

# INTRODUCTION TO RELATIONAL DATABASE SYSTEMS

## DATENBANKSYSTEME 1 (INF 3131)

Torsten Grust  
Universität Tübingen

Summer 2023

# 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”).


# 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),
  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**).

# RELATIONAL ALGEBRA: SELECTION ( $\sigma$ )

## 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  $\sigma[A = 1]$  (also consider:  $\sigma[B]$ ,  $\sigma[A = A]$ ,  $\sigma[C > 20]$ ,   $\sigma[D = 0]$ ) 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 ( $\sigma$ )

- 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** ( $=$ ,  $<$ ,  $\leq$ , ...), 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  $\sigma$ , 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
  - $|\pi[\ell](R)| \leq |R|$  ⚠ (Why?)

# RELATIONAL ALGEBRA: PROJECTION ( $\pi$ )

- PyQL implementation of  $\pi[l](r)$ :

```
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
```

- $\pi[l]$  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:  
 $\pi[X \leftarrow A, Y \leftarrow B, C](R)$ , sometimes written as  $\rho[X \leftarrow A, Y \leftarrow B](R)$  ( $\rho$ : rho)
  - **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  $\pi[X \leftarrow A + C, Y \leftarrow \neg B, Z \leftarrow \text{"LEGO"}]$  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  $\pi[X \leftarrow A, Y \leftarrow B]$  to R to obtain

X	Y
1	true
2	false



# REL. ALGEBRA: CARTESIAN PRODUCT ( $\times$ )

## Cartesian Product

If the binary operator  $\times$  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  $\pi$ .) 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 ( $\times$ )

- **Example:** Given graph adjacency (edge) relation  $G(\text{from}, \text{to})$ , 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:

$\pi[\text{from}, \text{to} \leftarrow \text{to}_2](\sigma[\text{to} = \text{from}_2](G \times \pi[\text{from}_2 \leftarrow \text{from}, \text{to}_2 \leftarrow \text{to}](G)))$

- 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  $\sigma$ - $\times$  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  $\times$ :  $\text{sch}(R_1) \cap \text{sch}(R_2) = \emptyset$  and  $p$  may only refer to attributes in  $\text{sch}(R_1) \cup \text{sch}(R_2)$ .

# RELATIONAL ALGEBRA: JOIN ( $\bowtie$ )

- 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:

```

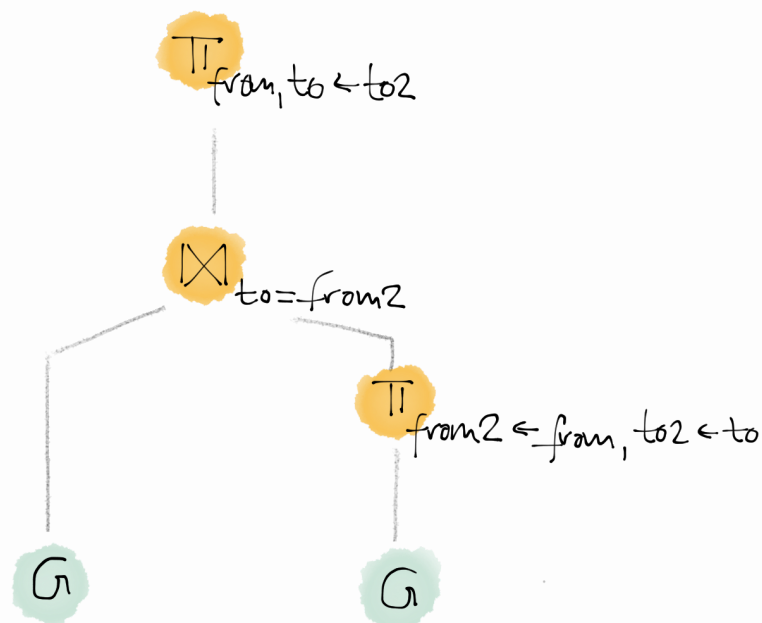
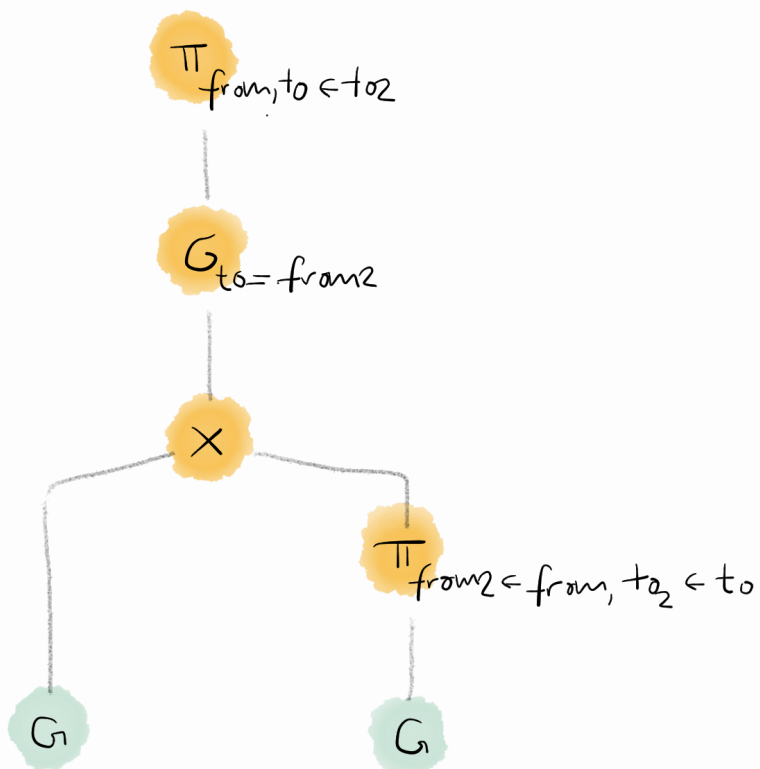
join(p, r1, r2)

# definition of join
≡ select(p, cross(r1, r2))
# definition of cross
≡ select(p, [ row1  $\oplus$  row2 for row1 in r1 for row2 in r2 ])
# definition of select
≡ [ row for row in [ row1  $\oplus$  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  $\oplus$  row2] if p(row) ]
# [ e1(y) for x in xs for y in [e2(x)] ] = [ e1(e2(x)) for x in xs ]
≡ [ row1  $\oplus$  row2 for row1 in r1 for row2 in r2 if p(row1  $\oplus$  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  $\times$  and  $\bowtie[p]$ :



# RELATIONAL ALGEBRA: NATURAL JOIN ( $\bowtie$ )

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

```
 $\pi[set, name, cat, x, y, z, weight, year, img, piece, color, extra, quantity]($ 
  sets  $\bowtie_{[set2 = set]}$   $\pi[set2+set, piece, color, extra, quantity](contains)$ )
```

# 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  $\{a_1, \dots, a_n\} = \text{sch}(R_1) \cap \text{sch}(R_2)$ :

$$R_1 \bowtie R_2 := \pi[\text{sch}(R_1) \cup \text{sch}(R_2)](R_1 \bowtie[a_1=b_1 \wedge \dots \wedge a_n=b_n] \pi[b_1 \leftarrow a_1, \dots, b_n \leftarrow a_n, \text{sch}(R_2) \setminus \{a_1, \dots, a_n\}](R_2))$$

- Note: the final (top-most) projection ensures that the shared attributes  $\{a_1, \dots, a_n\}$  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  $\theta$ -join (**theta-join**).

# RELATIONAL ALGEBRA: NATURAL JOIN ( $\bowtie$ )

- **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

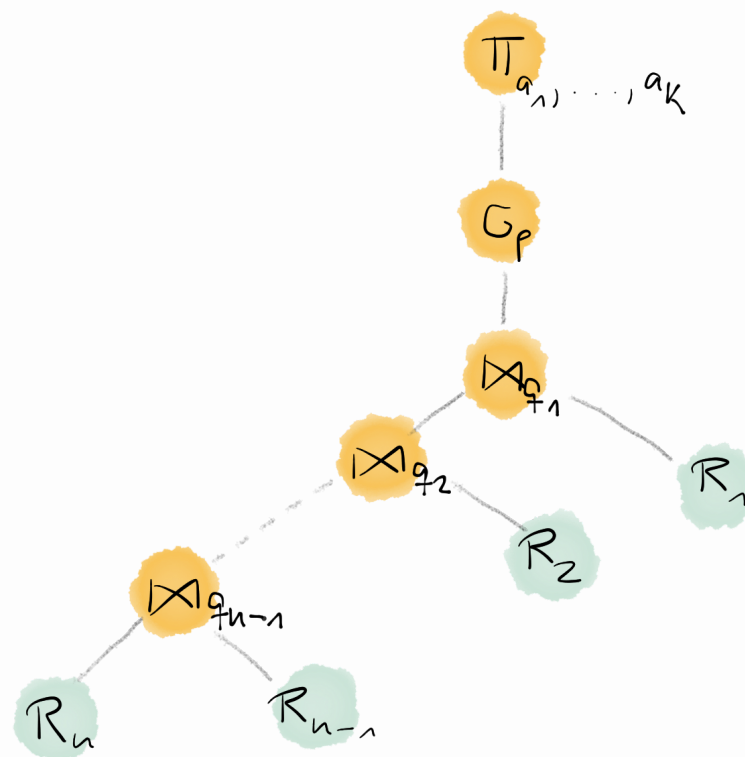
- Much like for the algebra  $(\mathbb{N}, \{\times, +\})$ , the operators of RA obey **laws**, i.e. strict equivalences that hold regardless of the state of the input relations.

## RA Laws (⚠ Excerpt Only)

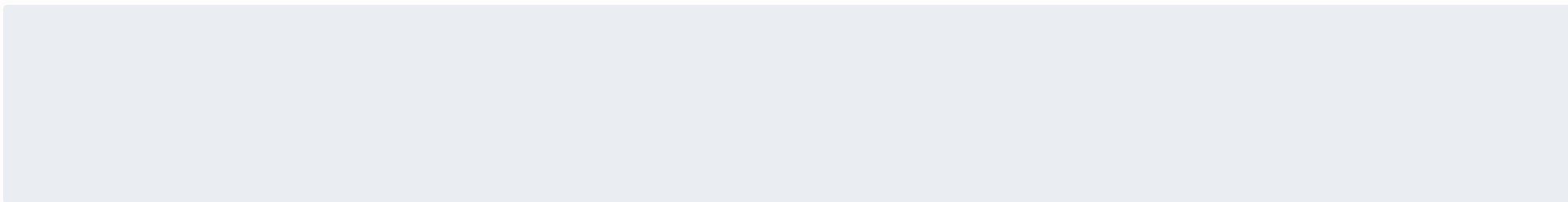
- $\bowtie$  (and  $\bowtie[p]$ ) are **associative** and **commutative**:  
 $(R_1 \bowtie R_2) \bowtie R_3 = R_1 \bowtie (R_2 \bowtie R_3)$  and  $R_1 \bowtie R_2 = R_2 \bowtie R_1$
- $\sigma[p]$  may be **pushed down** into  $\bowtie[q]$ , provided that ... *<fill in precondition> ...*:  
 $\sigma[p](R_1 \bowtie[q] R_2) = R_1 \bowtie[q] \sigma[p](R_2)$
- $\sigma[p]$  and  $\sigma[q]$  may be **merged**:  
 $\sigma[p](\sigma[q](R)) = \sigma[p \wedge q](R)$

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

# REL. ALGEBRA: A COMMON QUERY PATTERN



- This plan has the SQL equivalent:



# RELATIONAL ALGEBRA: $\pi$ - $\sigma$ - $\bowtie$ QUERIES

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

- Relevant relations:

bricks

<u>piece</u>	name	cat	weight	img	x	y	z

categories

<u>cat</u>	name
	...Animal...

- Algebraic plan:


# REL. ALGEBRA: SET OPERATIONS ( $\cup$ , $\setminus$ )

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

## Union ( $\cup$ ), Difference ( $\setminus$ )

The binary set operations  $\cup$  and  $\setminus$  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 ( $\sigma, \pi, \times, \cup, \setminus$ ). Any query language that is expressive as this core is **relationally complete**.
-  Set **intersection** ( $\cap$ ) 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 ( $\cup$ , $\setminus$ )

```
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

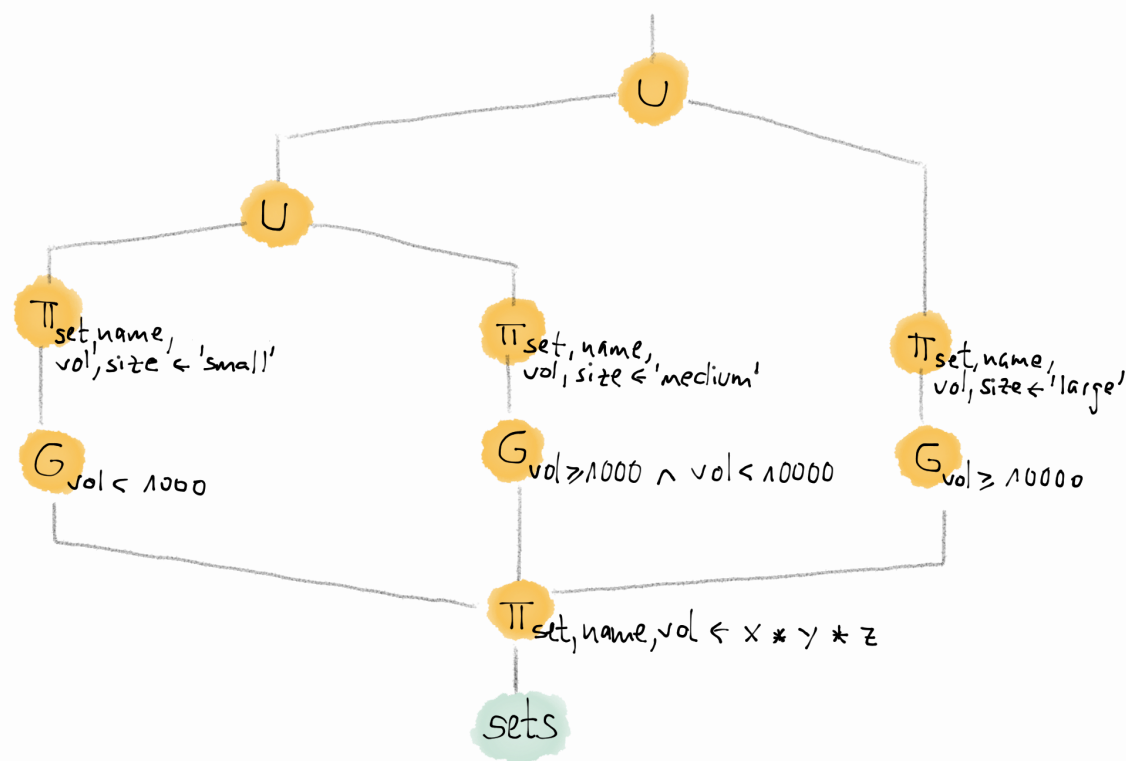
- $\cup$  is associative and commutative:  
 $(R_1 \cup R_2) \cup R_3 = R_1 \cup (R_2 \cup R_3)$  and  $R_1 \cup R_2 = R_2 \cup R_1$
- The empty relation  $\phi$  is the neutral element for  $\cup$ :  
 $R \cup \phi = \phi \cup R = R$

# REL. ALGEBRA: SET OPERATIONS ( $\cup$ , $\setminus$ )

- In RA queries,  $\cup$  can straightforwardly express **case distinction**.
- **Example:** Categorize LEGO sets according to their size (volume measured in  $\text{stud}^3$ ).

sets

set	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*<sub>*i*</sub> whose Boolean *condition*<sub>*i*</sub> evaluates to TRUE. If no WHEN clause is satisfied return the value of the fall-back *expression*<sub>0</sub>, if present (otherwise return NULL).

- Expression *expression*<sub>*i*</sub> never evaluated if *condition*<sub>*i*</sub> evaluates to FALSE.
- The types of the *expression*<sub>*i*</sub> 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 ( $\cup$ , $\setminus$ )

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

1. **literals, attribute references, arithmetics, comparisons** are OK,
2. the **Boolean connectives** ( $\wedge$ ,  $\vee$ ,  $\neg$ ) 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  $\otimes$  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.

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

```
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 **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.”

sets

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

contains

<u>set</u>	<u>piece</u>	<u>color</u>	<u>extra</u>	<u>quantity</u>
<i>s</i>	<i>p</i>			

bricks

<u>piece</u>	name	cat	weight	img	x	y	z
<i>p</i>	...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

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

contains

<u>set</u>	<u>piece</u>	<u>color</u>	<u>extra</u>	<u>quantity</u>
s		c		

colors

<u>color</u>	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.”

- ① Lookup the colors of the individual bricks (identify yellow in relation *colors*).
- ② Select the bricks that are *not* yellow.

1

set	color
S <sub>1</sub>	green
S <sub>1</sub>	yellow
S <sub>2</sub>	yellow
S <sub>2</sub>	yellow
S <sub>3</sub>	green
S <sub>3</sub>	red

2

set	color
S <sub>1</sub>	green
S <sub>3</sub>	green
S <sub>3</sub>	red

# REL. ALGEBRA: NON-MONOTONIC QUERY

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

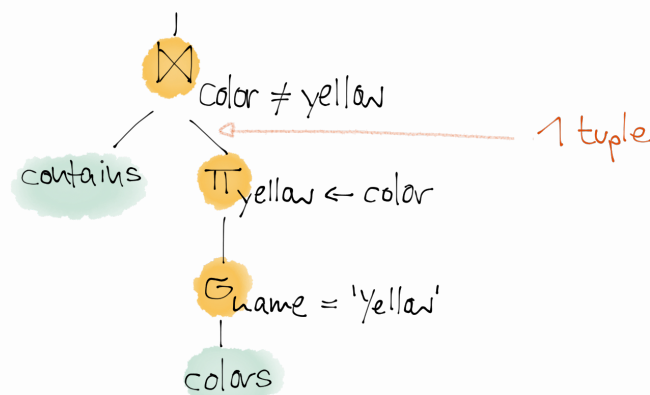
- ③ Identify the sets that contain (at least one) such non-yellow brick.
- Among all sets ④, the sets of ③ are exactly those we are *not* interested in.
- ⑤ Thus, return all other sets.

③  
set  
S<sub>1</sub>  
S<sub>3</sub>

④  
set  
S<sub>1</sub>  
S<sub>2</sub>  
S<sub>3</sub>

⑤  
set  
S<sub>2</sub>

# RELATIONAL ALGEBRA: SEMIJOIN ( $\bowtie$ )



Join used as a filter: above  $\bowtie$ , only attributes of contains are relevant.

## (Left) Semijoin

The **left semijoin**  $\bowtie[p]$  of input relations  $R_1, R_2$  returns those tuples of  $R_1$  for which at least one join partner in  $R_2$  **exists**:

$$R_1 \bowtie[p] R_2 := \pi[sch(R_1)](R_1 \bowtie[p] R_2)$$

- $\bowtie[p]$  acts like a **filter** on  $R_1$ :  $R_1 \bowtie[p] R_2 \subseteq R_1$ . Can be used to implement the semantics of **existential quantification** ( $\exists$ ) in RA.



# RELATIONAL ALGEBRA: ANTIJOIN ( $\triangleright$ )

## (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  $R_2$ :

$$R_1 \triangleright[p] R_2 := R_1 \setminus (R_1 \bowtie[p] R_2) = R_1 \setminus \pi[\text{sch}(R_1)](R_1 \bowtie[p] R_2)$$

- $\triangleright[p]$  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  $\max(S)$ :

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

# 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 ( $\div$ )

- **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**  $\bowtie[p]$  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  $\text{sch}(R_2) = \{a_1, \dots, a_n\}$ :

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

## - 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.

# 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  $\bowtie$ ,  $\bowtie$ ,  $\bowtie$ ,  $\bowtie$ , and  $\times$  respectively.

- For all join types but **CROSS**: specify **exactly one** of **NATURAL/ON/USING**.
- **USING** ( $a_1, \dots, a_n$ ) 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
<i>c</i>					

available\_in

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

- 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* ( $\equiv$  <brick count>  $\geq 0$ ) ...
  1. ... using SQL's alternative join syntax,
  2. ... using all SQL constructs *but* the alternative join syntax.