INTRODUCTION TO

RELATIONAL DATABASE SYSTEMS

DATENBANKSYSTEME 1 (INF 3131)

Torsten Grust
Universität Tübingen
Winter 2021/22

THE RELATIONAL ALGEBRA (RA)

- The **Relational Algebra** 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 easily extended to cover these as well).
  - 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 *"DB2"*, summer semester 2022).

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:** ($\mathbb{N}$, {×,+}) 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 ⚠️),
  2. the five operations are σ (**selection**), π (**projection**), × (**Cartesian product**), ∪ (**set union**), and \ (**set difference**).
- **Closure:** Any RA operator
  - takes as input one or two relations (the unary operators σ, π take additional parameters) and
  - returns one relation.
- Relations and operators may be **composed** to form complex expression (or **queries**).

RELATIONAL ALGEBRA: SELECTION (σ)

Selection

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

- **Example:** apply $\sigma[A = 1]$ (and also: $\sigma[B]$, $\sigma[A = A]$, $\sigma[C > 20]$, ⚠️ $\sigma[D = 0]$) to relation

R

A	B	C
1	true	20
1	true	10
2	false	10

to obtain

A	B	C
1	true	20
1	true	10

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

RELATIONAL ALGEBRA: SELECTION (σ)

- **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** ($=$, $<$, $\leqslant$, …), 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:

```
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 σ, are a higher-order functions.

RELATIONAL ALGEBRA: PROJECTION (π)

Projection

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

- Function argument ℓ (the "*projection list*") computes one output tuple from one input tuple. ℓ 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
  - $|\pi[\ell](R)| \leqslant |R|$ ⚠️ (Why?)

RELATIONAL ALGEBRA: PROJECTION (π)

- PyQL implementation of π[ℓ](r):

```
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 elements
```

  - π[ℓ] is a higher-order function (in functional programming languages, π is known as map).
- Common cases and notation for π (refer to relation *R*(A,B,C) shown above):
  - **Retain** some attributes of the input relation, i.e., **project (throw) away** all others:
    π[C,A](*R*)
  - **Rename** the attributes of the input relation, leaving their value unchanged:
    π[X ← A, Y ← B, C](*R*), sometimes written as ϱ[X ← A, Y ← B](*R*) (ϱ: rho)
  - **Compute** new attribute values:
    π[X ← A + C, Y ← ¬B, Z ← "LEGO"](*R*)

RELATIONAL ALGEBRA: PROJECTION (π)

- **Examples:**

1. Apply π[X ← A + C, Y ← ¬B, Z ← "LEGO"] to relation R

A	B	C
1	true	20
1	true	10
2	false	10

   to obtain

X	Y	Z
21	false	LEGO
11	false	LEGO
12	true	LEGO

2. Apply π[X ← A, Y ← B] to obtain

X	Y
1	true
2	false

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:

$$sch(R_1 \times R_2) = sch(R_1) \uplus sch(R_2) \text{ and } |R_1 \times R_2| = |R_1| \times |R_2|$$

- PyQL implementation:

```
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
```

REL. ALGEBRA: CARTESIAN PRODUCT (×)

- **Example:** Given graph adjacency (edge) relation G(from,to), compute the **paths of length two** (“*Where can I go in two hops?*”):

from	to
A	B
B	A
B	C
A	D

  - Algebraic query:

$$\pi[\text{from},\text{to}\leftarrow\text{to2}](\sigma[\text{to} = \text{from2}](G \times \pi[\text{from2}\leftarrow\text{from},\ \text{to2}\leftarrow\text{to}](G)))$$

  - Result:

from	to
A	A
A	C
B	B
B	D

RELATIONAL ALGEBRA: JOIN ($\bowtie$)

- The algebraic *two-hop* query relied on a combination of σ-$\times$ 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 σ and $\times$: $sch(R_1) \cap sch(R_2) = \phi$ and p may only refer to attributes in $sch(R_1) \uplus sch(R_2)$.

RELATIONAL ALGEBRA: JOIN (⋈)

- RDBMS are equipped with an entire **family of algorithms that efficiently compute joins.** In particular, the potentially large intermediate result (after ×) is not materialized.
- Consider a join implementation in PyQL. Equational reasoning:

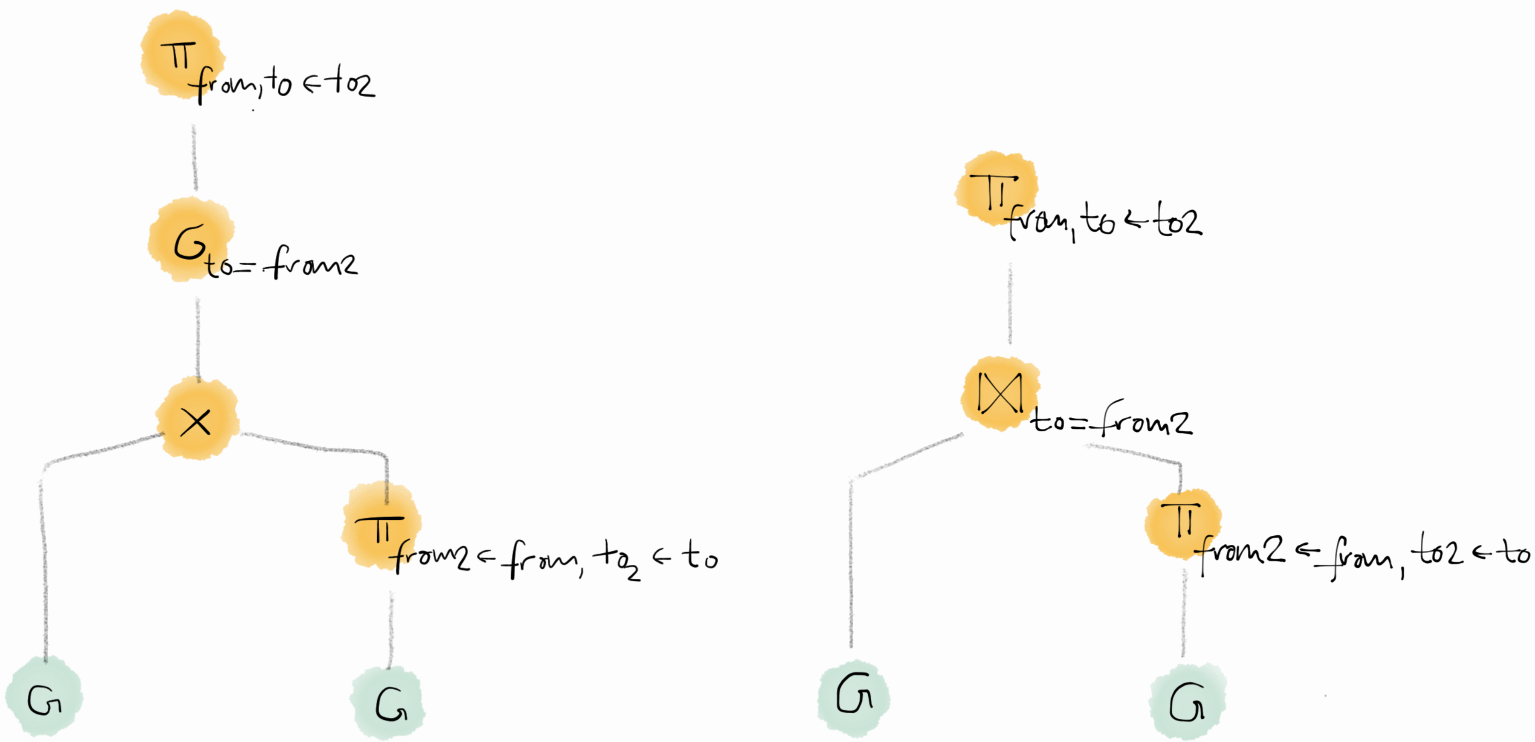
```
join(p, r1, r2)

 # definition of join
≡ select(p, cross(r1, r2))
 # definition of cross
≡ select(p, [ row1 ⊕ row2 for row1 in r1 for row2 in r2 ])
 # definition of select
≡ [ row for row in [ row1 ⊕ row2 for row1 in r1 for row2 in r2 ] if p(row) ]
 # [ e₁(y) for y in [ e₂(x) for x in xs ] ] = [ e₁(y) for x in xs for y in [e₂(x)] ]
≡ [ row for row1 in r1 for row2 in r2 for row in [row1 ⊕ row2] if p(row) ]
 # [ e₁(y) for x in xs for y in [e₂(x)] ] = [ e₁(e₂(x)) for x in xs ]

≡ [ row1 ⊕ row2 for row1 in r1 for row2 in r2 if p(row1 ⊕ row2) ]
```

PLANS (OPERATOR TREES)

- 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 × and ⋈[p]:



RELATIONAL ALGEBRA: NATURAL JOIN (⋈)

- 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:

sets

set	name	cat	x	y	z	weight	year	img

contains

set	piece	color	extra	quantity

  - We have $sch(\text{sets}) \cap sch(\text{contains}) = \{\text{set}\}$ and attribute set exactly determines the join condition.
  - Associated RA key-foreign key join query:

```
π[set,name,cat,x,y,z,weight,year,img,piece,color,extra,quantity](
   sets ⋈[set2 = set] π[set2←set,piece,color,extra,quantity](contains))
```

RELATIONAL ALGEBRA: NATURAL JOIN ($\bowtie$)

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 $\{a_1,\ldots,a_n\} = sch(R_1) \cap sch(R_2)$:

$$
\begin{aligned}
&R_1 \bowtie R_2 := \\
&\quad \pi[sch(R_1) \cup sch(R_2)](\\
&\quad\quad R_1 \bowtie[a_1=b_1 \wedge \cdots \wedge a_n=b_n]\ \pi[b_1{\leftarrow}a_1,\ldots,b_n{\leftarrow}a_n,sch(R_2) \setminus \{a_1,\ldots,a_n\}](R_2))
\end{aligned}
$$

- 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 $a_i=b_i$ 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 θ-join (**theta-join**).

RELATIONAL ALGEBRA: NATURAL JOIN (⋈)

- **Natural Join Quiz:** Consider relations

R_1

A	B	C
1	true	20
1	true	10
2	false	10

R_2

B	C	D
true	20	X
false	10	Y
true	30	Z

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. $\pi[A,C](R_1) \bowtie \pi[B,D](R_2)$

RELATIONAL ALGEBRA: LAWS

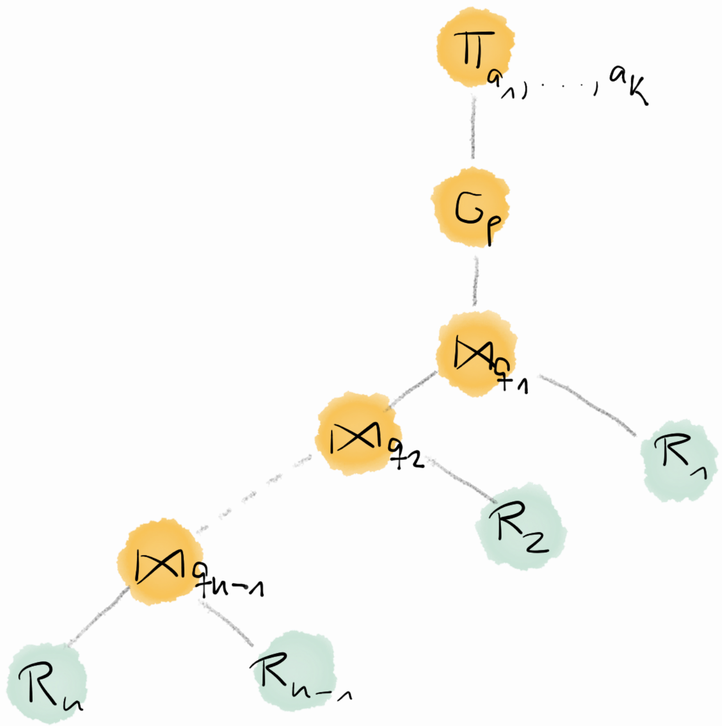
- Much like for the algebra $(\mathbb{N}, \{\times, +\})$, the operators of RA obey **laws**, i.e. strict equivalences that hold regardless of the state of the input relations.

RA Laws (⚠️ Excerpt Only)

- $\bowtie$ (and $\bowtie[p]$) are **associative** and **commutative:**
  $(R_1 \bowtie R_2) \bowtie R_3 = R_1 \bowtie (R_2 \bowtie R_3)$ 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:**
  $\sigma[p](\sigma[q](\mathrm{R})) = \sigma[p \wedge q](\mathrm{R})$

- Among these laws, **selection pushdown** is considered essential for query optimization. Why?

REL. ALGEBRA: A COMMON QUERY PATTERN



- This plan has the SQL equivalent:

RELATIONAL ALGEBRA: π–σ–⋈ QUERIES

- **Quiz:** Find piece ID and name of all LEGO bricks that belong to a category related to animals.
  - Relevant relations:

bricks

piece	name	cat	weight	img	x	y	z

categories

cat	name
	⋯Animal⋯

  - Algebraic plan:

REL. ALGEBRA: SET OPERATIONS (∪, \)

- 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:
>
> $$sch(R_1) = sch(R_2) = sch(R_1 \cup R_2) = sch(R_1 \setminus R_2).$$

- The two set operations complete the operator core of RA (σ, π, $\times$, $\cup$, $\setminus$). 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:

$$R_1 \cap R_2 :=$$

REL. ALGEBRA: SET OPERATIONS (∪, \)

```
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

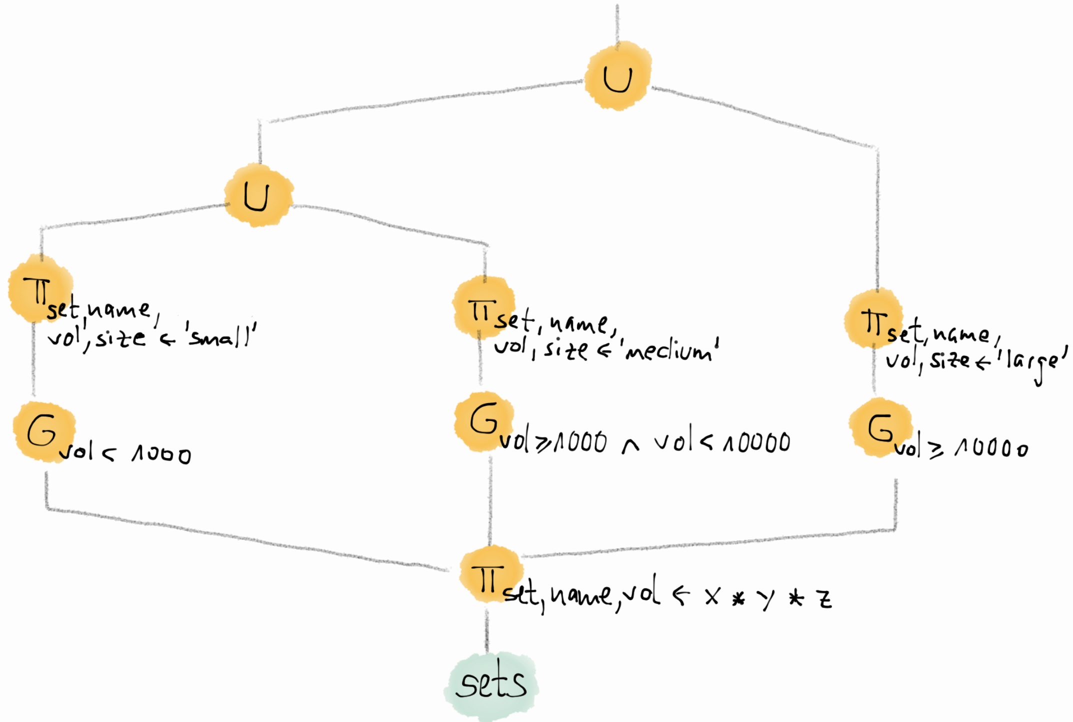
- ∪ is associative and commutative:
  $(R_1 \cup R_2) \cup R_3 = R_1 \cup (R_2 \cup R_3)$ and $R_1 \cup R_2 = R_2 \cup R_1$
- The empty relation ϕ is the neutral element for ∪:
  $R \cup \phi = \phi \cup R = R$

REL. ALGEBRA: SET OPERATIONS (∪, \)

- In RA queries, ∪ can straightforwardly express **case distinction.**
  - **Example:** Categorize LEGO sets according to their size (volume measured in stud³).

sets

<u>set</u>	name	cat	x	y	z	weight	year	img
			x	*y*	*z*			



SQL: CASE…WHEN (MULTI-WAY CONDITIONAL)

Conditional Expression (CASE…WHEN)

The SQL expression CASE...WHEN implements case distinction ("*multi-way if…else*"):

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

CASE...WHEN evaluates to the first *expression*$_i$ whose Boolean *condition*$_i$ evaluates to TRUE. If no WHEN clause is satisfied return the value of the fall-back *expression*$_0$, if present (otherwise return NULL).

- Expression *expression*$_i$ never evaluated if *condition*$_i$ evaluates to FALSE.
- The types of the *expression*$_i$ must be convertible into a single output type.

SQL: SET OPERATIONS

- 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 at the same position have convertible types). Modifier DISTINCT (i.e. set semantics) is the default:
>
> *SFW* { UNION | EXCEPT | INTERSECT } [ALL | DISTINCT] *SFW*

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

Operation	Duplicates in result
UNION ALL	$m + n$
EXCEPT ALL	$\max(m - n, 0)$
INTERSECT ALL	$\min(m, n)$

REL. ALGEBRA: SET OPERATIONS (∪, \)

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

1. **literals**, **attribute references**, **arithmetics**, **comparisons** are OK,
2. the **Boolean connectives** (∧, ∨, ¬) are *not* allowed.
   - $\sigma[p \wedge q](R)$ =

   - $\sigma[p \vee q](R)$ =

   - $\sigma[\neg p](R)$ =

RELATIONAL ALGEBRA: MONOTONICITY

Monotonic Operators and Queries

An algebraic operator ⊗ 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.

Operator (□: Input)	Monotonic?
$\sigma[p](\square)$	✓
$\pi[\ell](\square)$	✓
$\square \times$ _, _ $\times \square$	✓
$\square \cup$ _, _ $\cup \square$	✓
$\square \setminus$ _	✓
_ $\setminus \square$	✗

```
def difference(r1, r2):
  return [ row1 for row1 in r1 if row1 not in r2 ] # ⚠️ If |r2| ↑, |result| may ↓
```

RELATIONAL ALGEBRA: MONOTONICITY

- It follows that we **<u>require</u> 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.”

sets

set	name	cat	x	y	z	weight	year	img
s								

contains

set	piece	color	extra	quantity
s	*p*			

bricks

piece	name	cat	weight	img	x	y	z
p	…Sticker…						

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. (sets)
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.”

sets

set	name	cat	x	y	z	weight	year	img
s								

contains

set	piece	color	extra	quantity
s		*c*		

colors

color	name	finish	rgb	from_year	to_year
c	Yellow				

- 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.”

- ❶ Lookup the colors of the individual bricks (identify yellow in relation colors).
- ❷ Select the bricks that are *not* yellow.

1

set	color
s_1	
s_1	
s_2	
s_2	
s_3	
s_3	

2

set	color
s_1	
s_3	
s_3	

REL. ALGEBRA: NON-MONOTONIC QUERY

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

- ❸ Identify the sets that contain (at least one) such non-yellow brick.
- Among all sets ❹, the sets of ❸ are exactly those we are *not* interested in.
- ❺ Thus, return all other sets.

3

set
s_1
s_3

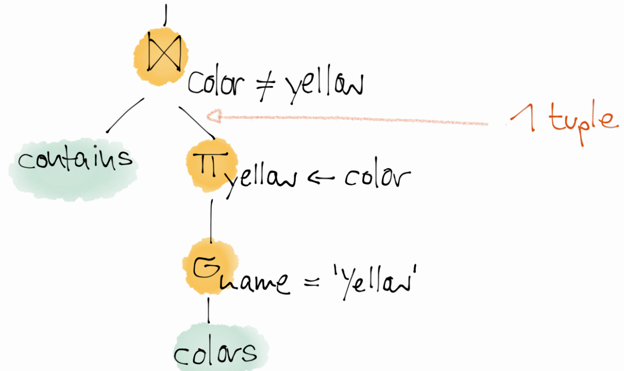
4

set
s_1
s_2
s_3

5

set
s_2

RELATIONAL ALGEBRA: SEMIJOIN (⋉)



Join used as a filter: above ⋈, only attributes of contains are relevant.

(Left) 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** (∃) in RA.

RELATIONAL ALGEBRA: ANTIJOIN (▷)

(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:

$$R_1 \triangleright[p]\ R_2 \quad := \quad R_1 \setminus (R_1 \ltimes[p]\ R_2) \quad = \quad R_1 \setminus \pi[sch(R_1)](R_1 \bowtie[p]\ R_2)$$

- $\triangleright[p]$ can be used to implement the semantics of **universal quantification:**

$$R_1 \triangleright[p]\ R_2 = \{x \mid x \in R_1,\ \neg\exists\ y \in R_2\colon p(x,y)\} = \{x \mid x \in R_1,\ \forall\ y \in R_2\colon \neg p(x,y)\}$$

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

$$\begin{aligned} S \triangleright[\mathrm{A} < \mathrm{A}']\ \pi[\mathrm{A}' \leftarrow \mathrm{A}](S) \quad &= \quad \{x \mid x \in S,\ \neg\exists\ y \in \pi[\mathrm{A}' \leftarrow \mathrm{A}](S)\colon x.\mathrm{A} < y.\mathrm{A}'\} \\ &= \quad \{x \mid x \in S,\ \forall\ y \in \pi[\mathrm{A}' \leftarrow \mathrm{A}](S)\colon x.\mathrm{A} \geqslant y.\mathrm{A}'\} \\ &= \quad \max[\mathrm{A}](\mathrm{S}) \end{aligned}$$

RELATIONAL ALGEBRA: DIVISION (÷)

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

Division

The **relational division** (÷) 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):

R_1

A	B
1	a
1	c
2	b
2	a
2	c
3	b
3	c
3	a
3	d

R_2

B
a
c

R_2

B
a
b
c

RELATIONAL ALGEBRA: OUTER JOIN

Find the bricks of LEGO Set 336 along with possible replacements

contains

set	piece	color	extra	quantity
s (= 336-1)	p_1			

bricks

piece	name	cat	weight	img	x	y	z
p_i							

replaces

piece1	piece2	set
p_1	p_2	s

- Relation replaces: In set s, piece p_2 is considered a replacement for original piece p_1 .
- Query (⚠️ unexpectedly returns way too few rows…):

```
π[piece,name,piece2](  π[piece,name](bricks)
                       ⋈ π[set,piece](σ[set='336-1'](contains))
                       ⋈ π[piece←piece1,piece2,set](replaces)))
```

RELATIONAL ALGEBRA: OUTER JOIN

(Left) Outer Join

The **(left) outer join** $⟕[p]$ 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 □). Let $sch(R_2) = \{a_1,\ldots,a_n\}$:

$$R_1 ⟕[p]\ R_2 := (R_1 \bowtie[p]\ R_2) \cup (\ (R_1 \triangleright[p]\ R_2) \times \{(a_1{:}\square,\ldots,a_n{:}\square)\}\ )$$

- Notes:

1. A Cartesian product with a singleton relation can conveniently implement the required padding.
2. $⟕[p]$ is non-monotonic: insertion into R_1 or R_2 may invalidate a former □-padded result tuple.
3. The variants **right** (⟖) and **full outer join** (⟗) are defined in the obvious fashion.

SQL: ALTERNATIVE JOIN SYNTAX

Alternative SQL Join Syntax

In the FROM clause, two *from_item* may be joined using RA-like syntax as follows:

```
from_item join_type JOIN from_item
  [ ON condition | USING (column [, …]) ]
```

where *join_type* is defined as

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

to indicate ⋈, ⟕, ⟖, ⟗, and × respectively.

- For all join types but CROSS: specify **exactly one** of NATURAL/ON/USING.
- USING (a_1,…,a_n) abbreviates a conjunctive equality condition over the n columns.

SQL: USING OUTER JOIN

Order *all* colors by popularity (# bricks available in that color)

colors

<u>color</u>	name	finish	rgb	from_year	to_year
c					

available_in

<u>color</u>	<u>brick</u>
c	

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

  [brick]—(0,*)—⟨available in⟩—(0,*)—[color]

- Formulate 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.