The Relational Algebra (RA) is a query language for the relational data model.

- The definition of RA is concise: the core of RA consist of five basic operators. More operators can be defined in terms of the core but this does not add to RA’s expressiveness.

- RA is expressive: all SQL (DML) queries as we have studied them in this course have an equivalent RA form (if we omit ORDER BY, GROUP BY, aggregate functions).

- RA is the original query language for the relational model. Proposed by E.F. Codd in 1970. Query languages that are as expressive as RA are called relationally complete. (SQL is relationally complete.)

- There are no RDBMSs that expose RA as a user-level query language but almost all systems use RA as an internal representation of queries.

- Knowledge of RA will help in understanding SQL, relational query processing, and the performance of relational queries in general (→ course “Datenbanksysteme II”).
THE RELATIONAL ALGEBRA

- RA is an algebra in the mathematical sense: an algebra is a system that comprises a
  1. a set (the carrier) and
  2. operations, of given arity, that are closed with respect to the set.
- Example: \((\mathbb{N}, \{\times, +\})\) forms an algebra with two binary (arity 2) operations.
- In the case of RA,
  1. the carrier is the set of all finite relations (= sets of tuples \(\Delta\)),
  2. the five operations are \(\sigma\) (selection), \(\pi\) (projection), \(\times\) (Cartesian product), \(\cup\) (set union), and \(\setminus\) (set difference).
- Closure: Any RA operator
  - takes as input one or two relations (the unary operators \(\sigma, \pi\) take additional parameters) and
  - returns one relation.
- Relations and operators may be composed to form complex expression (or queries).
Selection

If the unary selection operator $\sigma_p$ is applied to input relation $R$, the output relation holds the subset of tuples in $R$ that satisfy predicate $p$.

- **Example**: apply $\sigma_{A=1}$ (also consider: $\sigma_{B}$, $\sigma_{A=A}$, $\sigma_{C>20}$, $\sigma_{D=0}$) to relation $R$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>true</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>true</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>false</td>
<td>10</td>
</tr>
</tbody>
</table>

...to obtain

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>true</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>true</td>
<td>10</td>
</tr>
</tbody>
</table>

- Selection does *not* alter the input relation schema, i.e., $sch(\sigma_p(R)) = sch(R)$. 


RELATIONAL ALGEBRA: SELECTION ($\sigma$)

- **Predicate $p$ in $\sigma_p$ is restricted:** $p$ is evaluated for each input tuple in isolation and thus must exclusively be expressed in terms of
  1. **literals,**
  2. **attribute references** (occurring in $sch(R)$ of input relation $R$),
  3. **arithmetic, comparison** ($=, <, \leq, \ldots$), and **Boolean operators** ($\wedge, \vee, \neg$).
- In particular, quantifiers ($\exists, \forall$) or nested algebra expressions are **not** allowed in $p$.
- In PyQL, $\sigma_p(r)$ has the following implementation:

```python
def select(p, r):
    """Return those rows of relation r that satisfy predicate p."""
    return [row for row in r if p(row)]
```
- **select()**, and thus $\sigma$, are a higher-order functions.
**RELATIONAL ALGEBRA: PROJECTION ($\pi$)**

**Projection**

If the unary operator $\pi_\ell$ is applied to input relation $R$, it applies function $\ell$ to all tuples in $R$. The resulting tuples form the output relation.

- Function argument $\ell$ (the “projection list”) computes one output tuple from one input tuple. $\ell$ constructs new tuples that may
  1. **discard input attributes** (DB slang: “attributes are projected away”)
  2. **contain newly created output attributes** whose value is derived in terms of expressions over input attributes, literals, and pre-defined (arithmetic, string, comparison, Boolean) operators.

- Note:
  - $sch(\pi_\ell(R)) \neq sch(R)$ in general
  - $|\pi_\ell(R)| \leq |R| \quad \triangle$ (Why?)
RELATIONAL ALGEBRA: PROJECTION ($\pi$)

- PyQL implementation of $\pi_{\ell}$:

  ```python
def project(l, r):
      """Apply function l to relation r and return resulting relation."""
      return dupe([ l(row) for row in r ])  # dupe(): eliminate duplicate list elements
  
  - $\pi_{\ell}$ is a higher-order function (in functional programming languages, $\pi$ is known as map).
  
  - Common cases and notation for $\pi$ (refer to relation $R(A, B, C)$ shown above):
    
    - **Retain** some attributes of the input relation, i.e., *project (throw) away* all others:
      $\pi_{C,A}(R)$
    
    - **Rename** the attributes of the input relation, leaving their value unchanged:
      $\pi_{X\leftarrow A, Y\leftarrow B}(R)$
    
    - **Compute** new attribute values:
      $\pi_{X\leftarrow A+C, Y\leftarrow \neg B, Z\leftarrow \text{"LEGO"}}(R)$
RELATIONAL ALGEBRA: PROJECTION ($\pi$)

- Examples:

1. Apply $\pi_{X \leftarrow A + C, Y \leftarrow \neg B, Z \leftarrow "LEGO"}$ to relation $R$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>true</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>true</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>false</td>
<td>10</td>
</tr>
</tbody>
</table>

2. Apply $\pi_{X \leftarrow A, Y \leftarrow B}$ to obtain

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>false</td>
<td>LEGO</td>
</tr>
<tr>
<td>11</td>
<td>false</td>
<td>LEGO</td>
</tr>
<tr>
<td>12</td>
<td>true</td>
<td>LEGO</td>
</tr>
</tbody>
</table>

3. Apply $\pi_{X \leftarrow A, Y \leftarrow B}$ to obtain

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>false</td>
</tr>
</tbody>
</table>
**REL. ALGEBRA: CARTESIAN PRODUCT (×)**

**Cartesian Product**
If the binary operator \( × \) is applied to two input relations \( R_1, R_2 \), the output relation contains all possible concatenations of all tuples in both inputs.

- The schemata of inputs \( R_1 \) and \( R_2 \) must *not* share any attribute names. (This is no real restriction because we have \( π \).) We have:

\[
sch(R \times S) = sch(R) \cup sch(S) \quad |R \times S| = |R| \cdot |S|
\]

- PyQL implementation:

```python
def cross(r1, r2):
    """Return the Cartesian product of relations r1, r2."""
    assert(not(schema(r1) & schema(r2)))  # &: set intersection
    return [ row1 ⊕ row2 for row1 in r1 for row2 in r2 ]  # ⊕: dict merging
```

**Example:** Given graph adjacency (edge) relation $G(\text{from, to})$, compute the *paths of length two* ("Where can I go in two hops?"): 

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
</tr>
</tbody>
</table>

- Algebraic query:

$$\pi_{\text{from, to} \leftarrow \text{to} \leftarrow 2} (\sigma_{\text{to} = \text{from} \leftarrow 2} (G \times \pi_{\text{from} \leftarrow \text{from}, \text{to} \leftarrow 2}(G)))$$

- Result:

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
</tr>
</tbody>
</table>
The algebraic two-hop query relied on a combination of $\sigma \times$ that is typical: (1) generate all possible (arbitrary) combinations of tuples, then (2) filter for the interesting/sensible combinations.

**Join**

The **join** of input relations $R_1, R_2$ with respect to predicate $p$ is defined as

$$R_1 \bowtie_p R_2 := \sigma_p (R_1 \times R_2)$$

- **Note:**
  - Join does not add to the expressiveness of RA ($\bowtie_p$ is a derived operator or an “RA macro” in a sense).
  - $\bowtie_p$ comes with the same preconditions as $\sigma$ and $\times$: $sch(R_1) \cap sch(R_2) = \emptyset$ and $p$ may only refer to attributes in $sch(R_1) \cup sch(R_2)$. 

RDBMS are equipped with an entire family of algorithms that efficiently compute joins. In particular, the potentially large intermediate result (after \( \times \)) is not materialized.

Consider a join implementation in PyQL. Equational reasoning:

\[
\text{join}(p, r_1, r_2)
\]

# definition of join
\[
\equiv \text{select}(p, \text{cross}(r_1, r_2))
\]
# definition of cross
\[
\equiv \text{select}(p, [ \text{row}_1 \oplus \text{row}_2 \text{ for row}_1 \text{ in } r_1 \text{ for row}_2 \text{ in } r_2 ])
\]
# definition of select
\[
\equiv [ \text{row} \text{ for row} \text{ in } [ \text{row}_1 \oplus \text{row}_2 \text{ for row}_1 \text{ in } r_1 \text{ for row}_2 \text{ in } r_2 ] \text{ if } p(\text{row}) ]
\]
# \([ e_1(y) \text{ for y in } [ e_2(x) \text{ for x in xs } ] ] = [ \text{e}_1(y) \text{ for x in xs for y in [e}_2(x)\text{]} ]\]
\[
\equiv [ \text{row for row}_1 \text{ in } r_1 \text{ for row}_2 \text{ in } r_2 \text{ for row in } [\text{row}_1 \oplus \text{row}_2] \text{ if } p(\text{row}) ]
\]
# \([ e_1(y) \text{ for x in xs for y in [e}_2(x)\text{]} ] = [ \text{e}_1(\text{e}_2(x)) \text{ for x in xs } ]\)
\[
\equiv [ \text{row}_1 \oplus \text{row}_2 \text{ for row}_1 \text{ in } r_1 \text{ for row}_2 \text{ in } r_2 \text{ if } p(\text{row}_1 \oplus \text{row}_2) ]
\]
- Depict RA queries (or plans) as data-flow trees. Tuples flow from the leaves (relations) to the root which represents the final result.

- **Example:** Two-hop query using $\times$ and $\bowtie_p$:
Idiomatic relational database design often leads to joins between input relations $R_1, R_2$ in which the join predicate performs equality comparisons of attributes of the same name (think of key–foreign key joins).

Example: Consider the LEGO database:

- We have $sch(seats) \cap sch(contains) = \{\text{set}\}$ and attribute set exactly determines the join condition.

Associated RA key–foreign key join query:

$$
\pi_{\text{set}, \text{name}, \text{cat}, x, y, z, \text{weight}, \text{year}, \text{img}, \text{piece}, \text{color}, \text{extra}, \text{quantity}}(
sets \bowtie_{\text{set} = \text{set}_2} \pi_{\text{set} = \text{set}_2 \leftarrow \text{set}}.\text{piece}, \text{color}, \text{extra}, \text{quantity}(\text{contains}))
$$
**Natural Join**

The **natural join** of input relations $R_1, R_2$ performs a $\bowtie_p$ operation where $p$ is a conjunction of equality comparisons between the attributes $\{a_1, \ldots, a_n\} = sch(R_1) \cap sch(R_2)$:

$$R_1 \bowtie R_2 := \pi_{sch(R_1) \cup sch(R_2)}(R_1 \bowtie_{a_1 = a'_1 \land \ldots \land a_n = a'_n} \pi_{a'_1 \leftarrow a_1, \ldots, a'_n \leftarrow a_n, sch(R_2) \setminus \{a_1, \ldots, a_n\}}(R_2))$$

- Note: the final (top-most) projection ensures that the shared attributes $\{a_1, \ldots, a_n\}$ only occur once in the result schema (we have $a_i = a'_i$ anyway).

- Terminology:
  Joins $\bowtie_p$ with a conjunctive all-equalities predicates $p$ are also known as **equi-joins**. Otherwise, $\bowtie_p$ is also referred to as $\theta$-join (**theta-join**).
Natural Join Quiz: Consider relations

\[
\begin{array}{ccc}
A & B & C \\
1 & true & 20 \\
1 & true & 10 \\
2 & false & 10 \\
\end{array}
\]

\[
\begin{array}{ccc}
B & C & D \\
true & 20 & X \\
false & 10 & Y \\
true & 30 & Z \\
\end{array}
\]

and natural joins

1. \( R_1 \bowtie R_2 \)
2. \( \pi_{B,C}(R_1) \bowtie \pi_{B,C}(R_2) \)
3. \( R_1 \bowtie R_1 \)
4. \( \pi_{A,C}(R_1) \bowtie \pi_{B,D}(R_2) \)
Much like for the algebra \((\mathbb{N}, \{\ast, +\})\), the operators of RA obey laws, i.e. strict equivalences that hold regardless of the state of the input relations.

**RA Laws (△ Excerpt Only)**

- \(\bowtie\) (and \(\bowtie_p\)) are **associative** and **commutative**:
  \[(R_1 \bowtie R_2) \bowtie R_3 = R_1 \bowtie (R_2 \bowtie R_3) \text{ and } R_1 \bowtie R_2 = R_2 \bowtie R_1\]

- \(\sigma_p\) may be **pushed down** into \(\bowtie_q\), provided that … ⟨*fill in precondition*⟩ …:
  \[\sigma_p(R_1 \bowtie_q R_2) = R_1 \bowtie_q \sigma_p(R_2)\]

- \(\sigma_p\) and \(\sigma_q\) may be **merged**:
  \[\sigma_p(\sigma_q(R)) = \sigma_{p \land q}(R)\]

- Among these laws, **selection pushdown** is considered essential for query optimization. Why?
REL. ALGEBRA: A COMMON QUERY PATTERN

- This plan has the SQL equivalent:
Quiz: Find piece ID and name of all LEGO bricks that belong to a category related to animals.

Relevant relations:

- Algebraic plan:
Relations are sets of tuples. The usual family of binary **set operations** applies:

### Union (∪), Difference (\)

The binary set operations `∪` and `\` compute the **union** and **difference** of two input relations `R_1, R_2`. The schemata of the input relations must match:

\[
\text{sch}(R_1) \vdash \text{sch}(R_2) = \text{sch}(R_1 \cup R_2) = \text{sch}(R_1 \setminus R_2).
\]

The two set operations complete the operator core of RA (`σ, π, ×, ∪, \`). Any query language that is expressive as this core is **relationally complete**.

- Set **intersection** (∩) is not considered a core RA operator. There is more than one way to express intersection as an RA macro:
def union(r1, r2):
    """Return the union of relations r1, r2."""
    assert(matches(schema(r1), schema(r2)))
    return dupe([ row1 for row1 in r1 ] + [ row2 for row2 in r2 ])

def difference(r1, r2):
    """Return the difference of r1, r2 (a subset of r1)."""
    assert(matches(schema(r1), schema(r2)))
    return [ row1 for row1 in r1 if row1 not in r2 ]  # △ Note the negation (not)

More RA Laws

- ∪ is associative and commutative:
  \((R_1 \cup R_2) \cup R_3 = R_1 \cup (R_2 \cup R_3)\) and \(R_1 \cup R_2 = R_2 \cup R_1\)

- The empty relation \(\emptyset\) is the neutral element for ∪:
  \(R \cup \emptyset = \emptyset \cup R = R\)
In RA queries, \( \cup \) can straightforwardly express case distinction.

**Example:** Categorize LEGO sets according to their size (volume measured in stud\(^3\)).

<table>
<thead>
<tr>
<th>set name</th>
<th>cat</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>weight</th>
<th>year</th>
<th>img</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( \cup \) ∖ \( ∪ \)

\( \text{sets} \)

\( \Pi \text{set, name, vol, size } \) < 'small'

\( G \) vol < 1000

\( \Pi \text{set, name, vol, size } \) < 'medium'

\( G \) vol \( \geq \) 1000 \& vol < 10000

\( \Pi \text{set, name, vol, size } \) < 'large'

\( G \) vol \( \geq \) 10000

\( \text{sets} \)
The SQL expression `CASE...WHEN` implements case distinction ("multi-way if...else"): 

```sql
CASE WHEN ⟨condition₁⟩ THEN ⟨expression₁⟩
[WHEN …]
[ELSE ⟨expression₀⟩]
END
```

`CASE...WHEN` evaluates to the first ⟨expressionᵢ⟩ whose Boolean ⟨conditionᵢ⟩ evaluates to TRUE. If no WHEN clause is satisfied return the value of the fall-back ⟨expression₀⟩, if present (otherwise return NULL).

- Expression ⟨expressionᵢ⟩ never evaluated if ⟨conditionᵢ⟩ evaluates to FALSE.
- The types of the ⟨expressionᵢ⟩ must be convertible into a single output type.
The family of set operations is available in SQL as well. Since SQL operates over unordered lists (or: bags) of rows, modifiers control the inclusion/removal of duplicates:

**UNION, EXCEPT, INTERSECT**

The binary set (bag) operations connect two SQL SFW blocks. Schemata must match (columns of the same name have convertible types). Modifier DISTINCT (i.e. set semantics) is the default:

\[
\langle \text{SFW} \rangle \{ \text{UNION} \mid \text{EXCEPT} \mid \text{INTERSECT} \} \ [ \text{ALL} \mid \text{DISTINCT} ] \langle \text{SFW} \rangle
\]

Bag semantics (ALL) with \(m, n\) duplicate rows contributed by the two SFW blocks:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Duplicates in result</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNION ALL</td>
<td>(m + n)</td>
</tr>
<tr>
<td>EXCEPT ALL</td>
<td>(\max(m - n, 0))</td>
</tr>
<tr>
<td>INTERSECT ALL</td>
<td>(\min(m, n))</td>
</tr>
</tbody>
</table>
REL. ALGEBRA: SET OPERATIONS (∪, ∖)

- With ∪, ∖ (and ∩) now being available, we may be even more restrictive with respect to the admissable predicates $p$ in $σ_p$:

1. **literals, attribute references, arithmetics, comparisons** are OK,

2. the **Boolean connectives** ($∧$, $∨$, $¬$) are *not* allowed.

- $σ_{p∧q}(R) =$

- $σ_{p∨q}(R) =$

- $σ_{¬p}(R) =$
Monotonic Operators and Queries

An algebraic operator $\otimes$ is **monotonic** if a growing input relation implies that the output relation grows, too: $R \subseteq S \Rightarrow \otimes(R) \subseteq \otimes(S)$.

An RA **query is monotonic** if it exclusively uses monotonic operators.

```
def difference(r1, r2):
    : 
    return [row1 for row1 in r1 if row1 not in r2] # △ If r2 grows, result may shrink
```
RELATIONAL ALGEBRA: MONOTONICITY

- It follows that we **require the difference operator** whenever a non-monotonic query is to be answered:
  - “Find those LEGO sets that **do not contain** LEGO bricks with stickers.”
  - “Find those LEGO sets in which **all** bricks are colored yellow.”
  - “Find the LEGO set that is the **heaviest**.”

- More general: Typical **non-monotonic query problems** are those that …
  1. … check for the **non-existence** of an element in a set $S$,
  2. … check that a condition holds for all elements in a set $S$,
  3. … find an element that is **minimal/maximal** among all other elements in a set $S$.

⚠️ Note that cases 2 and 3 in fact indeed are instances of case 1. How?

- Insertion into $S$ may invalidate tuples that were valid query responses before $S$ grew.
REL. ALGEBRA: NON-MONOTONIC QUERY

“Find those LEGO sets that do not contain LEGO bricks with stickers.”

1. Identify the LEGO bricks with stickers. (bricks)
2. Find the sets that contain these bricks. (contains)
3. These are exactly those sets that do not interest us. (contains)
4. Attach set name and other set information of interest. (sets)
**REL. ALGEBRA: NON-MONOTONIC QUERY**

“Find those LEGO sets in which all bricks are colored yellow.”

<table>
<thead>
<tr>
<th>sets</th>
<th>name</th>
<th>cat</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>weight</th>
<th>year</th>
<th>img</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>contains</th>
<th>piece</th>
<th>color</th>
<th>extra</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td></td>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>colors</th>
<th>name</th>
<th>finish</th>
<th>rgb</th>
<th>from_year</th>
<th>to_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Yellow</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Query/problem indeed is non-monotonic:
  Insertion into relation ____________ can invalidate a formerly valid result tuple.
- Plan of attack resembles the *no stickers* query (see following slide).
**REL. ALGEBRA: NON-MONOTONIC QUERY**

“Find those LEGO sets in which all bricks are colored yellow.”

- ➊ Lookup the colors of the individual bricks (identify yellow in relation colors).
- ➋ Select the bricks that are *not* yellow.
REL. ALGEBRA: NON-MONOTONIC QUERY

“Find those LEGO sets in which all bricks are colored yellow.”

- ➊ Identify the sets that contain (at least one) such non-yellow brick.
- ➋ Among all sets ➌, the sets of ➌ are exactly those we are not interested in.
- ➎ Thus, return all other sets.
Join used as a filter: above \( \bowtie \), only attributes of \text{contains} are relevant.

**Semijoin**

The left semijoin \( \ltimes_p \) of input relations \( R_1, R_2 \) returns those tuples of \( R_1 \) for which at least one join partner in \( R_2 \) exists:

\[
R_1 \ltimes_p R_2 := \pi_{sch(R_1)}(R_1 \bowtie_p R_2)
\]

- \( \ltimes_p \) acts like a filter on \( R_1 \): \( R_1 \ltimes_p R_2 \subseteq R_1 \). Can be used to implement the semantics of existential quantification (\( \exists \)) in RA.
RELATIONAL ALGEBRA: ANTIJOIN ($\triangleright_p$)

(Left) Antijoin

The left antijoin $\triangleright_p$ of input relations $R_1, R_2$ returns those tuples of $R_1$ for which there does not exist any join partner in $R_2$:

$$R_1 \triangleright_p R_2 := R_1 \setminus (R_1 \bowtie_p R_2) = R_1 \setminus \pi_{sch(R_1)}(R_1 \bowtie_p R_2)$$

- $\triangleright_p$ can be used to implement the semantics of universal quantification:

$$R_1 \triangleright_p R_2 = \{ x \mid x \in R_1, \neg \exists y \in R_2 : p(x, y) \} = \{ x \mid x \in R_1, \forall y \in R_2 : \neg p(x, y) \}$$

- Example: Use self-left-antijoin on $S(A)$ to compute $\max(S)$:

$$S \triangleright_{A < A'} \pi_{A' \leftarrow A}(S) = \{ x \mid x \in S, \neg \exists y \in \pi_{A' \leftarrow A}(S) : x. A < y. A' \} = \{ x \mid x \in S, \forall y \in \pi_{A' \leftarrow A}(S) : x. A \geq y. A' \} = \{ \max(S) \}$$
RELATIONAL ALGEBRA: DIVISION (\(\div\))

- Certain query scenarios involving quantifiers can be concisely formulated using the derived RA operator **division** (\(\div\)):

**Division**

The relational division (\(\div\)) of input relation \(R_1(A, B)\) by \(R_2(B)\) returns those \(A\) values \(a\) of \(R_1\) such that **for every** \(B\) value \(b\) in \(R_2\) **there exists** a tuple \((a, b)\) in \(R_1\). Let \(s = sch(R_1) \setminus sch(R_2)\):

\[
R_1 \div R_2 := \pi_s(R_1) \setminus \pi_s((\pi_s(R_1) \times R_2) \setminus R_1)
\]

- Notes:
  - Schemata in general: \(sch(R_2) \subset sch(R_1)\) and \(sch(R_1 \div R_2) = sch(R_1) \setminus sch(R_2)\).
  - Division? Division!

  If \(R_1 \times R_2 = S\) then \(S \div R_1 = R_2\) and \(S \div R_2 = R_1\).
RELATIONAL ALGEBRA: DIVISION (÷)

- **Example:** Divide $R_1(A, B)$ by $R_2(B)$:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>c</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>
RELATIONAL ALGEBRA: OUTER JOIN

Find the bricks of LEGO Set 336–1 along with possible replacement bricks

contains

<table>
<thead>
<tr>
<th>set</th>
<th>piece</th>
<th>color</th>
<th>extra</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>s (= 336-1)</td>
<td>p₁</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

matches

<table>
<thead>
<tr>
<th>piece1</th>
<th>piece2</th>
<th>set</th>
</tr>
</thead>
<tbody>
<tr>
<td>p₁</td>
<td>p₂</td>
<td>s</td>
</tr>
</tbody>
</table>

bricks

<table>
<thead>
<tr>
<th>piece</th>
<th>name</th>
<th>cat</th>
<th>weight</th>
<th>img</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>pᵢ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Relation matches: In set s, piece p₂ is considered a replacement for original piece p₁.
- Query (∆ unexpectedly returns way too few rows...):

\[
\pi_{\text{piece}, \text{name}, \text{piece2}} \left( 
\pi_{\text{piece}, \text{name}}(\text{bricks}) \bowtie \pi_{\text{set}, \text{piece}}(\sigma_{\text{set}=336-1}(\text{contains})) \bowtie \pi_{\text{piece} \leftarrow \text{piece1}, \text{piece2}, \text{set}}(\text{matches}) \right) 
\]
RELATIONAL ALGEBRA: OUTER JOIN

(Left) Outer Join
The (left) outer join \( \bowtie_p \) of input relations \( R_1 \) and \( R_2 \) returns all tuples of the join of \( R_1, R_2 \) plus those tuples of \( R_1 \) that did not find a join partner (padded with \( \square \)). Let \( sch(R_2) = \{a_1, \ldots, a_n\} \):

\[
R_1 \bowtie_p R_2 := (R_1 \bowtie_p R_2) \cup ((R_1 \bowtie_p R_2) \times \{(a_1 : \square, \ldots, a_n : \square)\})
\]

- Notes:
  1. A Cartesian product with a singleton relation can conveniently implement the required padding.
  2. \( \bowtie_p \) is non-monotonic: insertion into \( R_1 \) or \( R_2 \) may invalidate a former \( \square \)-padded result tuple.
  3. The variants right (\( \bowtie \)) and full outer join (\( \bowtie \)) are defined in the obvious fashion.
Alternative SQL Join Syntax

In the FROM clause, two ⟨from_item⟩s may be joined using an RA-inspired syntax as follows:

\[
\text{⟨from_item⟩} \langle\text{join_type}\rangle \text{JOIN} \text{⟨from_item⟩} \\
\text{[ ON} \langle\text{condition}\rangle \text{]} \text{[ USING} \langle\text{column}\rangle [, …] \text{]} \text{]}
\]

where ⟨join_type⟩ is defined as

\[
\{ \text{[NATURAL]} \{ \text{[INNER]} | \{ \text{LEFT} | \text{RIGHT} | \text{FULL} \} \text{[OUTER]} \} | \text{CROSS} \}
\]

to indicate \(\bowtie\), \(\bowland\), \(\bowland\), \(\bowland\), and \(\times\) respectively.

- For all join types but CROSS, exactly one of NATURAL, ON, or USING must be specified.
- USING \((a_1, \ldots, a_n)\) abbreviates a conjunctive equality condition over the \(n\) columns.
SQL: USING OUTER JOIN

List all LEGO colors ordered by popularity (= number of bricks available in that color)

<table>
<thead>
<tr>
<th>color</th>
<th>name</th>
<th>finish</th>
<th>rgb</th>
<th>from_year</th>
<th>to_year</th>
<th>available_in</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Recall: the relationship between colors and bricks was described in the LEGO database ER diagram as follows, i.e., there may be unpopular colors that are not used (anymore):

  `[brick]–-(0,*)–⟨available in⟩–-(0,*)–[color]`

- Formulate the query with output columns: name, finish, popularity (= `⟨brick count⟩ ≥ 0`)...
  1. ... using SQL’s alternative join syntax,
  2. ... using all SQL constructs but the alternative join syntax.