greeting = "Hello"  
Mac.enter(greeting)
```

```
sbt:dotty-simple> compile  
[info] Compiling 1 Scala source to /Users...  
Inlined(Thicket(List()), List(), Ident(greeting))
```

# Inline All The Things!

```
object Mac {  
    inline def enter(inline str: String): Unit = ${ enterImpl('str) }  
  
    def enterImpl(str: Expr[String])(using qctx: QuoteContext): Expr[Unit] = {  
        import qctx.tasty._  
        println(pprint(str.unseal))  
        '{ () }  
    }  
}
```

---

```
inline def greeting = "Hello"  
Mac.enter(greeting)
```

```
val greeting = "Hello"  
PrintMac(greeting)
```

```
inline def greeting = "Hello"  
PrintMac(greeting)
```

```
[info] Compiling 1 Scala source to /Users...  
Inlined(  
  Thicket(List()),  
  List(),  
  Ident(greeting)  
)
```

```
[info] Compiling 1 Scala source to /Users...  
Inlined(  
  Thicket(List()),  
  List(),  
  Inlined(  
    Ident(greeting),  
    List(),  
    Literal(("Hello"))),  
  )  
)
```

```
inline def greeting = "Hello"
inline def suffix = " world"
inline def combo = greeting + suffix
PrintMac(combo)
```

```
Inlined(
  Thicket(List()), List(),
  Inlined(
    Ident(combo), List(),
    Typed(
      Apply(
        Select(
          Inlined(Ident(greeting), List(), Typed(Literal(("Hello")), TypeStuff())),
          +
          ),
          List(Inlined(Ident(suffix), List(), Typed(Literal((" world")), TypeStuff())))
        ),
        TypeStuff()
      )
    )
  )
)
```

Inlined(<originalArg>, <binds>, <expansion>)

NOTE: This Slide was skipped  
in the talk to save time.

```
inline def greeting = "Hello"
inline def suffix = " world"
inline def suffix2 = " today!"
inline def combo = greeting + suffix + suffix2
Mac.enter(combo)
```

```
Inlined(
  Thicket(List()),List(),
  Inlined(
    Ident(combo),List(),
    Typed(
      Apply(
        Select(
          Apply(
            Select(
              Inlined(Ident(greeting),List(),Typed(Literal(("Hello")),TypeStuff())),
              +
              ),
              List(Inlined(Ident(suffix),List(),Typed(Literal((" world")),TypeStuff())))
            ),
            +
            ),
            List(Inlined(Ident(suffix2),List(),Typed(Literal((" today!")),TypeStuff())))
          ),
          TypeStuff()
        )
      )
    )
  )
```

Inlined(<originalArg>, <binds>, <expansion>)

NOTE: This Slide was skipped  
in the talk to save time.

```
def enterImpl(str: Expr[String])(using qctx: QuoteContext): Expr[Unit] = {  
    import qctx.tasty._  
    println(pprint(str.unseal))  
    '{ () }  
}
```

---

```
def enterImpl(str: Expr[String])(using qctx: QuoteContext): Expr[Unit] = {  
    import qctx.tasty._  
    println(pprint(str.unseal.underlyingArgument))  
    '{ () }  
}
```

```
Typed(  
  Literal(("Hello")),  
  TypeStuff()  
)  
Typed(  
  Apply(  
    Select(  
      Apply(  
        Select(  
          Typed(Literal(("Hello")), TypeStuff()),  
          +  
        ),  
        List(Typed(Literal((" world")), TypeStuff()))  
      ),  
      +  
    ),  
    List(Typed(Literal((" today!")), TypeStuff()))  
  ),  
  TypeStuff()  
)
```

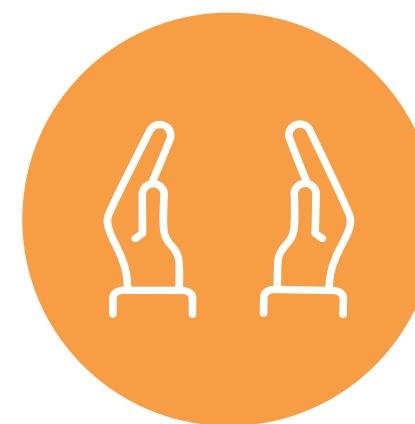
Inlined(<originalArg>, <binds>, <expansion>)

# Phase Consistency Principle

A.K.A Passing the Buck... in a good way!

$\forall \text{ pieceOfCode: Expr[T]}$

Splice it In



```
return '{ func($str) }
```

Throw it Out



```
val str: Expr[T] = ...  
'{ () }
```

Transform It

Then do one of the former...



```
val bar: Expr[Bar] = ...  
'{ change($foo) }
```

∴ Macros Will Always Be Hygienic

# Phase Consistency Principle

A.K.A Passing the Buck... in a good way!

```
inline def passThrough(inline str: String): String = ${ passThroughImpl('str) }
def passThroughImpl(str: Expr[String])(using qctx: QuoteContext): Expr[String] = {
  import qctx.tasty._
  println(pprint(str.unseal))
  str
}
```

```
class Space {
  class InnerSpace {
    inline def hello = Mac.passThrough("hello")
  }
  inline def world = Mac.passThrough(new InnerSpace().hello)
}
inline def today = Mac.passThrough(new Space().world + " today")
println(today)
```

This is not available

By the time this happens

# Phase Consistency Principle

A.K.A Passing the Buck... in a good way!

```
class Space {  
    class InnerSpace {  
        inline def hello = Mac.passThrough("hello")  
    }  
    inline def world = Mac.passThrough(new InnerSpace().hello + " world")  
}  
inline def today = Mac.passThrough(new Space().world + " today")  
println(today)
```

```
Inlined(Thicket(List()), List(), Apply(Select(  
    Inlined(Select(Apply(Select(New(Ident(Space)), <init>), List()), world), List(),  
        Inlined(Apply(Select(Ident(Mac), passThrough), List(Apply(Select(  
            Inlined(Select(Apply(Select(New(Ident(InnerSpace)), <init>), List()), hello), List(),  
                Inlined(Apply(Select(Ident(Mac), passThrough), List(Literal(("hello"))))), List(),  
                    Inlined(Thicket(List()), List(), Literal(("hello"))),  
                    ),  
                    ), + ), List(Literal(("world"))))  
                ),  
            List(),  
            Inlined(Thicket(List()), List(), Apply(Select(  
                Inlined(Select(Apply(Select(New(Ident(InnerSpace)), <init>), List()), hello), List(),  
                    Inlined(Apply(Select(Ident(Mac), passThrough), List(Literal(("hello"))))), List(),  
                        Inlined(Thicket(List()), List(), Literal(("hello"))),  
                        ), + ), List(Literal(("world"))))  
                    ),  
                    ), + ), List(Literal((" today")))  
    ))  
)}
```

Inlined(<originalArg>, <binds>, <expansion>)

# Phase Consistency Principle

A.K.A Passing the Buck... in a good way!

```
class Space {  
    class InnerSpace {  
        inline def hello = Mac.passThrough("hello")  
    }  
    inline def world = Mac.passThrough(new InnerSpace().hello + " world")  
}  
inline def today = Mac.passThrough(new Space().world + " today")  
println(today)
```

```
Inlined(Thicket(List()), List(), Apply(Select(  
    Inlined(Select(Apply(Select(New(Ident(Space)), <init>), List()), world), List(),  
        Inlined(Apply(Select(Ident(Mac), passThrough), List(Apply(Select(  
            Inlined(Select(Apply(Select(New(Ident(InnerSpace)), <init>), List()), hello), List(),  
                Inlined(Apply(Select(Ident(Mac), passThrough), List(Literal(("hello"))))), List(),  
                    Inlined(Thicket(List()), List(), Literal(("hello"))),  
                ),  
            ), + ), List(Literal(("world"))))  
        ),  
        List(),  
        Inlined(Thicket(List()), List(), Apply(Select(  
            Inlined(Select(Apply(Select(New(Ident(InnerSpace)), <init>), List()), hello), List(),  
                Inlined(Apply(Select(Ident(Mac), passThrough), List(Literal(("hello"))))), List(),  
                    Inlined(Thicket(List()), List(), Literal(("hello"))),  
                ), + ), List(Literal(("world"))))  
            ),  
        ), + ), List(Literal((" today")))  
    ))  
)
```

Inlined(<originalArg>, <binds>, <expansion>)



# Phase Consistency Principle

A.K.A Passing the Buck... in a good way!

```
class Space {  
    class InnerSpace {  
        inline def hello = Mac.passThrough("hello")  
    }  
    inline def world = Mac.passThrough(new InnerSpace().hello + " world")  
}  
inline def today = Mac.passThrough(new Space().world + " today")  
println(today)
```

```
Inlined(Thicket(List()), List(), Apply(Select(  
    Inlined(Select(Apply(Select(New(Ident(Space)), <init>), List()), world), List(),  
        Inlined(Apply(Select(Ident(Mac), passThrough), List(Apply(Select(  
            Inlined(Select(Apply(Select(New(Ident(InnerSpace)), <init>), List()), hello), List(),  
                Inlined(Apply(Select(Ident(Mac), passThrough), List(Literal(("hello"))))), List(),  
                    Inlined(Thicket(List()), List(), Literal(("hello"))),  
                ),  
            ), + ), List(Literal(("world"))))  
        ),  
        List(),  
        Inlined(Thicket(List()), List(), Apply(Select(  
            Inlined(Select(Apply(Select(New(Ident(InnerSpace)), <init>), List()), hello), List(),  
                Inlined(Apply(Select(Ident(Mac), passThrough), List(Literal(("hello"))))), List(),  
                    Inlined(Thicket(List()), List(), Literal(("hello"))),  
                ), + ), List(Literal(("world"))))  
            ),  
        ), + ), List(Literal((" today")))  
    ))  
)
```

Applying In

Expanding Out

Inlined(<originalArg>, <binds>, <expansion>)

# Phase Consistency Principle

A.K.A Passing the Buck... in a good way!

```
class Space {  
    class InnerSpace {  
        inline def hello = Mac.passThrough("hello")  
    }  
    inline def world = Mac.passThrough(new InnerSpace().hello + " world")  
}  
inline def today = Mac.passThrough(new Space().world + " today")  
println(today)
```

## Built-in Lineage

Applying In

Expanding Out

Inlined(<originalArg>, <binds>, <expansion>)

# Phase Consistency Principle

A.K.A Passing the Buck... in a good way!

```
class Space {  
    class InnerSpace {  
        inline def hello = Mac.passThrough("hello")  
    }  
    inline def world = Mac.passThrough(new InnerSpace().hello + " world")  
}  
inline def today = Mac.passThrough(new Space().world + " today")  
println(today)
```

Hygienic ↞ Built-in Lineage

# In Scala 3 Quill Quotes will be Inline

... and life will be awesome!

```
inline def p =  
query[Person]
```



```
inline def q = quote {  
query[Person]  
}
```

Inlined(EntityQuery[Person])

Inlined(Quoted(EntityQuery[Person] ...))

```
inline def p =  
  query[Person]
```



```
inline def q = quote {  
  query[Person]  
}
```

```
inline def p = Inlined(EntityQuery[Person])
```

Inlined(p ...)

```
inline def p =  
query[Person]
```

```
inline def q = quote {  
  p  
}
```



```
inline def p = Inlined(EntityQuery[Person])
```

**THE COMPILER  
DOES THIS!**

```
Inlined(Inlined(EntityQuery[Person]) ...)
```

```
inline def p =  
query[Person]
```

```
inline def q = quote {
```

```
    p  
}
```

```
inline def p = Inlin
```

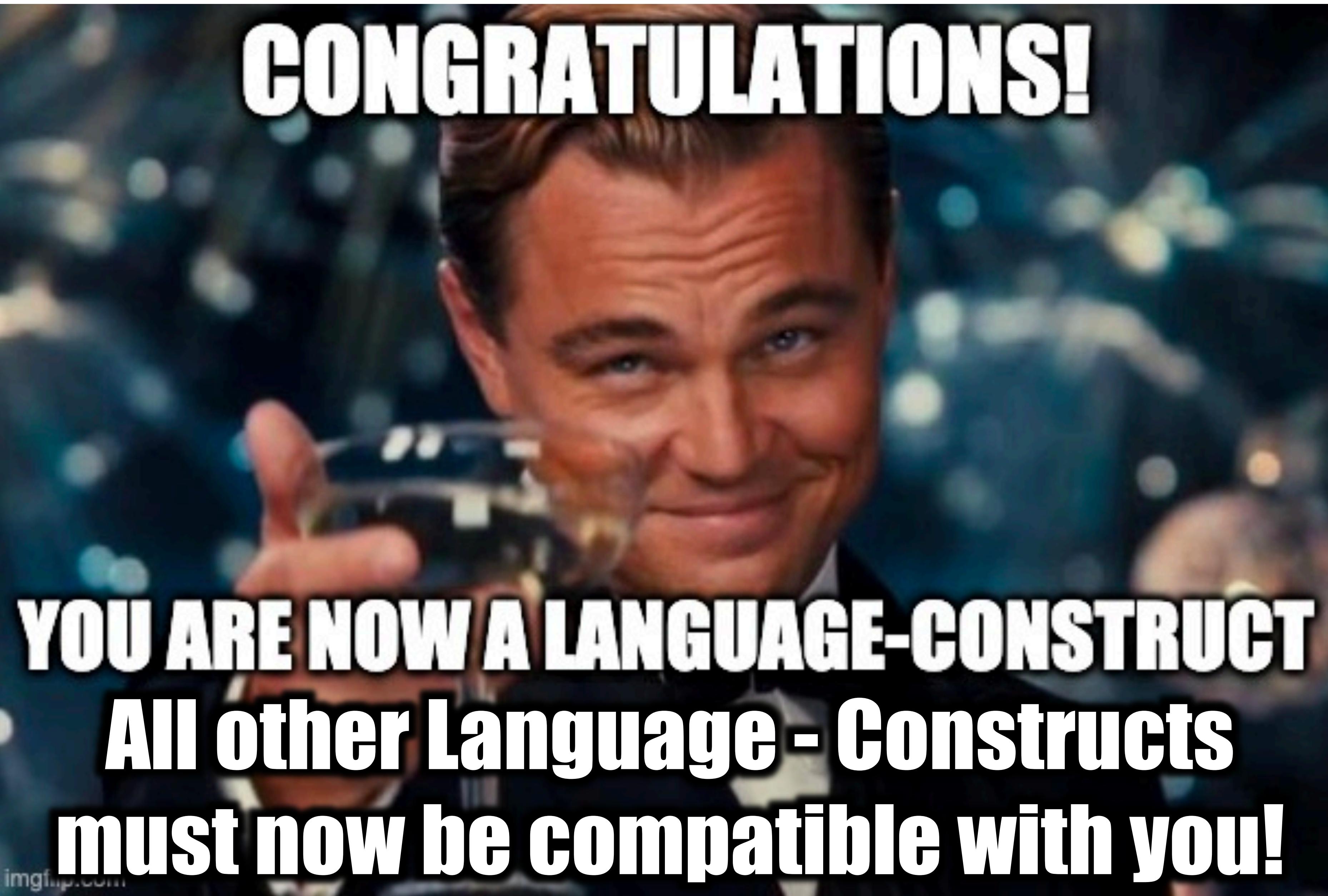
```
inline def p =  
query[Person]
```



KEEP CALM  
THROW  
WHITEBOX  
MACROS  
OUT OF  
A WINDOW

```
ityQuery[Person]) ...)
```

```
ef q = quote {
```



# In Scala 3 Quill Quotes will be Inline

... and life will be awesome!

```
inline def p =  
  query[Person]
```

```
inline def q = quote {  
  p  
}
```

```
sbt:dotty-simple> compile  
[info] Compiling 1 Scala source to /Users...  
SELECT x.name, x.age FROM Person x
```

# In Scala 3 Quill Quotes will be Inline

... and life will be awesome!

```
inline def onlyJoes =  
(p: Person) => p.name == "Joe"
```

```
inline def q = quote {  
    query[Person].filter(onlyJoes)  
}
```

```
List(Person("Joe", 22), Person("Jack", 33)).filter(onlyJoes)
```

```
[info] running miniquill.MiniExample3_InlineFilter  
List(Person(Joe,22))
```

```
inline def joes(inline q: Query[Person]) =  
  q.filter(p => p.name == "Joe")
```

```
inline def q = quote {  
  joes(query[Person])  
}
```

```
sbt:dotty-simple> compile  
[info] Compiling 1 Scala source to /Users...  
SELECT x.name, x.age FROM Person x WHERE p.name = 'Joe'
```

```
inline def joes(inline q: Query[Person], inline filter: Boolean) =  
  inline if (filter)  
    q.filter(p => p.name == "Joe")  
  else  
    q
```

```
inline def q = quote {  
  joes(query[Person], true)  
}
```

```
sbt:dotty-simple> compile  
[info] Compiling 1 Scala source to /Users...  
SELECT x.name, x.age FROM Person x WHERE p.name = 'Joe'
```

```
inline def joes(inline q: Query[Person], inline filter: Boolean) =  
  inline if (filter)  
    q.filter(p => p.name == "Joe")  
  else  
    q
```

```
inline def q = quote {  
  joes(query[Person], false)  
}
```

```
sbt:dotty-simple> compile  
[info] Compiling 1 Scala source to /Users...  
SELECT x.name, x.age FROM Person x
```

# Filters with Http4s

get: /app/people?isJoe=true

```
val service = HttpRoutes.of[IO] {
  case GET -> Root / "people" ?: JoesMatcher(isJoe: Boolean) =>
    Ok(stream(joinTables(lift(role), {
      val q =
        if (isJoe)
          quote { query[Person].filter(p => p.name == "Joe") }
        else
          quote { query[Person] }
      run(q)
    })).transact(xa).map(_.asJson))
```

[info] Compiling 1 Scala source to /Users  
Dynamic Query

```
val service = HttpRoutes.of[IO] {
  case GET -> Root / "people" ?: JoesMatcher(isJoe: Boolean) =>
    Ok(stream(joinTables(lift(role), {
      val q1 = quote { query[Person].filter(p => p.name == "Joe") }
      val q2 = quote { query[Person] }
      if (isJoe) run(q1) else run(q2)
    })).transact(xa).map(_.asJson))
```

[info] Compiling 1 Scala source to /Users  
SELECT p.name, p.age FROM Person p  
WHERE p.name = 'Joe'  
SELECT p.name, p.age FROM Person p

# Filters with Http4s

get: /app/people?isJoe=true

```
val service = HttpRoutes.of[IO] {
  case GET -> Root / "people" :? JoesMatcher(isJoe: Boolean) =>
    Ok(stream(joinTables(lift(role), {
      inline def people(inline isJoeInline: Boolean) =
        inline if (isJoeInline)
          query[Person].filter(p => p.name == "Joe")
        else
          query[Person]
        if (isJoe) run(q, true) else run(q, false)
    }))).transact(xa).map(_.asJson))
```

```
[info] Compiling 1 Scala source to /Users
SELECT p.name, p.page FROM Person p
  WHERE p.name = 'Joe'
SELECT p.name, p.page FROM Person p
```

# Optional Filters

```
inline def liftOrAny(inline field: String, inline filter: Option[String]) =  
  lift(filter.getOrElse(null)) == field ||  
  lift(filter.getOrElse(null)) == null
```

```
val runtimeValue = Some("Joe") // or None
```

```
inline def q = quote {  
  query[Person].filter(p => liftOrAny(p.name, runtimeValue))  
}
```

```
SELECT p.name, p.age FROM Person p WHERE ? = p.name OR ? IS NULL
```

# Optional Filters

```
inline def lift0rAny(inline field: String, inline filter: Option[String]) =  
  lift(filter.getOrElse(null)) == field ||  
  lift(filter.getOrElse(null)) == null
```

```
val runtimeValue = Some("Joe")
```

```
inline def q = quote {  
  query[Person].filter(p => lift0rAny(p.name, runtimeValue))  
}
```

```
SELECT p.name, p.age FROM Person p WHERE ? = p.name OR ? IS NULL
```



```
inline def liftOrAny(inline field: String, inline filter: Option[String]) =  
  lift(filter.getOrElse(null)) == field ||  
  lift(filter.getOrElse(null)) == null
```

get: /app/people or /app/people?name=Joe

```
val service = HttpRoutes.of[IO] {  
  case GET -> Root / "people" / NameMatcher(nameOpt: Option[String]) =>  
    Ok(stream(joinTables(lift(role), {  
      run { query[Person].filter(p => liftOrAny(p.name, nameOpt) }  
    })).transact(xa).map(_.asJson)))
```

SELECT p.name, p.age FROM Person p WHERE ? = p.name OR ? IS NULL



```
extension [T](inline q: EntityQuery[T]):  
  inline def filterByKeys(inline map: Map[String, String]): T =  
    q.filter(p => MapProc[T, PrepareRow](p, map, null, (a, b) => (a == b) || (b == (null) ) ))
```

```
case class Person(firstName: String, lastName: String, age: Int)  
  
val values: Map[String, String] = Map("firstName" -> "Joe", "age" -> "22")  
  
inline def q = quote {  
  query[Person].filterByKeys(values)  
}
```

```
SELECT p.firstName, p.lastName, p.age  
FROM Person p  
WHERE  
(p.firstName = ? OR ? IS NULL) AND  
(p.lastName = ? OR ? IS NULL) AND  
(p.age = ? OR ? IS NULL) AND  
true
```

```
extension [T](inline q: EntityQuery[T]):  
  inline def filterByKeys(inline map: Map[String, String]): T =  
    q.filter(p => MapProc[T, PrepareRow](p, map, null, (a, b) => (a == b) || (b == (null) ) ))
```

```
case class Person(firstName: String, lastName: String, age: Int)  
  
val values: Map[String, String] = Map("firstName" -> "Joe", "age" -> "22")  
  
inline def q = quote {  
  query[Person].filterByKeys(values)  
}
```

```
SELECT p.firstName, p.lastName, p.age  
FROM Person p  
WHERE  
(p.firstName = values.getOrElse("firstName",null) OR values.getOrElse("firstName",null) IS NULL) AND  
(p.lastName = values.getOrElse("lastName",null) OR values.getOrElse("lastName",null) IS NULL) AND  
(p.age = values.getOrElse("age",null) OR values.getOrElse("age",null) IS NULL) AND  
true
```

```
extension [T](inline q: EntityQuery[T]):  
  inline def filterByKeys(inline map: Map[String, String]): T =  
    q.filter(p => MapProc[T, PrepareRow](p, map, null, (a, b) => (a == b) || (b == (null) ) ))
```

```
case class Person(firstName: String, lastName: String, age: Int)  
  
val values: Map[String, String] = Map("firstName" -> "Joe", "age" -> "22")  
  
inline def q = quote {  
  query[Person].filterByKeys(values)  
}
```

```
SELECT p.firstName, p.lastName, p.age  
FROM Person p  
WHERE  
(p.firstName = 'Joe' OR 'Joe' IS NULL) AND  
(p.lastName = null OR null IS NULL) AND  
(p.age = '22' OR '22' IS NULL) AND  
true
```

get: /app/people?firstName=Joe&age=22

```
val service = HttpRoutes.of[IO] {
  case GET -> Root / "people" :? (multiFilters: Map[String, Seq[String]]) =>
    Ok(stream(joinTables(lift(role), {
      val filters: Map[String, String] = multiFilters.flatMap((k, v) => (k, v.take(1)))
      val q = quote {
        query[Person].filterByKeys(filters)
      }
      run(q)
    }))).transact(xa).map(_.asJson))
}
```

```
SELECT p.firstName, p.lastName, p.age
FROM Person p
WHERE
  (p.firstName = 'Joe' OR 'Joe' IS NULL) AND
  (p.lastName = null OR null IS NULL) AND
  (p.age = '22' OR '22' IS NULL) AND
  true
```

# Processing with Recursive Inlines

First Curse, then Recurse!

```
inline def oneOf(inline list: List[String], inline column:String): Boolean = {  
    inline if (ListProc.isNil(list))  
        false  
    else  
        ListProc.index(list, 0) == column || oneOf(ListProc.tail(list), column)  
}
```

```
case class Person(name: String, age: Int)  
  
inline def q = quote {  
    query[Person].filter(p => oneOf(List("Joe", "Jack"), p.name))  
}
```

```
[info] Compiling 1 Scala source to /Users...  
SELECT p.name, p.age FROM Person p WHERE 'Joe' = p.name OR 'Jack' = p.name OR false
```

# Processing with Recursive Inlines

First Curse, then Recurse!

```
inline def oneOf(inline list: List[String], inline column:String): Boolean = {  
    inline if (ListProc.isNil(list))  
        false  
    else  
        ListProc.index(list, 0) == column || oneOf(ListProc.tail(list), column)  
}
```

```
case class Node(status: String, lastStatus: String, backupStatus: Int)  
  
inline def q = quote {  
    query[Person].filter(n => oneOf(List(n.lastStatus, n.backupStatus), p.name))  
}
```

```
SELECT n.name, n.age FROM Node n  
WHERE n.lastStatus = n.status OR n.backupStatus = n.status OR false
```

# Processing with Recursive Inlines

First Curse, then Recurse!

```
inline def oneOf(inline list: List[String], inline column:String): Boolean = {  
    inline if (ListProc.isNil(list))  
        false  
    else  
        ListProc.index(list, 0) == column || oneOf(ListProc.tail(list), column)  
}
```

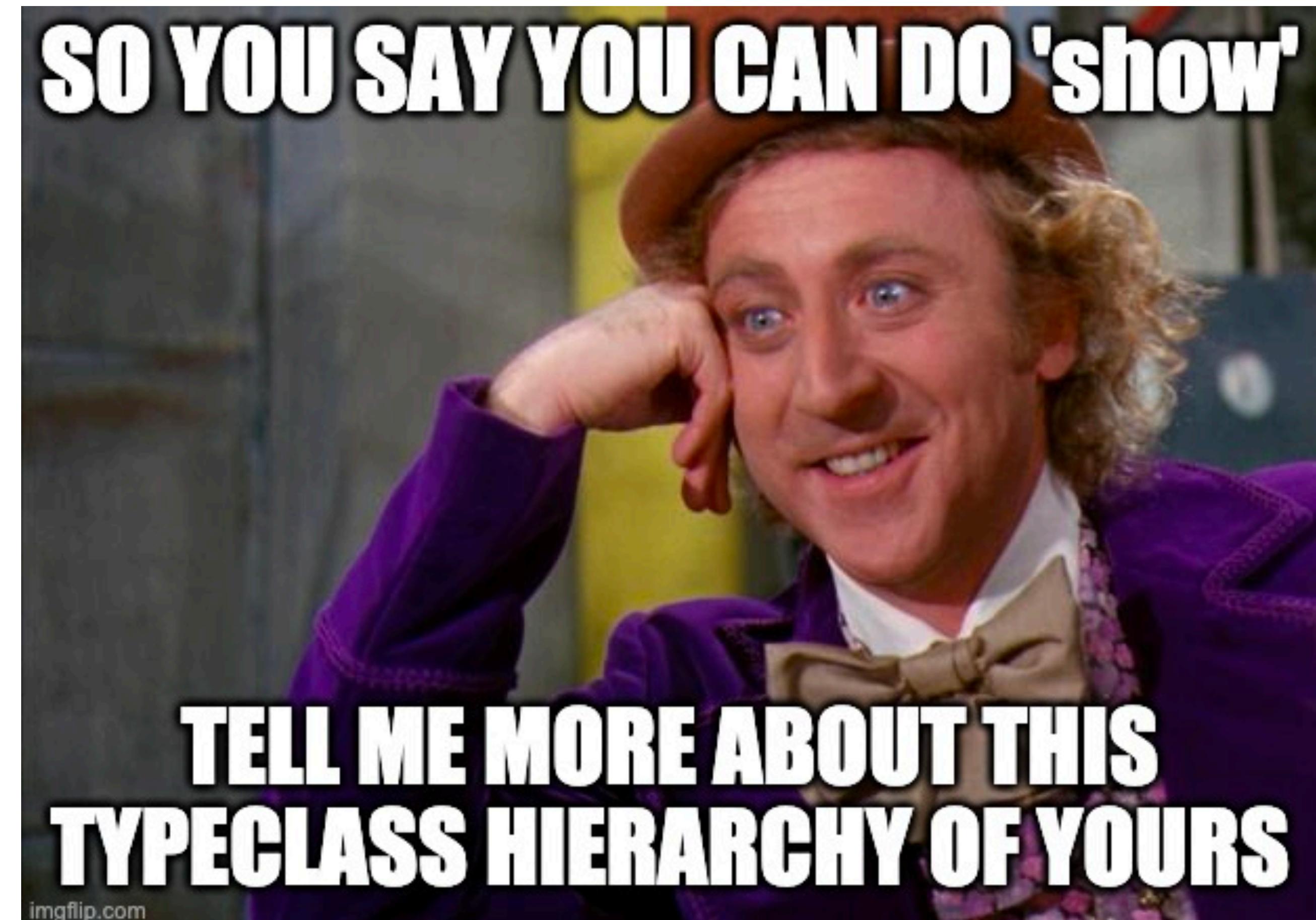
```
case class Node(status: String, lastStatus: String, backupStatus: Int)  
  
inline def q = quote {  
    query[Person].filter(n => oneOf(List(n.lastStatus,"restarting"), n.status))  
}
```

```
SELECT n.name, n.age FROM Node n  
WHERE n.lastStatus = n.status OR 'restarting' = n.status OR false
```

# — Inline Type-Classes

THERE'S ACTUALLY A GOOD REASON TO DO THIS!

NOTE: This Slide was skipped  
in the talk to save time.



# An Inline Functor...

No, really. This is actually possible!

```
trait Functor[F[_]]:  
  inline def map[A, B](inline xs: F[A], inline f: A => B): F[B]  
  
class ListFunctor extends Functor[List]:  
  inline def map[A, B](inline xs: List[A], inline f: A => B): List[B] = xs.map(f)  
  
class QueryFunctor extends Functor[Query]:  
  inline def map[A, B](inline xs: Query[A], inline f: A => B): Query[B] = xs.map(f)  
  
inline given listFunctor as ListFunctor = new ListFunctor  
inline given queryFunctor as QueryFunctor = new QueryFunctor  
  
inline def doMap[F[_], A, B](inline from: F[A], inline f: A => B)(using inline fun: Functor[F]): F[B] =  
  fun.map(from, f)
```

```
val list2 = doMap(List(1,2,3), (i: Int) => i + 1)          // Runtime  
inline def q = quote { doMap(select[Person], (p: Person) => p.name) } // Compile-Time!
```

```
[info] Compiling 1 Scala source to /Users...  
Compile Time Query Is: SELECT p.name FROM Person p
```

```
runMain miniquill.InlineMacroTest1FunctionalTypeclass  
List(2, 3, 4)
```

# An Inline Functor... Dotty Idiomatically!

Tell them you saw it here first!

```
trait Functor[F[_]]:  
  extension [A, B](inline x: F[A]):  
    inline def map(inline f: A => B): F[B]  
  
inline given Functor[List]:  
  extension [A, B](inline xs: List[A])  
    inline def map(inline f: A => B): List[B] = xs.map(f)  
  
inline given Functor[Query]:  
  extension [A, B](inline xs: Query[A])  
    inline def map(inline f: A => B): Query[B] = xs.map(f)  
  
extension [F[_], A, B](inline from: F[A])(using inline fun: Functor[F]):  
  inline def mapF(inline f: A => B) = from.map(f)
```

---

```
println( List(1,2,3).mapF(i => i + 1) )  
inline def q = quote { select[Person].up.mapF(p => p.name) }
```

NOTE: This Slide was skipped  
in the talk to save time.

# ... and a Monad!

Well, not really a Monad since we don't have Pure.  
Actually it's the FlatMap Typeclass

```
trait Monad[F[_]] extends Functor[F]:  
  extension [A, B](inline x: F[A]):  
    inline def map(inline f: A => B): F[B]  
    inline def flatMap(inline f: A => F[B]): F[B]  
  
  inline given Monad[List]:  
    extension [A, B](inline xs: List[A])  
      inline def map(inline f: A => B): List[B] = xs.map(f)  
      inline def flatMap(inline f: A => List[B]): List[B] = xs.flatMap(f)  
  
  inline given Monad[Query]:  
    extension [A, B](inline xs: Query[A])  
      inline def map(inline f: A => B): Query[B] = xs.map(f)  
      inline def flatMap(inline f: A => Query[B]): Query[B] = xs.flatMap(f)  
  
  extension [F[_], A, B](inline from: F[A])(using inline fun: Monad[F]):  
    inline def mapM(inline f: A => B) = from.map(f)  
    inline def flatMapM(inline f: A => F[B]) = from.flatMap(f)  
  
  inline def q = quote { select[Person].mapF(p => p.name) }  
  inline def q = quote { select[Person].flatMapM(p => query[Address]) }
```

# Comprehend This!

... and you will be saved!

```
trait For[F[_]]:
  extension [A, B](inline x: F[A]):
    inline def map(inline f: A => B): F[B]
    inline def flatMap(inline f: A => F[B]): F[B]
    inline def withFilter(inline f: A => Boolean): F[A]

  inline given For[List]:
    extension [A, B](inline xs: List[A])
      inline def map(inline f: A => B): List[B] = xs.map(f)
      inline def flatMap(inline f: A => List[B]): List[B] = xs.flatMap(f)
      inline def withFilter(inline f: A => Boolean): List[A] = xs.filter(f)

  inline given For[Query]:
    extension [A, B](inline xs: Query[A])
      inline def map(inline f: A => B): Query[B] = xs.map(f)
      inline def flatMap(inline f: A => Query[B]): Query[B] = xs.flatMap(f)
      inline def withFilter(inline f: A => Boolean): Query[A] = xs.withFilter(f)
```

---

```
object UseCase:
  extension [F[_]](inline people: F[Person])(using inline fun: For[F]):
    inline def joesAddresses(inline addresses: F[Address]) =
      for {
        p <- people if (p.name == "Joe")
        a <- addresses if (p.id == a.fk)
      } yield (p, a)
```

NOTE: This Slide was skipped  
in the talk to save time.

# Comprehend This!

... and you will be saved!

```
object UseCase:  
  extension [F[_]](inline people: F[Person])(using inline fun: For[F]):  
    inline def joesAddresses(inline addresses: F[Address]) =  
      for {  
        p <- people if (p.name == "Joe")  
        a <- addresses if (p.id == a.fk)  
      } yield (p, a)
```

```
inline def q = quote { people.joesAddresses(addresses) }
```

Compile Time

```
[info] Compiling 1 Scala source to /Users...  
SELECT p.id, p.name, p.age, a.fk, a.street, a.zip FROM Person p, Address a  
WHERE p.name = 'Joe' AND p.id = a.fk
```

```
val peopleL = List(Person(1, "Joe", 22), Person(2, "Jack", 33), Person(3, "James", 44))  
val addressesL = List(Address(1, "123 St.", 111), Address(2, "456 St.", 222), Address(3, "789 St.", 333))
```

Runtime

```
println(peopleL.joesAddresses(addressesL))
```

```
[info] running miniquill.InlineMacroTest1FunctionalTypeclassIdiomatic  
List((Person(1,Joe,22),Address(1,123 St.,111)))
```

TypeclassExample\_Forscala

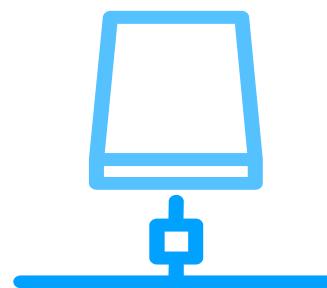
**BLAH BLAH BLAH  
TYPE-THEORY**

**GIVE ME AN ACTUAL  
EXAMPLE!**

# A Practical Example

... you say boring, I say believable!

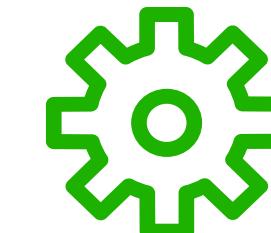
**Node**



**Master**



**Worker**



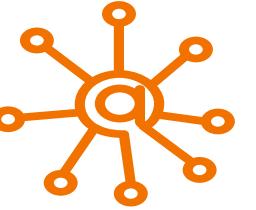
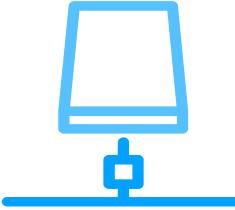
<b>Log</b>
Time1 - Node1 Update
Time2 - Node2 Update
Time3 - Node1 Update
Time4 - Node1 Update
Time5 - Node2 Update

<b>Log</b>
Time1 - Master1 Update
Time2 - Master2 Update
Time3 - Master3 Update
Time4 - Master1 Update
Time5 - Master3 Update

<b>Log</b>
Time1 - Node1 Update
Time2 - Node3 Update
Time3 - Node3 Update
Time4 - Node2 Update
Time5 - Node2 Update

# A Practical Example

... you say boring, I say believable!



Node		
<b>id</b>	<b>timestamp</b>	<b>status</b>
1	1	UP
2	2	UP
1	3	DOWN
1	4	UP
2	5	DOWN

Master		
<b>id</b>	<b>lastCheck</b>	<b>state</b>
1	1	IDLE
2	2	WAITING
3	3	WAITING
1	4	WAITING
3	5	FINISHED

Worker		
<b>shard</b>	<b>lastTime</b>	<b>reply</b>
1	1	BUSY
3	2	BUSY
3	3	DONE
2	4	DONE
2	5	BUSY

Latest Node Status		
<b>id</b>	<b>timestamp</b>	<b>status</b>
1	4	UP
2	5	DOWN

Latest Master Status		
<b>id</b>	<b>lastCheck</b>	<b>state</b>
1	5	WAITING
2	2	WAITING
3	5	FINISHED

Latest Worker Status		
<b>shard</b>	<b>lastTime</b>	<b>reply</b>
1	5	BUSY
2	2	BUSY
3	5	DONE

# A Practical Example

... you say boring, I say believable!

```
SELECT  
  a.id, a.timestamp, a.status  
FROM Node a  
LEFT JOIN Node b  
ON b.id = a.id  
AND b.timestamp > a.timestamp  
WHERE b.id IS NULL
```

```
SELECT  
  a.key, a.lastCheck, a.state  
FROM Master a  
LEFT JOIN Master b  
ON b.key = a.key  
AND b.lastCheck > a.lastCheck  
WHERE b.key IS NULL
```

```
SELECT  
  a.shard, a.lastTime, a.reply  
FROM Worker a  
LEFT JOIN Worker b  
ON b.shard = a.shard  
AND b.lastTime > a.lastTime  
WHERE b.shard IS NULL
```

Latest Node Status		
<b>id</b>	<b>timestamp</b>	<b>status</b>
1	4	UP
2	5	DOWN

Latest Master Status		
<b>id</b>	<b>lastCheck</b>	<b>state</b>
1	5	WAITING
2	2	WAITING
3	5	FINISHED

Latest Worker Status		
<b>shard</b>	<b>lastTime</b>	<b>reply</b>
1	5	BUSY
2	2	BUSY
3	5	DONE

# A Practical Example

... you say boring, I say believable!

```
inline def nodes = quote {  
  query[Node].leftJoin(query[Node])  
  .on((a, b) =>  
    b.id == a.id &&  
    b.timestamp > a.timestamp  
)  
  .filter((a, b) =>  
    b.map(_.id).isEmpty)  
  .map((a, b) => a)  
}
```

```
inline def masters = quote {  
  query[Master].leftJoin(query[Master])  
  .on((a, b) =>  
    b.key == a.key &&  
    b.lastCheck > a.lastCheck  
)  
  .filter((a, b) =>  
    b.map(_.key).isEmpty)  
  .map((a, b) => a)  
}
```

```
inline def workers = quote {  
  query[Worker].leftJoin(query[Worker])  
  .on((a, b) =>  
    b.shard == a.shard &&  
    b.lastTime > a.lastTime  
)  
  .filter((a, b) =>  
    b.map(_.shard).isEmpty)  
  .map((a, b) => a)  
}
```

Latest Node Status		
<b>id</b>	<b>timestamp</b>	<b>status</b>
1	4	UP
2	5	DOWN

Latest Master Status		
<b>id</b>	<b>lastCheck</b>	<b>state</b>
1	5	WAITING
2	2	WAITING
3	5	FINISHED

Latest Worker Status		
<b>shard</b>	<b>lastTime</b>	<b>reply</b>
1	5	BUSY
2	2	BUSY
3	5	DONE

# A Practical Example

... you say boring, I say believable!

```
inline def nodes = quote {
  query[Node].leftJoin(query[Node])
  .on((a, b) =>
    b.id == a.id &&
    b.timestamp > a.timestamp
  )
  .filter((a, b) =>
    b.map(_.id).isEmpty)
  .map((a, b) => a)
}
```

Latest Node Status		
id	timestamp	status
1	4	UP
2	5	DOWN

```
inline def masters = quote {
  query[Master].leftJoin(query[Master])
  .on((a, b) =>
    b.key == a.key &&
    b.lastCheck > a.lastCheck
  )
  .filter((a, b) =>
    b.map(_.key).isEmpty)
  .map((a, b) => a)
}
```

Latest Master Status		
id	lastCheck	state
1	5	WAITING
2	2	WAITING

```
inline def masters = quote {
  query[Master].leftJoin(query[Master])
  .on((a, b) =>
    b.key == a.key &&
    b.lastCheck > a.lastCheck
  )
  .filter((a, b) =>
    b.map(_.key).isEmpty)
  .map((a, b) => a)
}
```

Latest Worker Status		
shard	lastTime	reply
1	5	BUSY
2	2	BUSY

```
inline def masters = quote {
  query[Entity].leftJoin(query[Entity])
  .on((a, b) =>
    b.key == a.key &&
    b.lastCheck > a.lastCheck
  )
  .filter((a, b) =>
    b.map(_.key).isEmpty)
  .map((a, b) => a)
}
```

Latest Node Entity		
id	timestamp	status
1	4	UP
2	5	DOWN

```
inline def workers = quote {
  query[Service].leftJoin(query[Service])
  .on((a, b) =>
    b.shard == a.shard &&
    b.lastTime > a.lastTime
  )
  .filter((a, b) =>
    b.map(_.shard).isEmpty)
  .map((a, b) => a)
}
```

Latest Master Service		
id	lastCheck	state
1	5	WAITING
2	2	WAITING

```
inline def masters = quote {
  query[Switch].leftJoin(query[Switch])
  .on((a, b) =>
    b.key == a.key &&
    b.lastCheck > a.lastCheck
  )
  .filter((a, b) =>
    b.map(_.key).isEmpty)
  .map((a, b) => a)
}
```

Latest Worker Switch		
shard	lastTime	reply
1	5	BUSY
2	2	BUSY

```
inline def nodes = quote {
  query[App].leftJoin(query[App])
  .on((a, b) =>
    b.id == a.id &&
    b.timestamp > a.timestamp
  )
  .filter((a, b) =>
    b.map(_.id).isEmpty)
  .map((a, b) => a)
}
```

Latest Node App		
id	timestamp	status
1	4	UP
2	5	DOWN

```
inline def masters = quote {
  query[Bridge].leftJoin(query[Bridge])
  .on((a, b) =>
    b.key == a.key &&
    b.lastCheck > a.lastCheck
  )
  .filter((a, b) =>
    b.map(_.key).isEmpty)
  .map((a, b) => a)
}
```

Latest Master Bridge		
id	lastCheck	state
1	5	WAITING
2	2	WAITING

```
inline def masters = quote {
  query[Combo].leftJoin(query[Combo])
  .on((a, b) =>
    b.key == a.key &&
    b.lastCheck > a.lastCheck
  )
  .filter((a, b) =>
    b.map(_.key).isEmpty)
  .map((a, b) => a)
}
```

Latest Worker Combo		
shard	lastTime	reply
1	5	BUSY
2	2	BUSY

```
inline def masters = quote {
  query[Reso].leftJoin(query[Reso])
  .on((a, b) =>
    b.key == a.key &&
    b.lastCheck > a.lastCheck
  )
  .filter((a, b) =>
    b.map(_.key).isEmpty)
  .map((a, b) => a)
}
```

Latest Node Reso		
id	timestamp	status
1	4	UP
2	5	DOWN

```
inline def workers = quote {
  query[Database].leftJoin(query[Database])
  .on((a, b) =>
    b.shard == a.shard &&
    b.lastTime > a.lastTime
  )
  .filter((a, b) =>
    b.map(_.shard).isEmpty)
  .map((a, b) => a)
}
```

Latest Master Database		
id	lastCheck	state
1	5	WAITING
2	2	WAITING

```
inline def masters = quote {
  query[Route].leftJoin(query[Route])
  .on((a, b) =>
    b.key == a.key &&
    b.lastCheck > a.lastCheck
  )
  .filter((a, b) =>
    b.map(_.key).isEmpty)
  .map((a, b) => a)
}
```

Latest Worker Route		
shard	lastTime	reply
1	5	BUSY
2	2	BUSY

# A Practical Example

... you say boring, I say believable!

```
inline def nodes = quote {  
    query[Node].leftJoin(query[Node])  
    .on((a, b) =>  
        b.id == a.id &&  
        b.timestamp > a.timestamp  
    )  
    .filter((a, b) =>  
        b.map(_.id).isEmpty)  
    .map((a, b) => a)  
}
```

Latest Node Status		
id	timestamp	status
1	4	UP
2	5	DOWN

```
inline def nodes = quote {  
    query[App].leftJoin(query[App])  
    .on((a, b) =>  
        b.id == a.id &&  
        b.timestamp > a.timestamp  
    )  
    .filter((a, b) =>  
        b.map(_.id).isEmpty)  
    .map((a, b) => a)  
}
```

Latest Node App		
id	timestamp	status
1	4	UP
2	5	DOWN



```
inline def masters = quote {  
    query[Switch].leftJoin(query[Switch])  
    .on((a, b) =>  
        b.key == a.key &&  
        b.lastCheck > a.lastCheck  
    )  
    .filter((a, b) =>  
        b.map(_.key).isEmpty)  
    .map((a, b) => a)  
}
```

Latest Worker Switch		
shard	lastTime	reply
1	5	BUSY
2	2	BUSY
3	5	DONE

```
inline def masters = quote {  
    query[Route].leftJoin(query[Route])  
    .on((a, b) =>  
        b.key == a.key &&  
        b.lastCheck > a.lastCheck  
    )  
    .filter((a, b) =>  
        b.map(_.key).isEmpty)  
    .map((a, b) => a)  
}
```

Latest Worker Route		
shard	lastTime	reply
1	5	BUSY
2	2	BUSY

# A Practical Example

... you say boring, I say believable!

```
inline def latestStatus[T, G] =  
  inline q: Query[T])(  
    inline groupKey: T => G,  
    inline earlierThan: (T, T) => Boolean  
  ): Query[T] =  
    q.leftJoin(q)  
      .on((a, b) =>  
        groupKey(b) == groupKey(a) &&  
        earlierThan(b, a)  
      )  
      .filter((a, b) =>  
        b.map(b => groupKey(b)).isEmpty)  
      .map((a, b) => a)
```

Latest Node Status		
<b>id</b>	<b>timestamp</b>	<b>status</b>
1	4	UP
2	5	DOWN

Latest Master Status		
<b>id</b>	<b>lastCheck</b>	<b>state</b>
1	5	WAITING
2	2	WAITING
3	5	FINISHED

Latest Worker Status		
<b>id</b>	<b>lastTime</b>	<b>reply</b>
1	5	BUSY
2	2	BUSY
3	5	DONE

```
inline def nodesLatest = quote {  
  latestStatus(query[Node])(  
    n => n.id,  
    (a, b) => a.timestamp < b.timestamp  
  )}
```

```
inline def mastersLatest = quote {  
  latestStatus(query[Master])(  
    m => m.key,  
    (a, b) => a.lastCheck < b.lastCheck  
  )}
```

```
inline def workersLatest = quote {  
  latestStatus(query[Worker])(  
    w => w.shard,  
    (a, b) => a.lastTime < b.lastTime  
  )}
```

# A Practical Example

... you say boring, I say believable!

```
trait GroupKey[T, G]:  
    inline def apply(inline t: T): G  
trait EarlierThan[T]:  
    inline def apply(inline a: T, inline b: T): Boolean
```

```
inline given GroupKey[Node, Int]:  
    inline def apply(inline t: Node): Int = t.id  
inline given GroupKey[Master, Int]:  
    inline def apply(inline t: Master): Int = t.key  
inline given GroupKey[Worker, Int]:  
    inline def apply(inline t: Worker): Int = t.shard  
  
inline given EarlierThan[Node]:  
    inline def apply(inline a: Node, inline b: Node) = a.timestamp < b.timestamp  
inline given EarlierThan[Master]:  
    inline def apply(inline a: Master, inline b: Master) = a.lastCheck < b.lastCheck  
inline given EarlierThan[Worker]:  
    inline def apply(inline a: Worker, inline b: Worker) = a.lastTime < b.lastTime
```

```
inline def latestStatus[T, G](
  inline q: Query[T])(  

  using inline groupKey: GroupKey[T, G],  

  inline earlierThan: EarlierThan[T]
): Query[T] =  

  q.leftJoin(q)  

    .on((a, b) =>  

      groupKey(b) == groupKey(a) &&  

      earlierThan(b, a)
    )
    .filter((a, b) =>  

      b.map(b => groupKey(b)).isEmpty)
    .map((a, b) => a)
```

```
quote { latestStatus(select[Node]) }  

quote { latestStatus(select[Master]) }  

quote { latestStatus(select[Worker]) }
```

```

inline def latestStatus[T, G](
  inline q: Query[T])(
  using inline groupKey: GroupKey[T, G],
  inline earlierThan: EarlierThan[T]
): Query[T] =
  q.leftJoin(q)
    .on((a, b) =>
      groupKey(b) == groupKey(a) &&
      earlierThan(b, a))
    )
    .filter((a, b) =>
      b.map(b => groupKey(b)).isEmpty)
    .map((a, b) => a)

```

```

quote { latestStatus(select[Node]) }

  SELECT
    a.id, a.timestamp, a.status
  FROM Node a
  LEFT JOIN Node b
  ON b.id = a.id
  AND b.timestamp > a.timestamp
  WHERE b.id IS NULL

```

```

quote { latestStatus(select[Master]) }

  SELECT
    a.key, a.lastCheck, a.state
  FROM Master a
  LEFT JOIN Master b
  ON b.key = a.key
  AND b.lastCheck > a.lastCheck
  WHERE b.key IS NULL

```

```

quote { latestStatus(select[Worker]) }

  SELECT
    a.shard, a.lastTime, a.reply
  FROM Worker a
  LEFT JOIN Worker b
  ON b.shard = a.shard
  AND b.lastTime > a.lastTime
  WHERE b.shard IS NULL

```



imgflip.com

quote { latestStatus(select [Node] ) }

```
SELECT  
    a.id, a.timestamp, a.status  
FROM Node a  
LEFT JOIN Node b  
ON b.id = a.id  
AND b.timestamp > a.timestamp  
WHERE b.id IS NULL
```

quote { latestStatus(select [Master] ) }

```
SELECT  
    a.key, a.lastCheck, a.state  
FROM Master a  
LEFT JOIN Master b  
ON b.key = a.key  
AND b.lastCheck > a.lastCheck  
WHERE b.key IS NULL
```

quote { latestStatus(select [Worker] ) }

```
SELECT  
    a.shard, a.lastTime, a.reply  
FROM Worker a  
LEFT JOIN Worker b  
ON b.shard = a.shard  
AND b.lastTime > a.lastTime  
WHERE b.shard IS NULL
```

# Let's Take One More Step

... because we can!

```
trait JoiningFunctor[F[_]]:  
  extension [A, B](inline xs: F[A])  
    inline def map(inline f: A => B): F[B]  
    inline def filter(inline f: A => Boolean): F[A]  
    inline def leftJoin(inline ys: F[B])(inline f: (A, B) => Boolean): F[(A, Option[B])]
```

# Now Implement!

Faster! Faster!

```
inline given JoiningFunctor[Query]:  
  extension [A, B](inline xs: Query[A])  
    inline def map(inline f: A => B): Query[B] = xs.map(f)  
    inline def filter(inline f: A => Boolean): Query[A] = xs.filter(f)  
    inline def leftJoin(inline ys: Query[B])(inline f: (A, B) => Boolean): Query[(A, Option[B])] =  
      xs.leftJoin(ys).on(f)  
  
inline given JoiningFunctor[List]:  
  extension [A, B](inline xs: List[A])  
    inline def map(inline f: A => B): List[B] = xs.map(f)  
    inline def filter(inline f: A => Boolean): List[A] = xs.filter(f)  
    inline def leftJoin(inline ys: List[B])(inline f: (A, B) => Boolean): List[(A, Option[B])] =  
      xs.flatMap { x =>  
        val matching = ys.filter(y => f(x, y)).map(y => (x, Some(y)))  
        if (matching.length == 0) List((x, None)) else matching  
      }
```

# With a Query! With a List!

Believe me now?

```
inline def latestStatus[F[_], T, G](inline q: F[T])(  
  using inline fun: JoiningFunctor[F],  
  inline groupKey: GroupKey[T, G],  
  inline earlierThan: EarlierThan[T]): F[T] =  
  q.leftJoin(q)((a, b) =>  
    groupKey(b) == groupKey(a) &&  
    earlierThan(b, a)  
)  
.filter((a, b) =>  
  b.map(b => groupKey(b)).isEmpty)  
.map((a, b) => a)
```

```
quote {  
  latestStatus(select [Node])  
}
```

```
SELECT  
  a.id, a.timestamp, a.status  
FROM Node a  
LEFT JOIN Node b  
ON b.id = a.id  
AND b.timestamp > a.timestamp  
WHERE b.id IS NULL
```

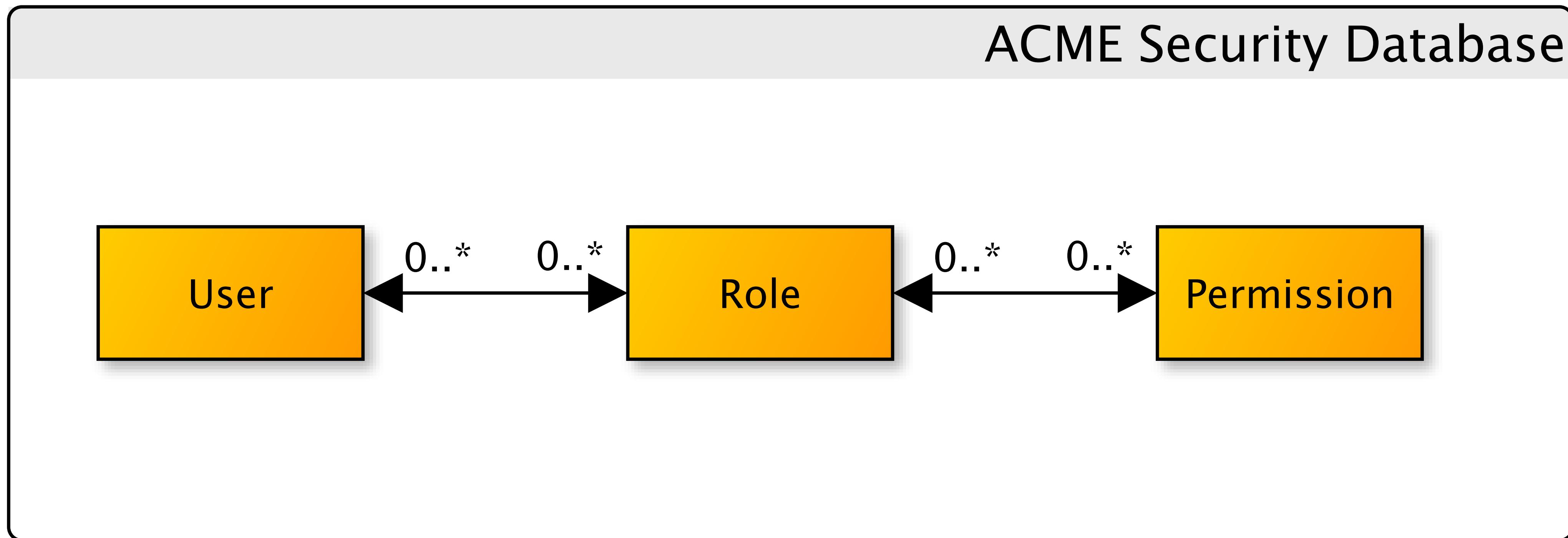
```
val nodesList = List(  
  Node(1, 1, "UP"),  
  Node(1, 2, "DOWN"),  
  Node(2, 3, "WAITING")  
)  
println( latestStatus(nodesList) )  
List(Node(1,1,UP), Node(2,3,WAITING))
```

# Inline Type-Level Programming

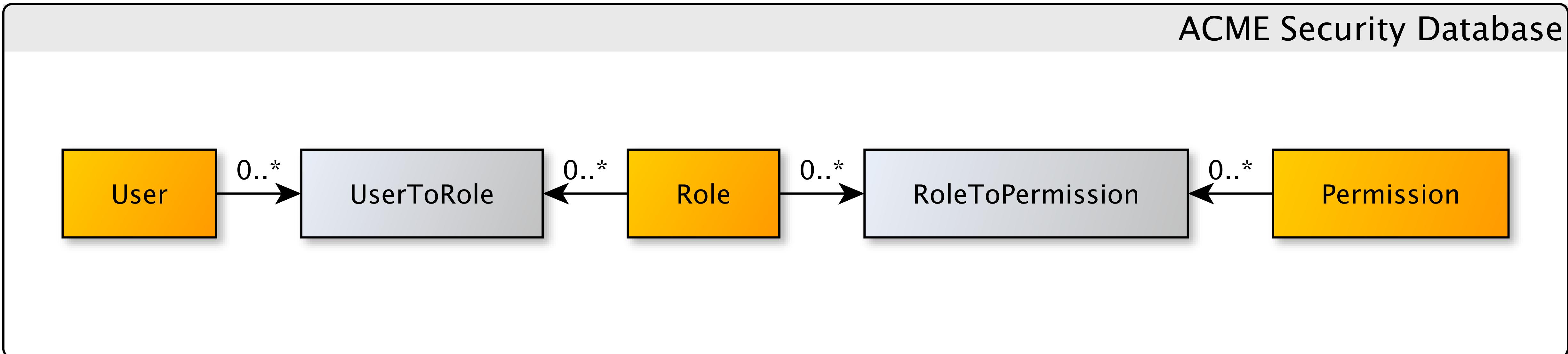
---

SERIOUSLY... WE'RE DOING THIS!

# Every Security DB in the Universe...

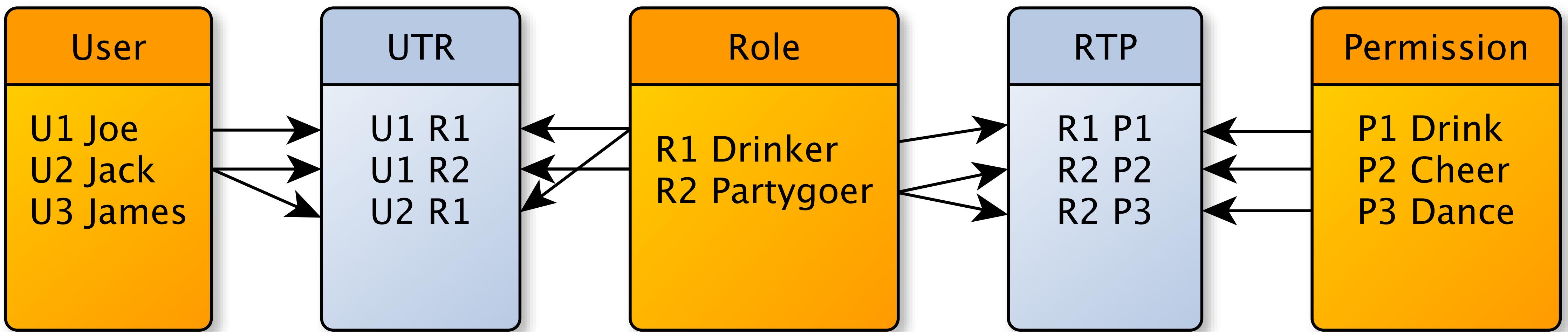


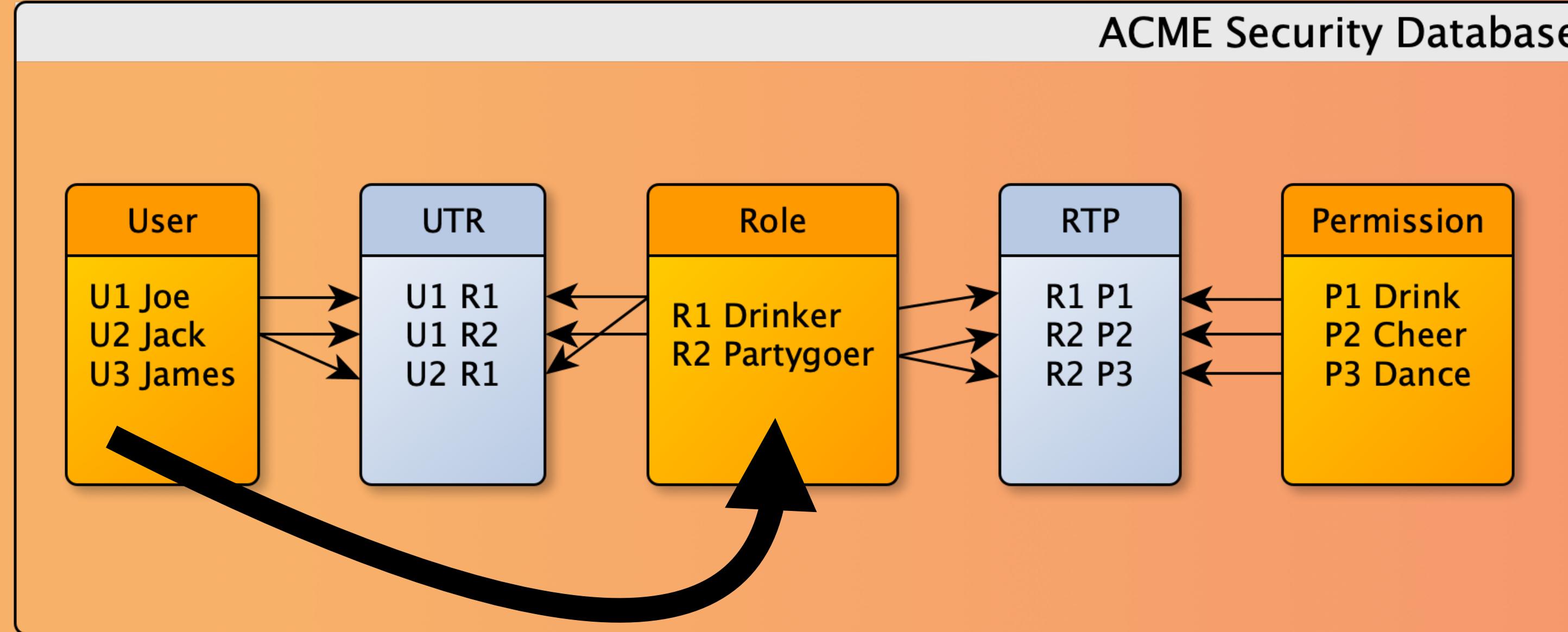
# Actually it's...



# Actually it's...

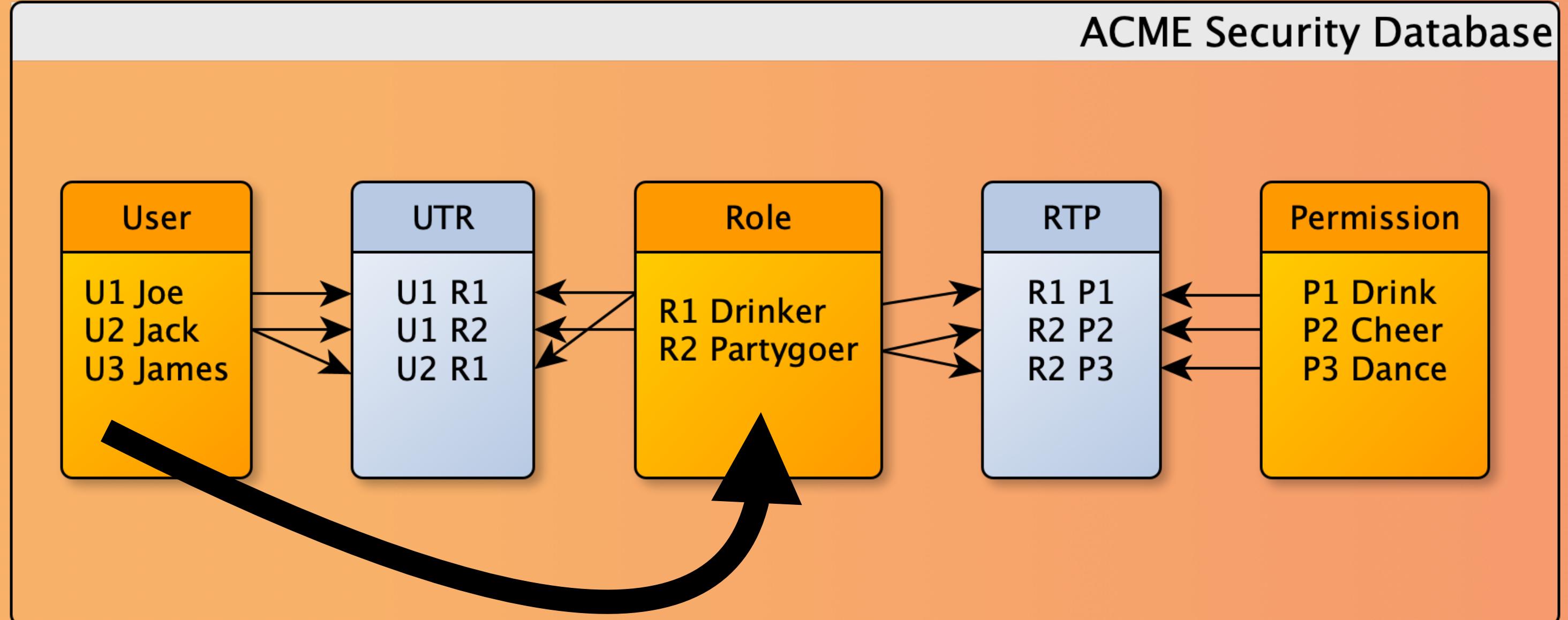
ACME Security Database





```

SELECT
    s.id, s.name,
    r.id, r.name
FROM
    User s
    INNER JOIN UserToRole so ON so.userId = s.id
    INNER JOIN Role r ON r.id = so.roleId
  
```

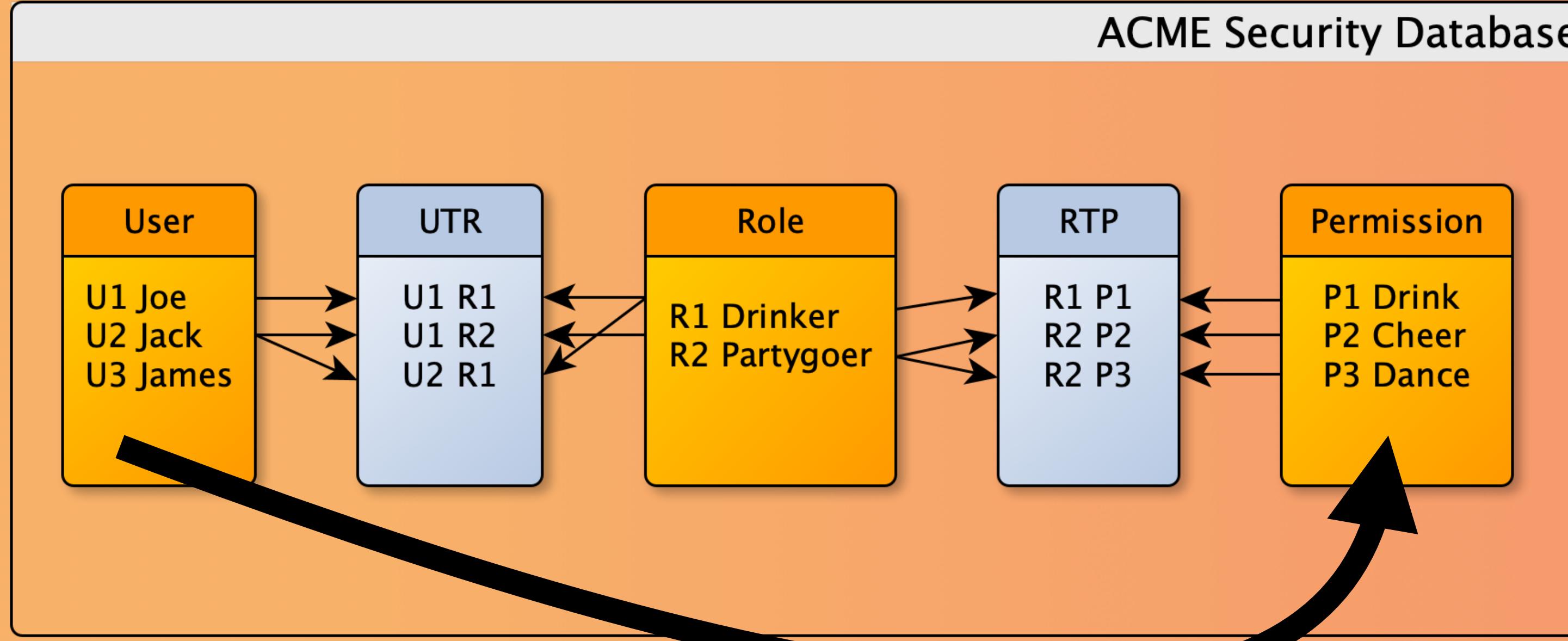


```

SELECT
  s.id, s.name,
  r.id, r.name
FROM
  User s
INNER JOIN UserToRole so ON so.userId = s.id
INNER JOIN Role r ON r.id = so.roleId
  
```

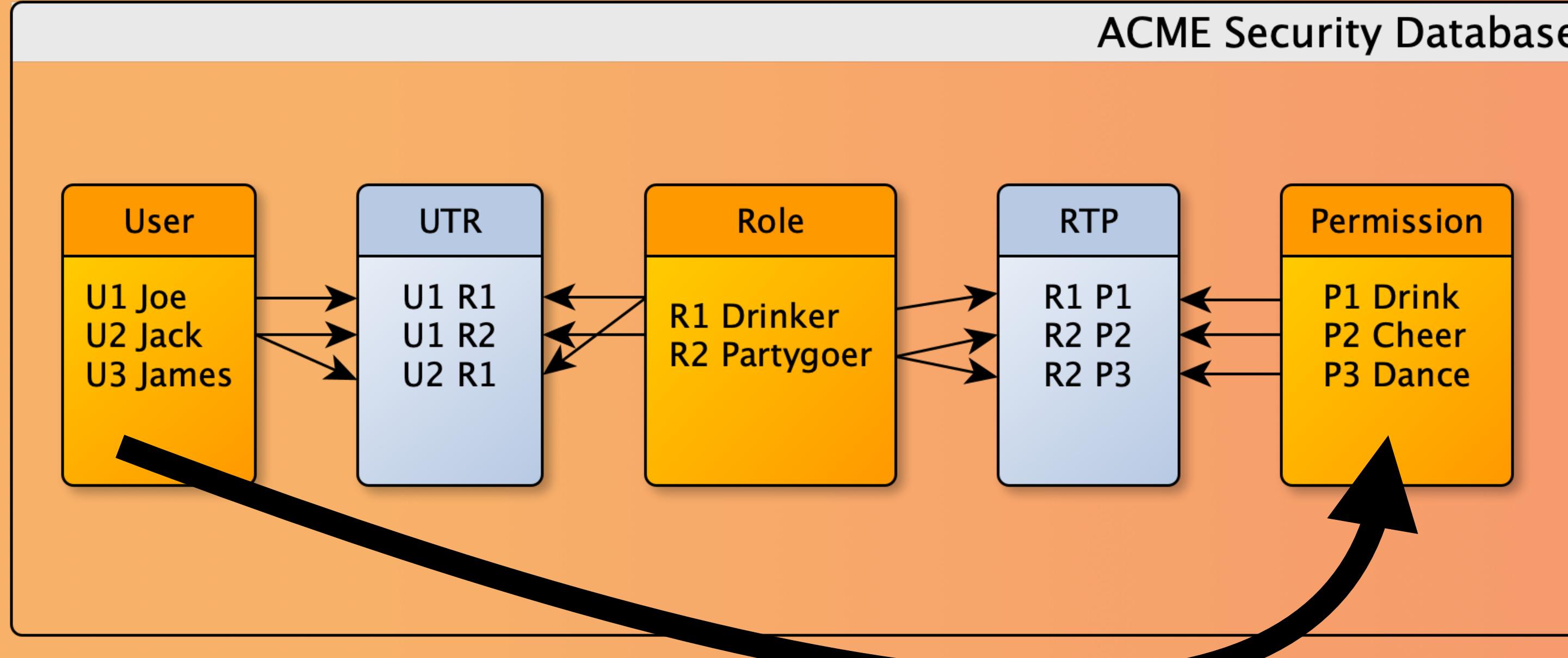
```

for {
  s <- query[User]
  sr <- query[UserToRole].join(sr => sr.userId == s.id)
  r <- query[Role].join(r => r.id == sr.roleId)
} yield (s, r)
  
```



```

SELECT
    s.id, s.name,
    r.id, r.name,
    p.id, p.name
FROM
    User s
    INNER JOIN UserToRole so ON so.userId = s.id
    INNER JOIN Role r ON r.id = so.roleId
    INNER JOIN RoleToPermission rp ON rp.roleId = r.id
    INNER JOIN Permission p ON p.id = rp.roleId
    
```



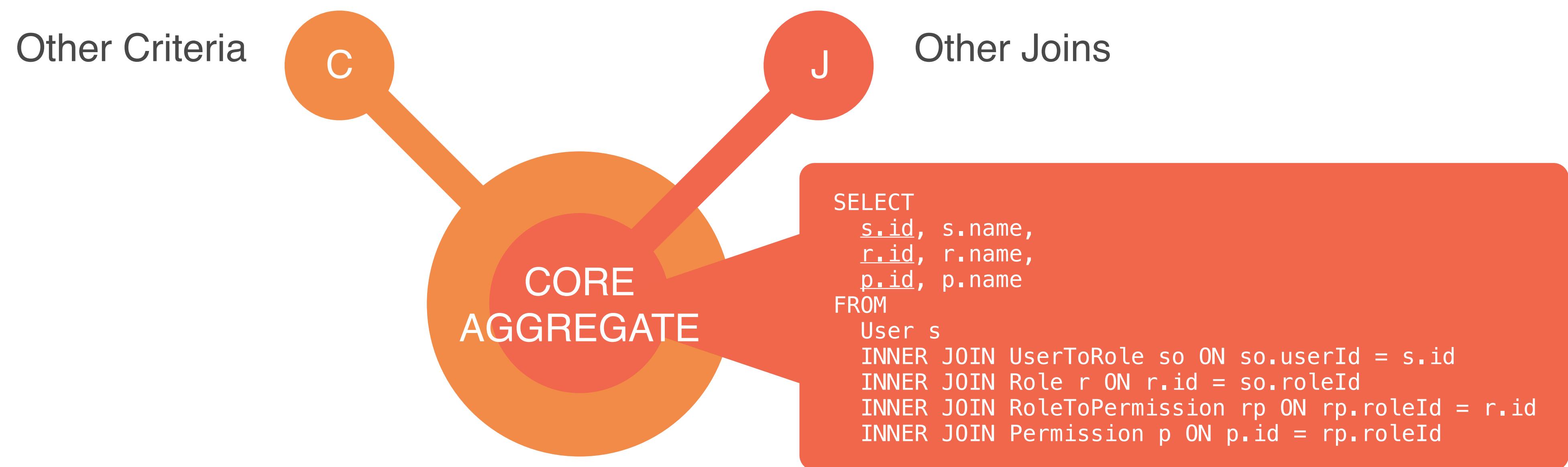
```

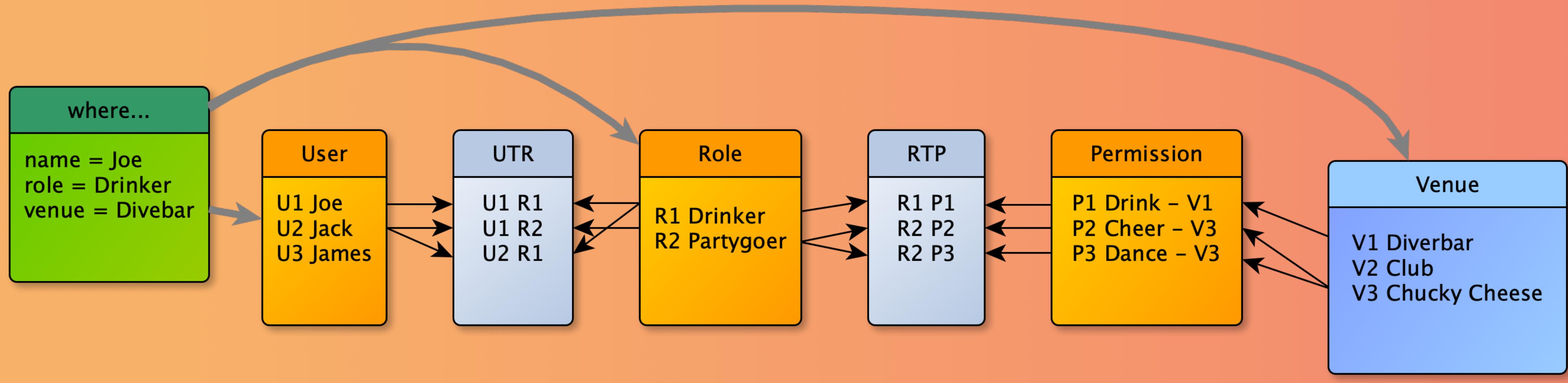
SELECT
  s.id, s.name,
  r.id, r.name,
  p.id, p.name
FROM
  User s
  INNER JOIN UserToRole so ON so.userId = s.id
  INNER JOIN Role r ON r.id = so.roleId
  INNER JOIN RoleToPermission rp ON rp.roleId = r.id
  INNER JOIN Permission p ON p.id = rp.roleId
  
```

```

for {
  u <- query[User]
  ur <- query[UserToRole].join(so => so.userId == u.id)
  r <- query[Role].join(r => r.id == ur.roleId)
  rp <- query[RoleToPermission].join(rp => rp.roleId == r.id)
  p <- query[Permission].join(p => p.id == rp.roleId)
} yield (u, r, p)
  
```

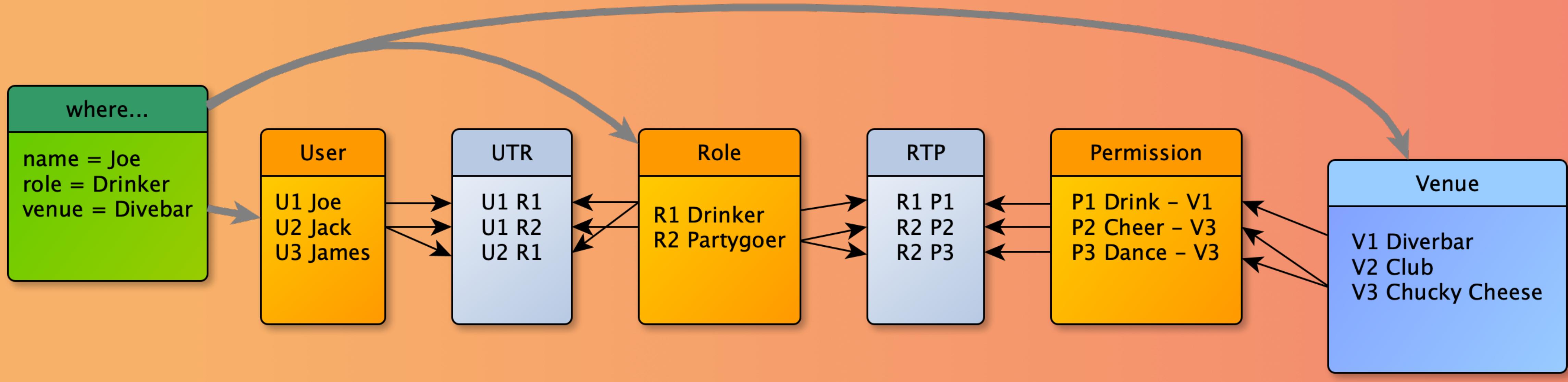
## ... but usually, there's other stuff





```

SELECT
    u.id, u.name, r.id, r.name, p.id, p.name, p.venueId, v.id, v.name
FROM
    User u
    INNER JOIN UserToRole so ON so.userId = u.id
    INNER JOIN Role r ON r.id = so.roleId
    INNER JOIN RoleToPermission rp ON rp.roleId = r.id
    INNER JOIN Permission p ON p.id = rp.roleId
    INNER JOIN Venue v ON v.id = p.venueId
WHERE
    u.name = 'Joe' AND r.name = 'Drinker' AND p.name = 'Drink' AND v.name = 'Divebar'
  
```



```

for {
    u <- query[User] if (u.name == "Joe")
    ur <- query[UserToRole].join(so => so.userId == u.id)
    r <- query[Role].join(r => r.id == ur.roleId) if (r.name == "Drinker")
    rp <- query[RoleToPermission].join(rp => rp.roleId == r.id)
    p <- query[Permission].join(p => p.id == rp.roleId)
    v <- query[Venue].join(v => v.id == p.venueId) if (v.name == "Divebar")
} yield (u, r, p, v)
  
```



```
for {  
    u <- query[User] if (u.name == "Joe")  
    ur <- query[UserToRole].join(so => so.userId == u.id)  
    r <- query[Role].join(r => r.id == ur.roleId) if (r.name == "Drinker")  
    rp <- query[RoleToPermission].join(rp => rp.roleId == r.id)  
    p <- query[Permission].join(p => p.id == rp.roleId)  
    v <- query[Venue].join(v => v.id == p.venueId) if (v.name == "Divebar")  
} yield (u, r, p, v)
```

# Bring on the API Calls!...

Sanity Not Included

```
val userAndRole = quote {  
    for {  
        u <- query[User]  
        ur <- query[UserToRole].join(so => so.userId == u.id)  
        r <- query[Role].join(r => r.id == ur.roleId)  
    } yield (u, r, p)  
}
```



**inline def userAndRole =**  
**quote { ... } => Query[(User,Role)]**

```
val userAndRoleAndPermission = quote {  
    for {  
        u <- query[User]  
        ur <- query[UserToRole].join(so => so.userId == u.id)  
        r <- query[Role].join(r => r.id == ur.roleId)  
        rp <- query[RoleToPermission].join(rp => rp.roleId == r.id)  
        p <- query[Permission].join(p => p.id == rp.roleId)  
    } yield (u, r, p)  
}
```



**inline def userAndRoleAndPermission =**  
**quote { ... } => Query[(User,Role,Permission)]**

```
val userAndRoleAndPermissionAndVenue = quote {  
    for {  
        u <- query[User]  
        ur <- query[UserToRole].join(so => so.userId == u.id)  
        r <- query[Role].join(r => r.id == ur.roleId)  
        rp <- query[RoleToPermission].join(rp => rp.roleId == r.id)  
        p <- query[Permission].join(p => p.id == rp.roleId)  
        v <- query[Venue].join(v => v.id == p.venueId)  
    } yield (u, r, p, v)  
}
```



**inline def userAndRoleAndPermissionAndVenue =**  
**quote { ... } => Q[(User,Role,Permission,Venue)]**



Is there a better way  
to do this???

---

WHEN THE ALTERNATIVE IS PURE MADNESS...

# Type-Level It!



—

WHEN THE ALTERNATIVE IS PURE MADNESS...

# Type-Level It!

```
trait Path[From, To]:  
    type Out  
    inline def get: Out
```



**Magic Is Here!**

—  
WHEN ALL ELSE FAILS...

# Type - Level It!

```
trait Path[User, Role]:  
    type Out = Query[(User, Role)]
```

```
trait Path[User, Permission]:  
    type Out = Query[(User, Role, Permission)]
```

```
trait Path[User, Venue]:  
    type Out = Query[(User, Role, Permission, Venue)]
```

```
inline given Path[User, Role]:
  type Out = Query[(User, Role)]
  inline def get: Query[(User, Role)] =
    for {
      u <- query[User]
      ur <- query[UserToRole].join(ur => ur.userId == s.id)
      r <- query[Role].join(r => r.id == ur.roleId)
    } yield (u, r)

inline given Path[User, Permission]:
  type Out = Query[(User, Role, Permission)]
  inline def get: Query[(User, Role, Permission)] =
    for {
      u <- query[User]
      ur <- query[UserToRole].join(ur => ur.userId == s.id)
      r <- query[Role].join(r => r.id == ur.roleId)
      rp <- query[RoleToPermission].join(rp => rp.roleId == r.id)
      p <- query[Permission].join(p => p.id == rp.roleId)
    } yield (u, r, p)

inline def path[F, T](using inline path: Path[F, T]): path.Out = path.get
```

—  
WHEN ALL ELSE FAILS...

# Type - Level It!

path[User, Role]: Query[(User, Role)]

path[User, Permission]: Query[(User, Role, Permission)]

path[User, Venue]: Query[(User, Role, Permission, Venue)]

WHEN ALL ELSE FAILS...

# Type - Level It!

```
inline def q: Quoted[Query[(User, Role)]] =  
  quote {  
    path[User,Role]  
  }
```

```
inline def q: Quoted[Query[(User, Role, Permission)]] =  
  quote {  
    path[User,Permission]  
  }
```

```
inline def q: Quoted[Query[(User, Role, Permission, Venue)]] =  
  quote {  
    path[User,Venue]  
  }
```

```
inline def q: Quoted[Query[(User, Role)]] =  
  quote {  
    path[User,Role]  
  }
```

```
inline def q: Quoted[Query[(User, Role, Permission)]] =  
  quote {  
    path[User,Permission]  
  }
```

```
inline def q: Quoted[Query[(User, Role, Permission, Venue)]] =  
  quote {  
    path[User,Venue]  
  }
```

```
val userAndRole = quote {  
  for {  
    u <- query[User]  
    ur <- query[UserToRole].join(so => so.userId == u.id)  
    r <- query[Role].join(r => r.id == ur.roleId)  
  } yield (u, r, p)  
}
```

```
val userAndRoleAndPermission = quote {  
  for {  
    u <- query[User]  
    ur <- query[UserToRole].join(so => so.userId == u.id)  
    r <- query[Role].join(r => r.id == ur.roleId)  
    rp <- query[RoleToPermission].join(rp => rp.roleId == r.id)  
    p <- query[Permission].join(p => p.id == rp.roleId)  
  } yield (u, r, p)  
}
```

```
val userAndRoleAndPermissionAndVenue = quote {  
  for {  
    u <- query[User]  
    ur <- query[UserToRole].join(so => so.userId == u.id)  
    r <- query[Role].join(r => r.id == ur.roleId)  
    rp <- query[RoleToPermission].join(rp => rp.roleId == r.id)  
    p <- query[Permission].join(p => p.id == rp.roleId)  
    v <- query[Venue].join(v => v.id == p.venueId)  
  } yield (u, r, p, v)  
}
```

NOTE: This Slide was skipped  
in the talk to save time.

```
inline def q: Quoted[Query[(User, Role)]] =  
  quote {  
    path[User,Role]  
      .filter(t => t._1.name == "Joe" && t._2.name == "Drinker")  
  }
```

```
inline def q: Quoted[Query[(User, Role, Permission)]] =  
  quote {  
    path[User,Permission]  
      .filter(t => t._1.name == "Joe" && t._2.name == "Drinker" && t._3.name == "Drink")  
  }
```

```
inline def q =  
  quote {  
    path[User,Venue]  
      .filter(t => t._1.name == "Joe" && t._2.name == "Drinker" && t._3.name == "Drink"  
        && t._4.name == "Divebar")  
  }
```

NOTE: This Slide was skipped  
in the talk to save time.

```
inline def q: Quoted[Query[(User, Role, Permission, Venue)]] =  
  quote {  
    path[User, Venue]  
    .filter(t => t._1.name == "Joe" && t._2.name == "Drinker" && t._3.name == "Drink"  
      && t._4.name == "Divebar")  
  }
```

```
SELECT  
  u.id, u.name, r.id, r.name, p.id, p.name, p.venueId, v.id, v.name  
FROM  
  User u  
  INNER JOIN UserToRole so ON so.userId = u.id  
  INNER JOIN Role r ON r.id = so.roleId  
  INNER JOIN RoleToPermission rp ON rp.roleId = r.id  
  INNER JOIN Permission p ON p.id = rp.roleId  
  INNER JOIN Venue v ON v.id = p.venueId  
WHERE  
  u.name = 'Joe' AND r.name = 'Drinker' AND p.name = 'Drink' AND v.name = 'Divebar'
```

NOTE: This Slide was skipped  
in the talk to save time.

```
inline def q =  
  quote {  
    path[User,Venue].filter { case (u, r, p, v) =>  
      u.name == "Joe" && r.name == "Drinker" && p.name == "Drink" && v.name == "Divebar"  
    }  
  }
```

```
SELECT  
  u.id, u.name, r.id, r.name, p.id, p.name, p.venueId, v.id, v.name  
FROM  
  User u  
  INNER JOIN UserToRole so ON so.userId = u.id  
  INNER JOIN Role r ON r.id = so.roleId  
  INNER JOIN RoleToPermission rp ON rp.roleId = r.id  
  INNER JOIN Permission p ON p.id = rp.roleId  
  INNER JOIN Venue v ON v.id = p.venueId  
WHERE  
  u.name = 'Joe' AND r.name = 'Drinker' AND p.name = 'Drink' AND v.name = 'Divebar'
```

NOTE: This Slide was skipped  
in the talk to save time.

```
inline def q =  
  quote {  
    path[User, Venue].filter((u, r, p, v) =>  
      u.name == "Joe" && r.name == "Drinker" && p.name == "Drink" && v.name == "Divebar"  
    )  
  }
```

```
SELECT  
  u.id, u.name, r.id, r.name, p.id, p.name, p.venueId, v.id, v.name  
FROM  
  User u  
  INNER JOIN UserToRole so ON so.userId = u.id  
  INNER JOIN Role r ON r.id = so.roleId  
  INNER JOIN RoleToPermission rp ON rp.roleId = r.id  
  INNER JOIN Permission p ON p.id = rp.roleId  
  INNER JOIN Venue v ON v.id = p.venueId  
WHERE  
  u.name = 'Joe' AND r.name = 'Drinker' AND p.name = 'Drink' AND v.name = 'Divebar'
```

NOTE: This Slide was skipped  
in the talk to save time.

```
inline def q =  
  quote {  
    path[User, Venue].filter((u, r, p, v) =>  
      u.name == "Joe" && r.name == "Drinker" && p.name == "Drink" && v.name == "Divebar"  
    )  
  }  
  
run(q)
```

```
sbt:dotty-simple> compile  
SELECT  
  u.id, u.name, r.id, r.name, p.id, p.name, p.venueId, v.id, v.name  
FROM  
  User u  
  INNER JOIN UserToRole so ON so.userId = u.id  
  INNER JOIN Role r ON r.id = so.roleId  
  INNER JOIN RoleToPermission rp ON rp.roleId = r.id  
  INNER JOIN Permission p ON p.id = rp.roleId  
  INNER JOIN Venue v ON v.id = p.venueId  
WHERE  
  u.name = 'Joe' AND r.name = 'Drinker' AND p.name = 'Drink' AND v.name = 'Divebar'  
[success] Total time: 7 s, completed Nov 23, 2020 12:16:08 AM
```

```

inline def q =
  quote {
    path[User,Role]
      .filter((u, r) =>
        u.name == "Joe" &&
        r.name == "Drinker")
  }
}

inline def q = quote {
  path[User,Permission]
    .filter((u, r, p) =>
      u.name == "Joe" &&
      r.name == "Drinker" &&
      p.name == "Drink")
}

inline def q = quote {
  path[User,Venue]
    .filter((u, r, p, v) =>
      u.name == "Joe" &&
      r.name == "Drinker" &&
      p.name == "Drink" &&
      v.name == "Divebar")
}

```

SELECT  
   u.id, u.name, r.id, r.name  
 FROM  
   User u  
   INNER JOIN UserToRole so ON so.userId = u.id  
   INNER JOIN Role r ON r.id = so.roleId  
 WHERE  
   u.name = 'Joe' AND r.name = 'Drinker'

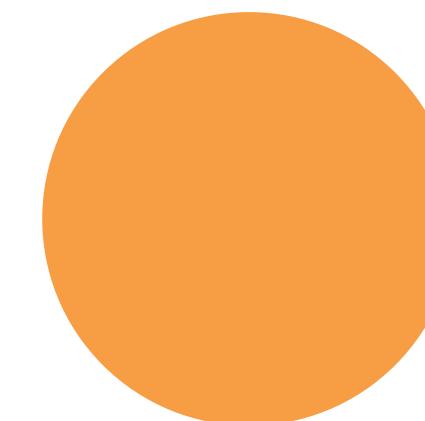
SELECT  
   u.id, u.name, r.id, r.name, p.id, p.name, p.venueId  
 FROM  
   User u  
   INNER JOIN UserToRole so ON so.userId = u.id  
   INNER JOIN Role r ON r.id = so.roleId  
   INNER JOIN RoleToPermission rp ON rp.roleId = r.id  
   INNER JOIN Permission p ON p.id = rp.roleId  
 WHERE  
   u.name = 'Joe' AND r.name = 'Drinker' AND p.name = 'Drink'

SELECT  
   u.id, u.name, r.id, r.name, p.id, p.name, p.venueId, v.id, v.name  
 FROM  
   User u  
   INNER JOIN UserToRole so ON so.userId = u.id  
   INNER JOIN Role r ON r.id = so.roleId  
   INNER JOIN RoleToPermission rp ON rp.roleId = r.id  
   INNER JOIN Permission p ON p.id = rp.roleId  
   INNER JOIN Venue v ON v.id = p.venueId  
 WHERE  
   u.name = 'Joe' AND r.name = 'Drinker' AND p.name = 'Drink' AND v.name = 'Divebar'

**How Far...  
Is just that little bit over there?**

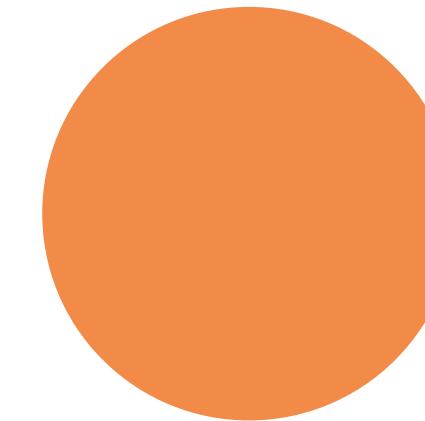


# Conclusions



## Inline is the new Implicit

In a good way! It will reveal many new applications  
in the years to come.



## Dotty Enables Quill to be Awesome!

Many new capabilities we've only dreamed about  
up to now!