Formalisation of Relational Algebra and a SQL-like Language with the RISCAL Model Checker

Joachim Borya

Johannes Kepler Universität

January 26th, 2023

Outline

1. Topics
2. Recap on Relational Algebra
3. Implementation in RISCAL
4. Model Checking Algorithms and Theorems
5. A SQL-like wrapper language
6. Translation into standardized SQL
7. Current progress and further goals

Overview

- Formalization and verification of Relational Algebra with RISCAL
  - implementation of the common operations
  - model checking for proving correctness and theorems regarding the Relational Algebra
- Creation of a SQL-like language within RISCAL
- Demonstration by various examples

Recap on Relational Algebra

Definition

Let $n \in \mathbb{N}$ and $\mathcal{A} = \{A_1, \dots, A_n\}$ be an attribute schema. Then the set

$$\texttt{Row}_{\mathcal{A}} := \left\{ t : A \to \bigcup_{i=1}^{n} \mathrm{dom}(A_i) : t(A_i) \in \mathrm{dom}(A_i) \text{ for all } 1 \leq i \leq n \right\}$$

contains all tuples. Furthermore the set

$$\texttt{Relation}_{\mathcal{A}} := \{ r \subseteq \texttt{Row}_{\mathcal{A}} : |r| < \infty \}$$

contains all relations.

- Attributes are identifieres for columns. They are combined to get an attribute schema.
- Any attribute has a set of allowed values, a so called domain.
- The algebra consists of these types and some operations.

Recap on Relational Algebra

```
val M:ℕ; // maximum cardinality of relations
val N:ℕ; // maximum length of rows/tuples
val K:ℕ; // maximum number of tables
val D:ℕ; // maximum query depth
val L:ℕ; // maximum list length

type Element = ℕ[1];
type Attribute = ℕ[N-1];
type Length = ℕ[N];
type TableId = ℕ[K];
type Row = Map[Attribute, Element];
type Relation = Record[len:Length, tup:Set[Row]]
with |value.tup| ≤ M ∧ ∀ t:Row, i:Attribute. t ∈ value.tup ∧ i ≥ value.len ⇒ t[i] = 0;
type Database = Array[K, Relation];
```

Recap on Relational Algebra

- **Set operations**
- **Cartesian product**
- **Join:** Linking tables based on attributes
- **Selection:** Filtering on rows
- **Projection:** Filtering on columns

Implementation in RISCAL

Given an operator to implement:

1. Start off with a predicate representing the specification of an operation.
2. Rapid Prototyping: Choose a result that conforms to the predicate.
3. Algorithm: Implement it in terms of ...
    - RISCAL's high-level features like set constructors
    - and (only) loops, conditionals and variable assignments.

After the operations are verified we can check several theorems involving them.

Implementation in RISCAL

Definition

Let $\mathcal{A} = \{A_1, \dots, A_n\}$, $\mathcal{B} = \{B_1, \dots, B_m\}$ be relation schemas. The function $c : \texttt{Row}_{\mathcal{A}} \times \texttt{Row}_{\mathcal{B}} \to \texttt{Row}_{\mathcal{A} \cup \mathcal{B}}$ given by

$$c(t, u) := (t(A_1), \dots, t(A_n), u(B_1), \dots, u(B_m))$$

is called the concatenation of the rows t and u.

RISCAL

```
pred concat_spec(t:Row, t1:Row, t2:Row, n1:Length, n2:Length) ⇔
∀ i:Attribute. (
  if i < n1 then t[i] = t1[i]
  else if i ≥ n1 ∧ i < n1+n2 then t[i] = t2[i-n1]
  else t[i] = 0
);
```

Implementation in RISCAL

```
fun concat1(t1:Row, t2:Row, n1:Length, n2:Length):Row
requires n1 + n2 ≤ N;
= choose t:Row with concat_spec(t,t1,t2,n1,n2);
```

```
proc concat2(t1:Row, t2:Row, n1:Length, n2:Length):Row
requires n1 + n2 ≤ N;
ensures concat_spec(result,t1,t2,n1,n2); {
  var t:Row := Array[N,Element](0);
  for var i:Length:=0; i<n1; i:=i+1 do {
    t[i] := t1[i];
  }
  for var i:Length:=n1; i<n1+n2; i:=i+1 do {
    t[i] := t2[i-n1];
  }
  return t;
}
```

Implementation in RISCAL

Definition

Let $\mathcal{A} = \{A_1, \ldots, A_n\}$, $\mathcal{B} = \{B_1, \ldots, B_m\}$ be attribute schemas. The cartesian product $C : \texttt{Relation}_{\mathcal{A}} \times \texttt{Relation}_{\mathcal{B}} \to \texttt{Relation}_{\mathcal{A} \cup \mathcal{B}}$ is given by

$$C(r, s) := \{c(t, u) : t \in r \wedge u \in s\}.$$

RISCAL

```
pred cartesian_spec(r:Relation, r1:Relation, r2:Relation) ⇔
r.len = r1.len+r2.len ∧
∀ t:Row. t∈r.tup ⇔ ∃ t1:Row, t2:Row.
  t1∈r1.tup ∧ t2∈r2.tup ∧ t = concat1(t1,t2,r1.len,r2.len);
```

Implementation in RISCAL

```
fun cartesian1(r1:Relation, r2:Relation):Relation
requires r1.len+r2.len ≤ N ∧ |r1.tup|*|r2.tup| ≤ M;
  = choose r:Relation with cartesian_spec(r,r1,r2);
```

```
fun cartesian2(r1:Relation, r2:Relation):Relation
requires r1.len+r2.len ≤ N ∧ |r1.tup|*|r2.tup| ≤ M;
ensures cartesian_spec(result,r1,r2);
  = ⟨len: r1.len+r2.len, tup: {concat1(t1,t2,r1.len,r2.len) | t1∈r1.tup, t2∈r2.tup}⟩;
```

```
proc cartesian3(r1:Relation, r2:Relation):Relation
requires r1.len+r2.len ≤ N ∧ |r1.tup|*|r2.tup| ≤ M;
ensures cartesian_spec(result,r1,r2); {
  var q:Relation := ⟨len: r1.len+r2.len, tup: choose s:Set[Row] with |s|=0⟩;

  for t1 ∈ r1.tup do {
    for t2 ∈ r2.tup do {
      q.tup := q.tup ∪ {concat2(t1, t2, r1.len, r2.len)};
    }
  }

  return q;
}
```

Implementation in RISCAL

Definition

Let $\mathcal{A} = \{A_1, \ldots, A_n\}$, $\mathcal{B} = \{B_1, \ldots, B_m\}$ be attribute schemas. The function $\bowtie$: $\texttt{Relation}_{\mathcal{A}} \times \texttt{Relation}_{\mathcal{B}} \times \mathcal{A} \times \mathcal{B} \to \texttt{Relation}_{\mathcal{A} \cup \mathcal{B}}$ given by

$$\bowtie (r, s, A, B) := \{c(t, u) : t \in r \wedge u \in s \wedge t(A) = u(B)\}$$

is called the (equi-)join of r and s on A and B.

RISCAL

```
pred join_spec(s:Relation, r1:Relation, r2:Relation, n1:Attribute, n2:Attribute) ⇔
s.len = r1.len+r2.len ∧
∀ t:Row. t∈s.tup ⇔ ∃ t1:Row, t2:Row.
  (t1∈r1.tup ∧ t2∈r2.tup ∧ t = concat1(t1,t2,r1.len,r2.len) ∧ t1[n1] = t2[n2]);
```

Implementation in RISCAL

```
fun join1(r1:Relation, r2:Relation, n1:Attribute, n2:Attribute):Relation
requires n1<r1.len ∧ n2<r2.len ∧ r1.len+r2.len ≤ N ∧ |r1.tup|*|r2.tup| ≤ M;
  = choose s:Relation with join_spec(s,r1,r2,n1,n2);
```

```
fun join2(r1:Relation, r2:Relation, n1:Attribute, n2:Attribute):Relation
requires n1<r1.len ∧ n2<r2.len ∧ r1.len+r2.len ≤ N ∧ |r1.tup|*|r2.tup| ≤ M;
ensures join_spec(result,r1,r2,n1,n2);
  = ⟨len: r1.len+r2.len, tup:
    {concat1(t1,t2,r1.len,r2.len) | t1∈r1.tup, t2∈r2.tup with t1[n1] = t2[n2]}⟩;
```

```
proc join3(r1:Relation, r2:Relation, n1:Attribute, n2:Attribute):Relation
requires n1<r1.len ∧ n2<r2.len ∧ r1.len+r2.len ≤ N ∧ |r1.tup|*|r2.tup| ≤ M;
ensures join_spec(result,r1,r2,n1,n2); {
  var q:Relation := ⟨len: r1.len+r2.len, tup: choose s:Set[Row] with |s|=0⟩;

  for t1 ∈ r1.tup do {
    for t2 ∈ r2.tup do {
      if t1[n1] = t2[n2] then {
        q.tup := q.tup ∪ {concat2(t1, t2, r1.len, r2.len)};
      }
    }
  }

  return q;
}
```

Implementation in RISCAL

Definition

Let $\mathcal{A} = \{A_1, \ldots, A_n\}$ be an attribute schema. Then the function $\sigma : \texttt{Relation}_{\mathcal{A}} \times \mathcal{A} \times \bigcup_{A \in \mathcal{A}} \mathrm{dom}(A) \rightarrow \texttt{Relation}_{\mathcal{A}}$ given by

$$\sigma(r, A, a) := \{t \in r : t(A) = a\}$$

is called the selection of all rows t in r with value a at attribute A.

RISCAL

```
pred select_spec(s:Relation, r:Relation, a:Attribute, e:Element) ⇔
s.len = r.len ∧ ∀ t:Row. t∈s.tup ⇔ t∈r.tup ∧ t[a] = e;
```

Implementation in RISCAL

```
fun select1(r:Relation, a:Attribute, e:Element):Relation
requires a < r.len;
  = choose s:Relation with select_spec(s,r,a,e);
```

```
fun select2(r:Relation, a:Attribute, e:Element):Relation
requires a < r.len;
ensures select_spec(result,r,a,e);
  = ⟨len: r.len, tup: {t | t∈r.tup with t[a] = e}⟩;
```

```
proc select3(r:Relation, a:Attribute, e:Element):Relation
requires a < r.len;
ensures select_spec(result,r,a,e); {
  var q:Relation := ⟨len: r.len, tup: choose s:Set[Row] with |s|=0⟩;

  for t ∈ r.tup do {
    if t[a] = e then {
      q.tup = q.tup ∪ {t};
    }
  }
  return q;
}
```

Implementation in RISCAL

Definition

Let $\mathcal{A} = \{A_1, \ldots, A_n\}$ be an attribute schema. The operation $\pi : \texttt{Relation}_{\mathcal{A}} \times \mathcal{P}(\mathcal{A}) \to \bigcup_{\mathcal{A}' \subseteq \mathcal{A}} \texttt{Relation}_{\mathcal{A}'}$ defined by

$$\pi(r, \mathcal{B}) = \{t|_{\mathcal{B}} : t \in r\}$$

is called projection of r on $\mathcal{B}$.

RISCAL

```
pred project_spec(s:Relation, r:Relation, columns:Array[N,Length]) ⇔
s.len = |{i | i:Attribute with columns[i] ≠ N}| ∧ (∀ tr:Row. tr∈r.tup ⇒
  ∃ ts:Row. ts∈s.tup ∧ ∀ i:Attribute. i < s.len ⇒ ts[i]=tr[columns[i]]);
```

Implementation in RISCAL

```
fun project1(r:Relation, columns:Array[N,Length]):Relation
requires (∃ i:Attribute. ∀ j:Attribute. (j>i ⇒ columns[j] = N) ∧ (j≤i ⇒ columns[j] < r.
      len));
  = choose s:Relation with project_spec(s,r,columns);
```

```
proc project2(r:Relation, columns:Array[N,Length]):Relation
requires (∃ i:Attribute. ∀ j:Attribute. (j>i ⇒ columns[j] = N) ∧ (j≤i ⇒ columns[j] < r.
      len));
ensures project_spec(result,r,columns); {

  var l:Length := |{i | i:Attribute with columns[i] ≠ N}|;
  var q:Relation := ⟨len: l, tup: choose s:Set[Row] with |s|=0⟩;

  for t ∈ r.tup do {
    var tn:Row := Array[N,Element](0);

    var j:Length := 0;
    for var i:Length := 0; i<N; i:=i+1 do {
      if columns[i] ≠ N then {
        tn[j] := t[columns[i]];
        j := j+1;
      }
    }
    q.tup := q.tup ∪ {tn};
  }

  return q;
}
```

Model Checking Algorithms and Theorems

- For each function/procedure:
  - execute the function and
  - check the postcondition for every valid input
- For each theorem: Check validity

Both tasks can be accomplished by the RISCAL task "Execute operation".

Theorem 1

Let $\mathcal{A}$ be an attribute schema, $A \in \mathcal{A}$, $a \in \mathrm{dom}(A)$ and $r, s \in \texttt{Relation}_{\mathcal{A}}$ relations. Then

$$\sigma(r \gamma s, A, a) = \sigma(r, A, a) \gamma \sigma(s, A, a)$$

holds for all $\gamma \in \{\cup, \cap, \setminus\}$.

```
theorem select_union_equiv(r1:Relation, r2:Relation, a:Attribute, e:Element)
requires a < r1.len ∧ a < r2.len ∧ union_compatible(r1,r2) ∧ |r1.tup| + |r2.tup| ≤ M; ⇔
select2(rUnion(r1,r2),a,e) = rUnion(select2(r1, a, e), select2(r2, a, e));

theorem select_intersect_equiv(r1:Relation, r2:Relation, a:Attribute, e:Element)
requires a < r1.len ∧ a < r2.len ∧ union_compatible(r1,r2); ⇔
select2(rIntersect(r1,r2),a,e) = rIntersect(select2(r1, a, e), select2(r2, a, e));

theorem select_minus_equiv(r1:Relation, r2:Relation, a:Attribute, e:Element)
requires a < r1.len ∧ a < r2.len ∧ union_compatible(r1,r2); ⇔
select2(rMinus(r1,r2),a,e) = rMinus(select2(r1, a, e), select2(r2, a, e));
```

Theorem 2

Let $\mathcal{A}$ be an attribute schema, $A, B \in \mathcal{A}$, $a \in \mathrm{dom}(A), b \in \mathrm{dom}(B)$ and $r \in \texttt{Relation}_{\mathcal{A}}$ a relation. Then

$$\sigma(\sigma(r, B, b), A, a) = \sigma(r, A, a) \cap \sigma(r, B, b)$$

holds.

```
theorem select_intersect_comp(r:Relation, a:Attribute, e:Element, b:Attribute, f:Element)
requires a < r.len ∧ b < r.len; ⇔
select2(select2(r, a, e), b, f) = rIntersect(select2(r, a, e), select2(r, b, f));
```

Model Checking Algorithms and Theorems

Theorem 3

Let $\mathcal{A} = \{A_1, \ldots, A_n\}$ be an attribute schema, $A, B \in \mathcal{A}$ and $r \in \texttt{Relation}_{\mathcal{A}}$ a relation. Then

$$\pi(r, \{A, B\}) \subseteq C(\pi(r, \{A\}), \pi(r, \{B\}))$$

holds.

```
theorem project_cartesian_subset(r:Relation, a:Attribute, b:Attribute)
requires a < r.len ∧ b < r.len ∧ 2*|r.tup| ≤ M; ⇔
project2(r,attributes(List!node(a, List!node(b, List!nil)))).tup
  ⊆ cartesian2(project2(r, attributes(List!node(a, List!nil))),
          project2(r, attributes(List!node(b, List!nil)))).tup;
```

Theorem 3

Let $\mathcal{A}, \mathcal{B}$ be attribute schemas, $A \in \mathcal{A}$, $B \in \mathcal{B}$ and $r \in \texttt{Relation}_{\mathcal{A}}$, $s \in \texttt{Relation}_{\mathcal{A}}$ relations. Then

$$\bowtie (r, s, A, B) \subseteq C(r, s)$$

holds.

```
theorem join_cartesian_subset(r1:Relation, r2:Relation, n1:Attribute, n2:Attribute)
requires n1<r1.len ∧ n2<r2.len ∧ r1.len+r2.len ≤ N ∧ |r1.tup|*|r2.tup| ≤ M; ⇔
join3(r1, r2, n1, n2).tup ⊆ cartesian3(r1, r2).tup;
```

An SQL-like wrapper language

Grammar

$$
\begin{array}{lcl}
L & \in & \mathit{List} \\
Q & \in & \mathit{Query} \\
L & ::= & () \mid (A, L) \\
Q & ::= & \texttt{from}(T) \\
 & \mid & \texttt{on}(A_1, A_2, Q_1, Q_2) \\
 & \mid & \texttt{where}(A, E, Q) \\
 & \mid & \texttt{select}(L, Q) \\
 & \mid & \texttt{add}(Q_1, Q_2) \\
 & \mid & \texttt{inters}(Q_1, Q_2) \\
 & \mid & \texttt{minus}(Q_1, Q_2) \\
 & \mid & \texttt{cart}(Q_1, Q_2)
\end{array}
$$

- The natural variables T and A identify tables and attributes, whereas E denotes either 0 or 1.
- The language $\texttt{SQL}_{\text{min}}$ consists of all expressions in the syntactical domain `Query`.

An SQL-like wrapper language

```
rectype(L) List = nil | node(Attribute, List);
rectype(D) Query = from(TableId)
          | on(Attribute, Attribute, Query, Query)
          | where(Attribute, Element, Query)
          | select(List, Query)
          | add(Query, Query)
          | inters(Query, Query)
          | minus(Query, Query)
          | cart(Query, Query);
```

An SQL-like wrapper language

Definition

Let $\mathrm{db} : \mathtt{TableId} \to \bigcup_{i=1}^{n} \mathtt{Relation}_{\mathcal{A}_i}$ be a database. We define the denotational semantics $[\![\cdot]\!]_{\mathrm{db}} : \mathit{Query} \to \mathtt{Relation}_{\mathcal{A}}$ of $\mathtt{SQL}_{\mathsf{min}}$ by recursive induction in the following way:

$$
\begin{aligned}
[\![\mathtt{from}(T)]\!]_{\mathrm{db}} &:= \mathrm{db}(T) \\
[\![\mathtt{on}(A_1, A_2, Q_1, Q_2)]\!]_{\mathrm{db}} &:= \bowtie([\![Q_1]\!]_{\mathrm{db}}, [\![Q_2]\!]_{\mathrm{db}}, A_1, A_2) \\
[\![\mathtt{where}(A, E, Q)]\!]_{\mathrm{db}} &:= \sigma([\![Q]\!]_{\mathrm{db}}, A, E) \\
[\![\mathtt{select}(L, Q)]\!]_{\mathrm{db}} &:= \pi([\![Q]\!]_{\mathrm{db}}, L) \\
[\![\mathtt{add}(Q_1, Q_2)]\!]_{\mathrm{db}} &:= [\![Q_1]\!]_{\mathrm{db}} \cup [\![Q_2]\!]_{\mathrm{db}} \\
[\![\mathtt{inters}(Q_1, Q_2)]\!]_{\mathrm{db}} &:= [\![Q_1]\!]_{\mathrm{db}} \cap [\![Q_2]\!]_{\mathrm{db}} \\
[\![\mathtt{minus}(Q_1, Q_2)]\!]_{\mathrm{db}} &:= [\![Q_1]\!]_{\mathrm{db}} \backslash [\![Q_2]\!]_{\mathrm{db}} \\
[\![\mathtt{cart}(Q_1, Q_2)]\!]_{\mathrm{db}} &:= C([\![Q_1]\!]_{\mathrm{db}}, [\![Q_2]\!]_{\mathrm{db}})
\end{aligned}
$$

An SQL-like wrapper language

```
proc query(db:Database, q:Query):Relation {
  var r:Relation := match q with {
    from(tid:TableId) -> db[tid];
    on(n1:Attribute, n2:Attribute, q1:Query, q2:Query) -> join(query(db, q1), query(db,
         q2), n1, n2);
    where(a:Attribute, e:Element, q:Query) -> select(query(db, q), a, e);
    select(a:List, q:Query) -> project(query(db, q), attributes(a));
    add(q1:Query, q2:Query) -> rUnion(query(db, q1), query(db, q2));
    inters(q1:Query, q2:Query) -> rIntersect(query(db, q1), query(db, q2));
    minus(q1:Query, q2:Query) -> rMinus(query(db, q1), query(db, q2));
    cart(q1:Query, q2:Query) -> cartesian(query(db, q1), query(db, q2));
  };
  return r;
}
```

Translation into standardized SQL

A SQL query is always executed in the same order, i.e. we can sort the clauses in the way

$$\underset{4}{\texttt{FROM}} \succ \underset{3}{\texttt{ON}} \succ \underset{2}{\texttt{WHERE}} \succ \underset{1}{\texttt{SELECT}},$$

where $C_1 \succ C_2$ means that clause C_1 is evaluated before clause C_2. If the corresponding operations of $\texttt{SQL}_{\mathsf{min}}$ are composed in this order, they are equivalent to a single SQL query without subqueries.

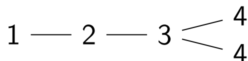
Translation into standardized SQL

We can list every possibility, because

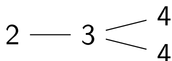
- the leaves of any AST must contain `FROM` clauses and
- under the assumption that the tree nodes of an AST contain clauses of strictly ascending priority, there are maximal 4 layers.

This equates to the eigth "atomic" SQL expressions.

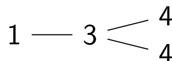
Translation into standardized SQL




```
SELECT <attributes>
FROM <table>
JOIN <table>
ON <attribute> = <attribute>
WHERE <attribute> = <value>;
```

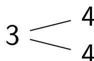



```
SELECT *
FROM <table>
JOIN <table>
ON <attribute> = <attribute>
WHERE <attribute> = <value>;
```

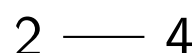



```
SELECT <attributes>
FROM <table>
JOIN <table>
ON <attribute>
  = <attribute>;
```

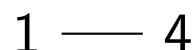
```
SELECT <attributes>
FROM <table>
WHERE <attribute> = <value>;
```




```
SELECT *
FROM <table>
JOIN <table>
ON <attribute> = <attribute>;
```




```
SELECT *
FROM <table>
WHERE <attribute>
= <value>;
```




```
SELECT <attributes>
FROM <table>;
```

```
SELECT *
FROM <table>;
```

Translation into standardized SQL

```
proc query_to_sql(q0:Query):()
proc translate_on(n1:Attribute, n2:Attribute, q0_1:Query, q0_2:Query):()
proc translate_cart(q0_1:Query, q0_2:Query):()
proc translate_where(a:Attribute, e:Element, q0:Query):()
proc translate_from(tid:TableId):()
proc translate_blank(q0:Query):()
proc translate_select(l:List, q0:Query):()
```

Translation into standardized SQL

```
proc query_to_sql(q0:Query):() {
  match q0 with {
    add(q1_1:Query, q1_2:Query) -> {
      translate_add(q1_1, q1_2);
    }
    inters(q1_1:Query, q1_2:Query) -> {
      translate_inters(q1_1, q1_2);
    }
    minus(q1_1:Query, q1_2:Query) -> {
      print "Not implemented";
    }
    cart(q1_1:Query, q1_2:Query) -> {
      translate_cart(q1_1, q1_2);
    }
    select(l:List, q1:Query) -> {
      translate_select(l, q1);
    }
    where(a:Attribute, e:Element, q1:Query) -> {
      print "SELECT *";
      translate_where(a, e, q1);
    }
    on(n1:Attribute, n2:Attribute, q1_1:Query, q1_2:Query) -> {
      print "SELECT *";
      translate_on(n1, n2, q1_1, q1_2);
    }
    from(tid:TableId) -> {
      print "SELECT *";
      translate_from(tid);
    }
  }
}
```

Translation into standardized SQL

<table>
<tr><th>RISCAL</th><th>query_to_sql</th><th>Sqlite</th></tr>
<tr><td>

```
Query!on(0,0,
  Query!select(
    List!node(0,
      List!node(2, List!nil)),
    Query!where(1,1,
      Query!from(0)
    )
  ),
  Query!from(1)
);
```

</td><td>

```
SELECT * FROM (
  SELECT 0, 2
  FROM 0
  WHERE 1 = 1
)
JOIN 1
ON 0 = 0
```

</td><td>

```
SELECT * FROM (
  SELECT "0"."0", "0"."2"
  FROM "0"
  WHERE "0"."1" = 1
) AS "_0"
JOIN "1"
ON "_0"."0" = "1"."0";
```

</td></tr>
</table>

Current progress and further goals

Done:

- Formalization and verification with a Model Checker
- Design of a Language

Work in progress:

- Theorem proving with RISCTP
- Various formal improvements