TYPE-SAFE SQL QUERIES IN SCALA

Martin Kučera

supervised by Matt Bovel

April 6, 2023

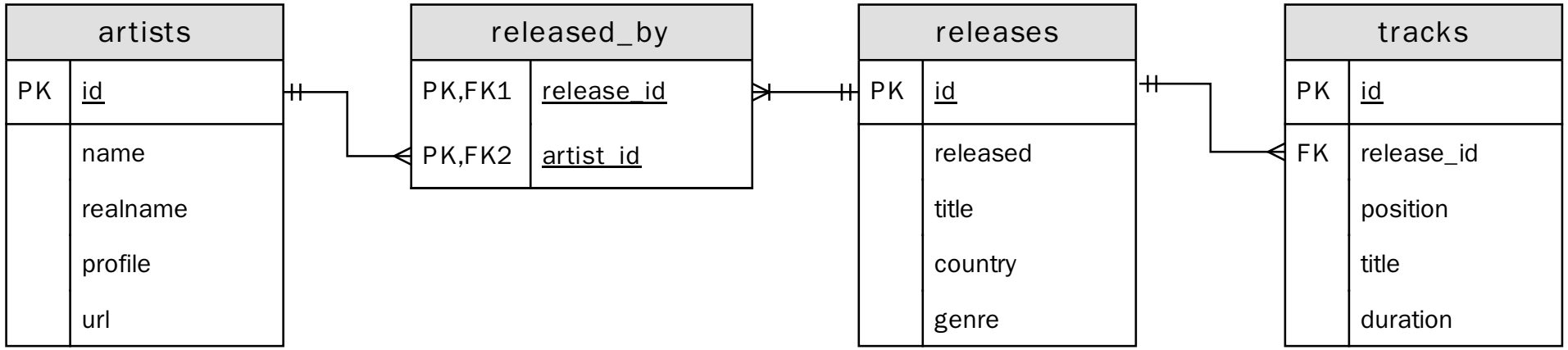
LINKS

These slides: go.epfl.ch/tyqu

GitHub: github.com/KuceraMartin/tyqu

Thesis WIP: go.epfl.ch/tyqu-thesis

DISCOGS DATABASE



MOTIVATION

WHY TYPE-SAFE SQL QUERIES?

```
import java.sql.*

val connection = DriverManager.getConnection("jdbc:postgresql://...")

val st = connection.createStatement()

val rs = st.executeQuery("SELECT 'Hello, world!'")
```

```
st.executeQuery("SELECT * FROM artists WHERE name >= 7")
```

```
org.postgresql.util.PSQLException: ERROR: operator does not exist:
character varying >= integer
  Hint: No operator matches the given name and argument types. You might
need to add explicit type casts.
  Position: 34
  at
org.postgresql.core.v3.QueryExecutorImpl.receiveErrorResponse(QueryExecuto
  at
org.postgresql.core.v3.QueryExecutorImpl.processResults(QueryExecutorImpl.
  at
org.postgresql.core.v3.QueryExecutorImpl.execute(QueryExecutorImpl.java:35
```

```
val rs = st.executeQuery("SELECT * FROM releases");

while rs.next() do
  val title = rs.getString("title")
  val pos = rs.getInt("position")
  println(s"$pos. $title ")
```

```
org.postgresql.util.PSQLException: Bad value for type int : A
  at org.postgresql.jdbc.PgResultSet.toInt(PgResultSet.java:3205)
  at org.postgresql.jdbc.PgResultSet.getInt(PgResultSet.java:2422)
  at org.postgresql.jdbc.PgResultSet.getInt(PgResultSet.java:2817)
  ... 32 elided
```

WHY ANOTHER LIBRARY?

STRUCTURAL REFINEMENTS

```
class Record(elems: (String, Any)*) extends Selectable:
  private val fields = elems.toMap
  def selectDynamic(name: String): Any = fields(name)

type Person = Record { val name: String; val age: Int }

val person = Record(
    "name" -> "Emma",
    "age" -> 42,
  ).asInstanceOf[Person]
```

Rethink Structural Types #1886

New issue

Closed **odersky** opened this issue on Jan 7, 2017 · 6 comments

odersky commented on Jan 7, 2017 • edited Contributor

Originally, structural types were introduced to make the language fit better the underlying foundations (type theorists prefer structural), and to emulate the idea of "duck typing" in dynamic languages but with static guarantees. But it turned out that almost nobody uses them. It seems that a combination of traits and classes, together with type classes represented by implicit parameters gives enough flexibility, so the need for duck typing is rarely felt.

However, there is another area where statically-typed languages are often more awkward than dynamically-typed ones: database access. In a dynamically typed language, it's quite natural to model a row as a record or object, and to select entries with simple dot notation, e.g. `row.columnName`. In a statically typed language. we can do that only if we somehow define a class for every possible row arising from a data-base manipulation (including rows arising from joins and projections), and set up a scheme to map between a row and the class representing it. This requires a lot of boilerplate code. So quite often one opts for a simpler scheme where column names are represented as strings that are passed to a select operator, e.g. `row.select("columnName")`. But this forgoes all the advantages of static typing and additionally is more awkward to write than the dynamically typed version.

A case in point is the Spark framework. The first version of Spark essentially supported distributed collections using RDDs. Used from Scala, this was very natural, an RDD was just some kind of collection, and was accessed in the same way as other Scala collections. Collection elements were defined by classes which were mapped transparently to database rows.

Assignees

No one assigned

Labels

itype:language enhancement

Projects

None yet

Milestone

No milestone

Development

No branches or pull requests

JOINS

In Slick:

```
tracks
  .join(releases).on(_.releaseId === _.id)
  .filter(_._2.title === "Californication")
  .map(_._1.title)
```

Or equivalently:

```
for
  t <- tracks
  r <- releases if t.releaseId === r.id
yield
  t.title
```

Vision:

```
tracks
  .filter(_.release.title === "Californication")
  .map(_.title)
```

EVEN BETTER TYPE SAFETY

e.g. group by in Quill:

```
releases.groupByMap(_.genre)(r => (r.genre, r.title))
```

READABLE SQL

TYQU

HOW TO DESCRIBE A SCHEMA

```
object Releases extends Table:
  val id = Column[Int](primary = true)
  val title = Column[String]()
  val country = Column[String]()
  val genre = Column[String]()

  lazy val artists = ManyToMany(target = Artists,
    joiningTable = ReleasedBy,
    sourceColumn = ReleasedBy.releaseId,
    targetColumn = ReleasedBy.artistId)
  lazy val tracks =
```

HOW TO WRITE QUERIES

```
val q = from(Tracks).map{ t => (
    t.title,
    t.position,
  ) }


for row <- q.execute() do
  println(s"${row.title} ${row.position}")
```

```
val q = from(Tracks).map{ t =>
    val fullTitle = (
          t.position + ". " + t.title + " (" + t.release.title + ")"
      ).as("fullTitle")
    (t.title, t.position, fullTitle)
  }


for row <- q.execute() do
  println(row.fullTitle)
```

```
val q = from(Releases)
  .filter(_.artists.exists(_.name === "Radiohead"))
  .filter(_.tracks.count < 5)
  .map(_.title)
  .sorted

q.execute().foreach(println)
```

```
val q = from(Artists)
  .filter(_.releases.flatMap(_.tracks).map(_.duration).sum >= 10000)
```

```
from(Tracks)
  .map{ t => (t.release.genre, t.duration) }
  .groupMap(_.genre){ r => (r.genre, r.duration.avg.as("avgDuration")) }
```

UNDER THE HOOD

```
def from[T <: Table](table: T) =
  val rel = FromRelation(table)
  val scope = TableScope(rel)
  QueryBuilder(scope, rel)
```

```
class QueryBuilder[S <: Scope](
  scope: S,
  from: FromRelation[?] | SubqueryRelation,
  where: Expression[Boolean] = NoFilterExpression,
  groupBy: List[Expression[?]] = List.empty,
  orderBy: List[OrderBy] = List.empty,
  limit: Option[Int] = None,
  offset: Int = 0,
):
  def map[S2 <: Scope](fn: S => S2): QueryBuilder[S2]
  def flatMap[S2 <: Scope](fn: S => QueryBuilder[S2]): QueryBuilder[S2]
```

```
inline transparent def map[Sc <: Scope, Tu <: Tuple, S2 <: (Sc | Tu)]
(inline fn: S => S2): QueryBuilder[?] =
  val (originalScope, newQb) = prepareMap
  QueryBuilderFactory.fromMap[S, Sc, Tu, S2](originalScope, newQb, fn)
```

```
// QueryBuilderFactory.fromMap (tuple case)
val selection = '{$fn($originalScope)}.asExprOf[Tu]
ScopeFactory.refine[Tu, TupleScope] match
case '[ScopeSubtype[t]] =>
    '{ $newQb.copy(scope =
TupleScope($selection)).asInstanceOf[QueryBuilder[t]] }
```

FUTURE WORK

- single-row queries
- async
- make ready for customizations
- DML (insert, update, delete)
- transactions
- generating:
  - DDL from schema
  - schema from DDL
  - migrations

CONCLUSION

- Tyqu
  - a type-safe SQL query builder
  - with convenient projections (i.e. `.map()`)
  - and convenient joins

- Achievements:
  - 3 reported issues in Dotty
  - 2 reported issues in scala-cli
  - 2 merged PRs in scala-cli
  - Accepted talk for Scala Days Madrid

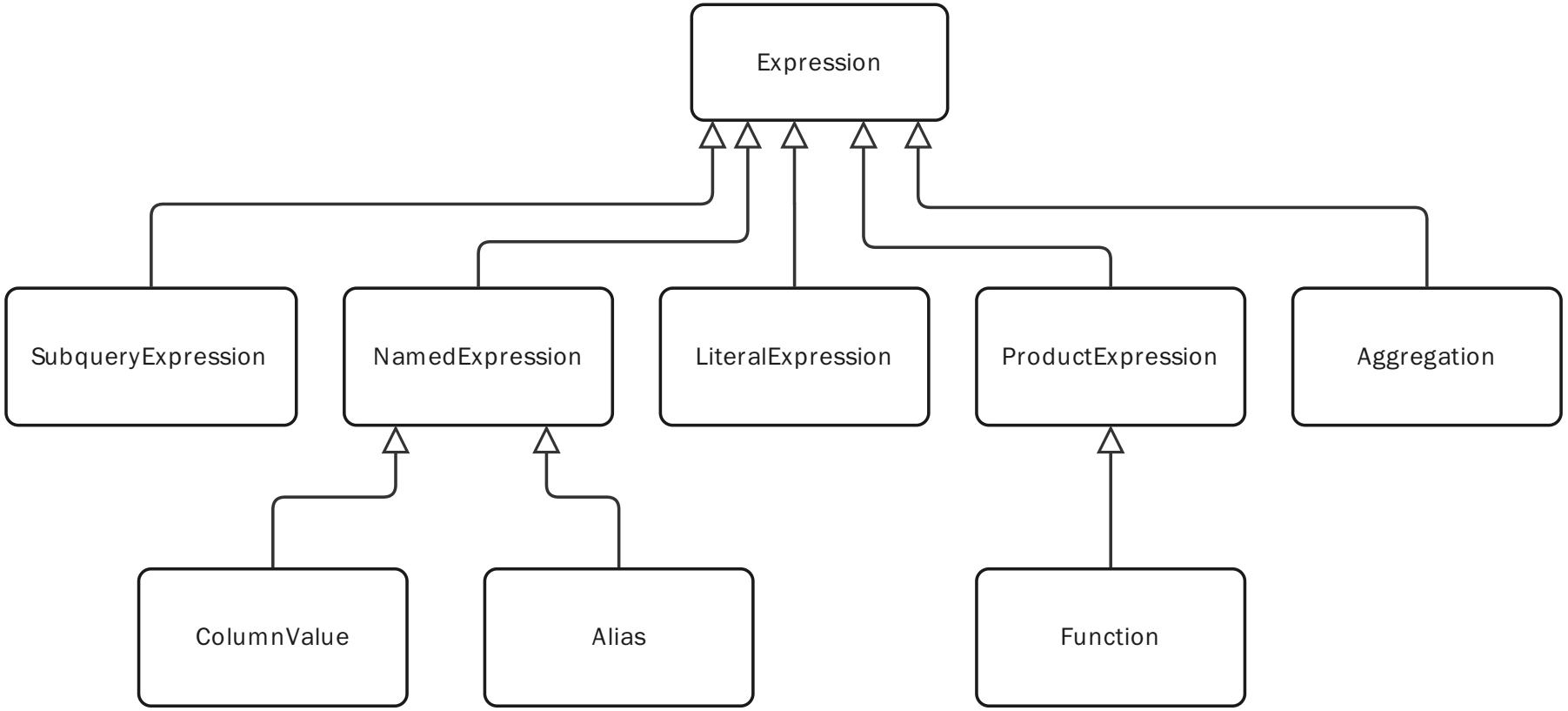
go.epfl.ch/tyqu

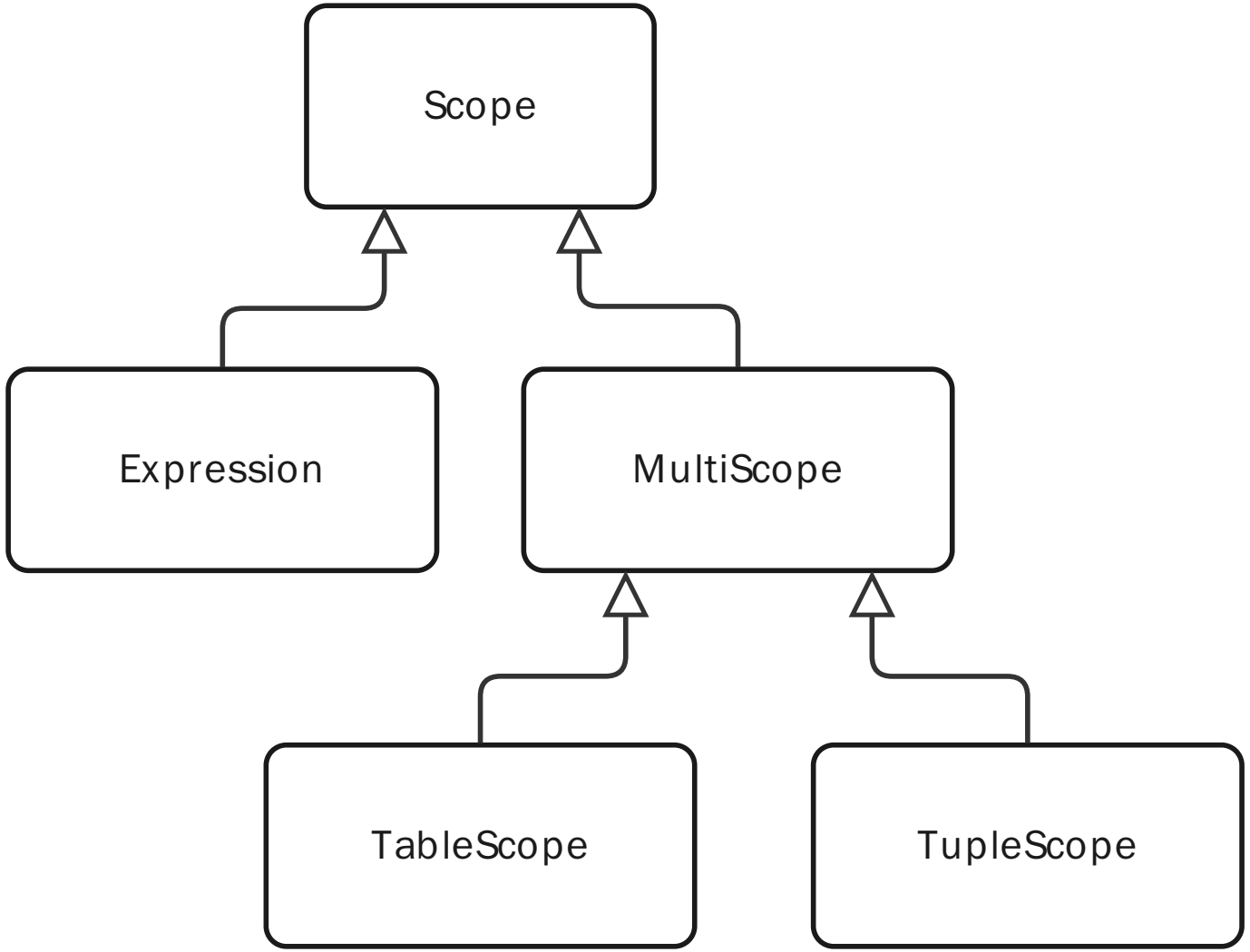
BACKUP SLIDES

AST EXAMPLE

```
from(Releases)
  .filter(r => r.genre === "Classical" && r.title.contains("Sonata"))
  .sortBy(_.title.asc)
  .limit(10)
```

```
QueryBuilder(
  scope = TableScope(releasesRelation),
  from = releasesRelation,
  where = And(
      Function("=", List(
        ColumnValue("genre", releasesRelation),
        LiteralExpression("Classical"),
      )),
      Contains("Sonata", ColumnValue("title", releasesRelation)),
    ),
  orderBy = List(Asc(ColumnValue("title", releasesRelation))),
```


Expression
SubqueryExpression
NamedExpression
LiteralExpression
ProductExpression
Aggregation
ColumnValue
Alias
Function


Scope
Expression
MultiScope
TableScope
TupleScope

RELATIONSHIPS

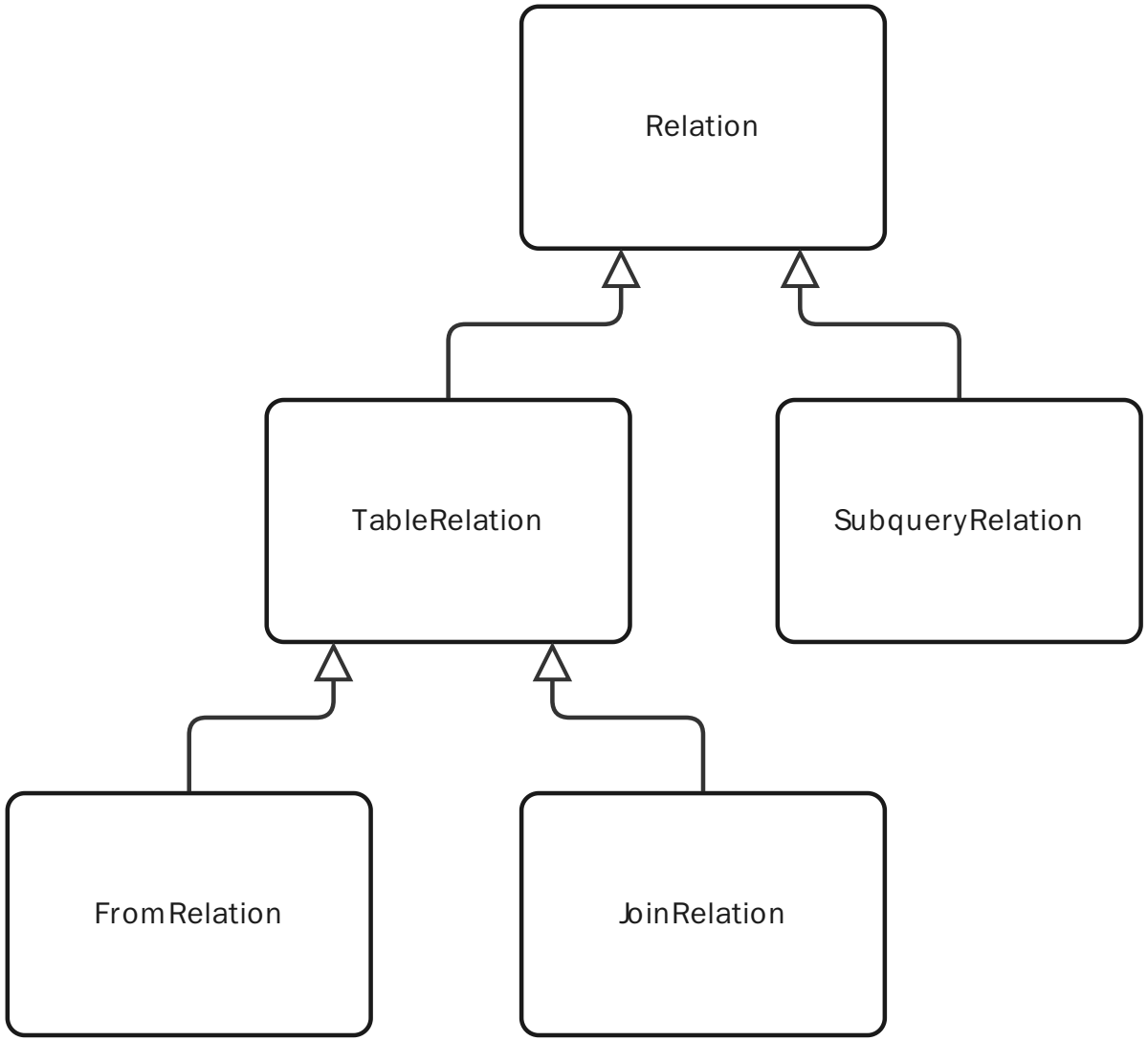
```
from(Releases)
  .map{ r => (
    r.title,
    from(Tracks)
      .filter(_.releaseId === r.id)
      .count
      .as("cnt"),
    ) }
  .limit(10)
```

```
class TableScope[T <: Table, Nullable <: Boolean](
  private[tyqu] val relation: TableRelation[T],
) extends MultiScope with Selectable:

  def selectDynamic(name: String): Any =
    relation.table.getClass.getMethod(name).invoke(relation.table) match

      case c: Column[?] =>
        relation.colToExpr(c)

      case OneToMany(sourceTable, ManyToOne(_, through)) =>
```


Relation
TableRelation
SubqueryRelation
FromRelation
JoinRelation

GROUP BY

```
abstract class Expression[T, CanSelect <: Boolean]

abstract class ProductExpression[T, Arguments <: Tuple | Expression[?,
?]] extends Expression[T, ArgsCanSelect[Arguments]]
```

```
inline transparent def groupMap
    [G <: (Tuple | Scope), Sc <: Scope, Tu <: Tuple, M <: (Sc | Tu)]
    (g: T => G)
    (using mapRef: GroupMapScope[G, T])
    (m: mapRef.Refined => M)
    (using IsValidMapResult[M] =:= true): QueryBuilder[?]
```