TDA357/DIT621 – Databases

Lecture 12 – Relational Algebra

Jonas Duregård

What is an algebra?

- An algebra is a set of values, and a collection of operations on those values
- Formulas built from those operations (and constants) are called expressions
- Example: The set of natural numbers and the operations addition and multiplication form a tiny algebra
  - Expressions are arithmetic expression like 5+3*2
  - The result of every expression (and subexpressions like 3*2 here) is also a natural number
- Another example: Boolean algebra has 2 values and operators like AND, OR …
  - SQL logic has 3 values though… (FALSE, TRUE and UNKNOWN)
- We can also have variables in our expressions like x+3

What is relational algebra (RA)?

- An algebra on the infinite set of relations, and operations like Cartesian product, union, etc.
- Relational algebra expressions are essentially queries (but not in SQL)
- Just like arithmetic and Booleans, this algebra is *closed* under its operations
  - If I apply addition to two numbers, I get a number
  - If I apply AND to two Booleans, I get a Boolean
  - If I apply Cartesian product to two relations, I get a relation

Relational algebra

- Our goal today is to define operations in relational algebra, that allow us to write expressions corresponding to most SQL queries
- There are at least two advantages to using Relational Algebra over SQL:
  - Reasoning: We can use hundreds of years of mathematical results and methods to prove that our queries do what we intend for them to do
  - Simplification: Similarly to how we can simplify (a+b*0+a) to (2*a), we can sometimes simplify complicated relational algebra expressions
    - Uses proven simplification rules
    - Can be used to make queries faster

Query optimization in practice

- There are often different ways of writing queries to solve a particular task
- A query optimizer is a part of a DBMS that tries to transform each query into its most efficient form, often (but not always) transforming equivalent queries into the same form
- This allows users to write queries the way they find most intuitive, and rely on the DBMS to deal with efficiency
- Also makes it hard to answer "which of these SQL queries is most efficient", since the answer is always "depends on what the query optimizer does"
- Query optimizers are based on relational algebra

What exactly is a relation?

- The first thing you need to do when defining an algebra, is define the set of values it operates on
- Good enough informal definition for relational algebra: Relations are tables.
- Slightly more formal: A relation is a schema (relation name + attribute list) and a collection of tuples, such that all tuples match the schema
- Typically we abstract away the tuples, focusing on the structure/schema of the relation when writing relational algebra expressions
- Some things that are not quite standardized for relational algebra:
    - Controversy 1: Is the collection a set, a bag or a list?
    - Controversy 2: How does naming work? Are there qualified names?
    - We will deal with these issues as we encounter them

It's all Greek!

- For historical reasons, operators in relational algebra use Greek letters
- Some symbols that everyone knows like π (pi)
- Some less familiar ones like ρ (rho)
- May take some getting used to if you do not write Greek on a regular basis

Projection – Our first RA operator!

- The π (pi) operator corresponds to the SELECT clause in SQL
- Syntax: $\pi_{\text{<attribute list>}}(R)$, where R is any relational algebra expression
- In SQL: `SELECT <attribute list> FROM (<SQL for R>);`

- Example: $\pi_{\text{id,name}}(\text{Students})$
- In SQL: `SELECT id,name FROM Students;`

- Called the projection operator (we project a certain view of the relation)

Sets, bags or lists? (Again)

- Remember: A set has no duplicates or internal ordering, bags allow duplicates, lists allow duplicates and each value has a position
- Traditionally, relations are considered sets of tuples in relational algebra
- This makes them harder to translate to/from SQL where results are bags
- There are also things like sorting operators in most Relational Algebra definitions, which is not compatible with either sets or bags
- In this course we use bag semantics
    - Semantics ≈ what expressions mean, as opposed to how they look (syntax)
    - You will need to understand the implications of this choice

Projection on sets/bags

- Projection is one of the operators where set/bag semantics differ
- The intuition of projection is that you just remove a few attributes
- If using set semantics, the number of tuples/rows may decrease, because duplicates are introduced when removing the attributes!
- One way to explain this in terms of SQL:
  - With bag semantics, projection corresponds to the SELECT clause
  - With set semantics, projection corresponds to SELECT DISTINCT
- In this course, we follow the intuition and use bag semantics for π

Table: WL

student	course	position
Student1	TDA357	1
Student2	TDA357	2
Student1	TDA143	1

$\pi_{student}(WL)$

set semantics

student
Student1
Student2

bag semantics

student
Student1
Student2
Student1

Selection

- The σ (sigma) operator corresponds to the WHERE-clause in SQL
- Syntax: $\sigma_{\text{<condition on rows>}}(R)$
- In SQL:
  `SELECT * FROM <SQL for R> WHERE <condition on rows>`
- Conditions should be simple row-wise checks, do not put RA-expressions in your conditions (unlike in SQL where subqueries are allowed)
  - Boolean syntax from SQL (AND, OR, NOT ...) or logical symbols (∧,∨,¬...)
  - Comparisons like <, >, = on constants and attributes
- Called the selection operator because it selects which rows to keep

The most unfortunate naming mismatch ever

- Selection (σ) does not correspond to the SELECT clause in SQL!
- σ corresponds more closely to the WHERE clause
- Projection (π) corresponds to SELECT

$\pi_{student}(Grades)$

$\Updownarrow$

SELECT student FROM Grades

$\sigma_{student=1}(Grades)$

$\Updownarrow$

SELECT * FROM Grades WHERE student=1

Base relations/tables

- Base relations like Students in $\pi_{id,name}$(Students) are part of the algebra
  - In one way they are like constants: The schema of the relations are known
  - In one way they are like variables: The tuples in the relations are unknown
  - Intuitively they are like created tables in SQL, not considering INSERTS
- A typical problem: "Using the schema Student($\underline{\text{idnr}}$,year,name), find the names of all students in the third year"
  - Solution: $\pi_{name}(\sigma_{year=3}(\text{Student}))$
  - The schema is important for the solution to work, but the data is not
- Base relations in expressions are simply table names in SQL

Cartesian product

- The relational algebra syntax for Cartesian product is R1 × R2
- In SQL: `SELECT * FROM <SQL for R1>,<SQL for R2>`
- We can now join relations:

$\sigma_{\text{<join condition>}}(T1 \times T2)$

- Equivalent SQL:

```
SELECT * FROM T1, T2 WHERE <join condition>;
```

Compositional expressions, monolithic queries

- Consider this SQL query and an equivalent relational algebra expression:

```
SELECT name, credits FROM Students, Grades
  WHERE idnr = student AND Grade >= 3
```

- $\pi_{\text{name,credits}}(\sigma_{\text{idnr=student AND grade >= 3}}(\text{Students} \times \text{Grades}))$
- The SQL code is a single query performing projection, selection and Cartesian product, whereas the expression does each of those in separate steps
  - This is a fundamental difference of RA and SQL
  - In RA each subexpression results in a relation, SQL "does everything at once" and gets a single results
- We could also express the same query as, for instance: $\pi_{\text{name,credits}}(\sigma_{\text{idnr=student}}(\text{Students} \times \sigma_{\text{grade >= 3}}(\text{Grades})))$

Translating ER to SQL using subqueries

- Consider the expression:
  $\pi_{name,credits}(\sigma_{idnr=student}(Students \times \sigma_{grade >= 3}(Grades)))$
- The most literal way to translate this into SQL is:

```
SELECT name, credits FROM -- Projection
  (SELECT * FROM          -- Selection: idnr=student
    (SELECT * FROM        -- Cartesian product
      Students,           -- Base table Students
      (SELECT *           -- Selection: grade >= 3
        FROM Grades       -- Base table Grades
        WHERE grade >= 3) AS r3
    ) AS r2 WHERE idnr=student)  AS r1;
```

- Here we have translated each subexpression (except tables) into a subquery
  - Highlights the difference between compositional RA and monolithic SQL
  - A more compact translation would be better in practice

Other set operations

- Just like in SQL, we have the three set operations:
  - Union: R1 U R2
  - Intersection: R1 ∩ R2
  - Difference/subtraction: R1 - R2
- Example (idnr of all students that have not passed any courses):

$$\pi_{idnr}(\text{Student}) - \pi_{student}(\sigma_{grade>=3}(\text{Grades}))$$

- "Take all idnr from students, and remove all idnr with a passing grade"
- Like in SQL, schemas must be compatible (same number of attributes)

Extending set operations to bags

- In sets, each tuple is either in or not in each relation
- In bags, each tuple occurs a number of times in each relation
- Assuming x occurs n times in R1 and m times in R2
  - x occurs n+m times in R1 U R2
  - x occurs min(n,m) times in R1 ∩ R2
  - x occurs n-m times in R1 - R2 (minimal of 0 times)
- Translates to UNION ALL, INTERSECT ALL and EXCEPT ALL
- This is the semantics we use for union, intersection and difference in this course

Grouping

- The grouping operator γ (gamma) is like a combined SELECT and GROUP BY
- Syntax: $\gamma_{\text{<attributes/aggregates>}}(R)$
- Example: $\gamma_{\text{student, AVG(grade)} \rightarrow \text{average}}(\text{Grades})$

Table: Grades

student	course	grade
S1	TDA357	3
S2	TDA357	3
S1	TDA143	5

student	average
S1	4
S2	3

- In SQL:

```
SELECT student, AVG(grade) AS average
  FROM Grades GROUP BY student;
```

- Automatically groups by and projects all attributes in the subscript
- The arrow indicates naming (required for all aggregates)
- Result has exactly one attribute for each attribute/aggregate!

Example

```
Students(idnr, name)
Grades(student, course, grade)
  student -> Students.idnr
```

- Select the name of all students that have passed at least 2 courses
- One solution (join first, group later):

$$\pi_{name}(\sigma_{passed>=2}(\gamma_{student,\ name,\ COUNT(*)\rightarrow passed}(\sigma_{grade>=3\ AND\ idnr=student}(Students \times Grades))))$$

Describing the expression "bottom up" (right to left here):

1) Take the product of students and grades
2) Select the rows with passing grades and matching id-numbers
3) Group what remains by student and calculate the number of passed
4) Select the rows with at least two passed
5) Project the name attribute

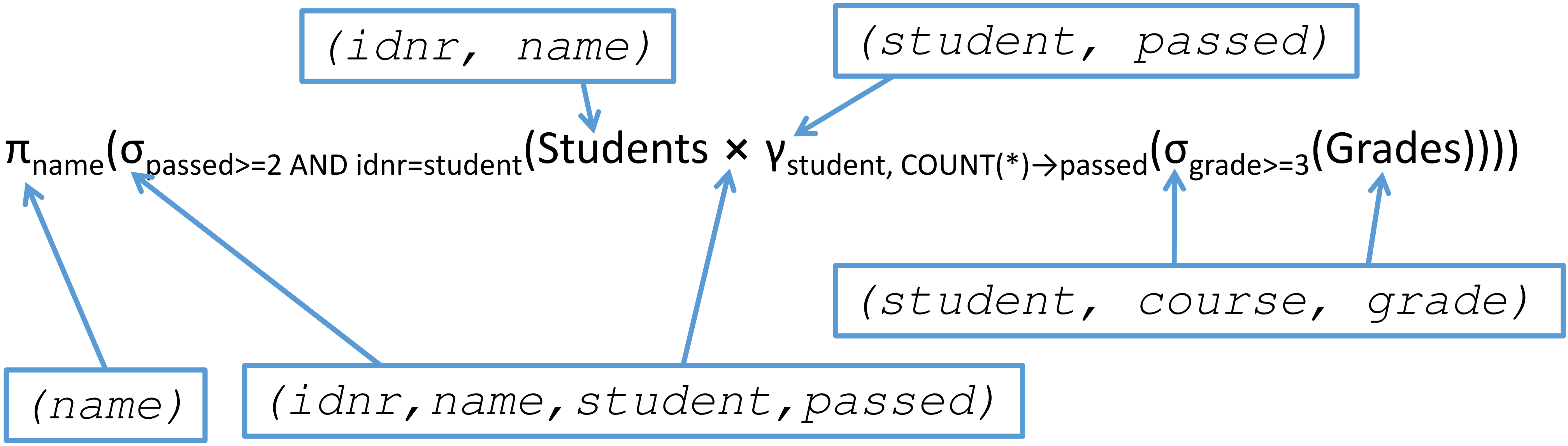
- Another solution (group first, join later)

$$\pi_{name}(\sigma_{passed>=2\ AND\ idnr=student}(Students \times \gamma_{student,\ COUNT(*)\rightarrow passed}(\sigma_{grade>=3}(Grades))))$$

Analyzing expressions

```
Students(idnr, name)
Grades(student, course, grade)
   student -> Students.idnr
```

- To make sure our expression is correct, we can compute the schema of the result for any subexpression (=result of any operator)



$$\pi_{\text{name}}(\sigma_{\text{passed>=2 AND idnr=student}}(\text{Students} \times \gamma_{\text{student, COUNT(*)}\rightarrow\text{passed}}(\sigma_{\text{grade>=3}}(\text{Grades}))))$$

- Sanity check: All our conditions, projections etc. only mention attributes that actually exist in their operands

Sanity check

```
Students(idnr, name)
Grades(student, course, grade)
   student -> Students.idnr
```

- What is wrong with this expression?

$$\pi_{name}(\sigma_{passed>=2 \text{ AND } idnr=student \text{ AND } grade>=3}(Students \times \gamma_{student,\ COUNT(*)\rightarrow passed}(Grades))))$$

Can not use grade here!

```
(idnr,name,student,passed)
```

- Not doing this simple sanity check is probably the most common way to unnecessarily loose points on the exam

What about HAVING?

- In SQL the HAVING-clause is like an extra WHERE-clause that happens after/during grouping, having such an operator in RA does not make sense
- This is only a feature of SQL to avoid using subqueries all the time
- This query:

```
SELECT student FROM Grades
 GROUP BY student
 HAVING AVG(grade)>4;
```

  Note: We need a name here

  Corresponds to this expression:

  $\pi_{\text{student}}(\sigma_{\text{average}>4}(\gamma_{\text{student, AVG(grade)}\rightarrow\text{average}}(\text{Grades})))$

- No need for a separate operator working on aggregates
  - But it is important to do the selection <u>outside</u> the grouping when translating a HAVING-clause to relational algebra
  - Do the sanity check!

Qualified names

- Base relations have names that can be used in conditions etc.
- The results of expressions do not have names though
- Technically, expressions like $\pi_{R1.x}(R1 \times R2)$ are invalid, because the result of $(R1 \times R2)$ does not have a name
    - Like SELECT R1.x FROM (SELECT * FROM R1 × R2), which is invalid
    - Essentially means qualified names are never useful in projections
- This is often ignored in examples of relational algebra and each attribute is understood to retain its qualified name
    - I will allow this in this course

Qualified names

Students($\underline{\texttt{idnr}}$, name)
Grades($\underline{\texttt{idnr}}$, $\underline{\texttt{course}}$, grade)
student -> Students.idnr

- If there are name clashes, it makes sense to sanity check with qualified names

$$\pi_{\text{name}}(\sigma_{\text{Student.idnr=Grades.idnr AND average>4}}(\text{Students} \times \gamma_{\text{idnr, AVG(grade)}\rightarrow\text{average}}(\text{Grades}))))$$

(Grades.idnr, average)

(Students.idnr, Students.name, Grades.idnr, average)

- Note that the attribute average does not have any qualified name

Renaming

- The ρ (rho) operator renames the result of an expression
- Syntax: $\rho_{\text{<new schema>}}(R)$
- Example $\rho_{S(\text{idnr},\text{studentname})}(\text{Students})$

Renames both the relation (for qualified names) and attributes

Students

idnr	**name**
1	Jonas
2	Emilia
3	Emil

S

idnr	**studentname**
1	Jonas
2	Emilia
3	Emil

- Use $\rho_S(\text{Students})$ to only rename the relation and keep attribute names

Table: Numbers

owner	num
Bart	11111
Lisa	22222
Bart	33333

Renaming example

- Consider this query (self join)

```
SELECT N1.num, N2.num, N1.owner
 FROM Numbers AS N1, Numbers AS N2
 WHERE N1.owner = N2.owner;
```

- Here the ρ operator is essential

$$\pi_{N1.num,\ N2.num,\ N1.owner}(\sigma(_{N1.owner\ =\ N2.owner}(\rho_{N1}(Numbers) \times \rho_{N2}(Numbers))))$$

Sanity check: *(N1.owner, N1.num, N2.owner, N2.num)*

Query optimization

- In relational algebra we can express (and prove) rules like:

$$\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c1\ \text{AND}\ c2}(R)$$

$$\pi_{p1}(\pi_{p2}(R)) = \pi_{p1}(R)$$

$$R1 \cap R2 = R1 - (R1 - R2)$$

$$\sigma_{c}(R1 \times R2) = \sigma_{c}(R1) \times R2, \text{ assuming c uses only attributes of R1}$$

- These rules can be used by DBMS to simplify or optimize queries

Join operator

- Like in SQL, there is a special join operator: $R1 \bowtie_{<condition>} R2$
- This is purely a convenience operator, we can define it using:

$R1 \bowtie_c R2 = \sigma_c(R1 \times R2)$

Expression layout

- When writing relational algebra expressions on paper, it is convenient to start each operator on its own row
  - It's often a good idea to start in the middle of the paper with a join, then add operators above it
  - You can easily extend conditions with an extra AND etc.

```
π_name
  (σ_passed>=2
     (Students
       ⋈_idnr=student
      γ_student, COUNT(*)→passed
         (σ_grade>=3
           (Grades))))
```

π_{name}
$(\sigma_{passed>=2}$
$(Students$
$\bowtie_{idnr=student}$
$\gamma_{student,\ COUNT(*)\rightarrow passed}$
$(\sigma_{grade>=3}$
$(Grades))))$

Splitting up expressions

- You can break out and name parts of your expressions for readability

$R1 = \gamma_{\text{student, COUNT(*)}\rightarrow\text{passed}}(\sigma_{\text{grade>=3}}(\text{Grades}))$

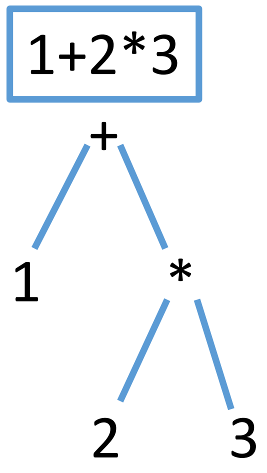
$R2 = \text{Students} \bowtie_{\text{idnr=student}} R1$

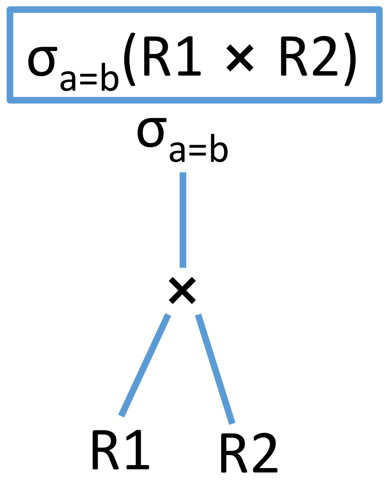
$\text{Result} = \pi_{\text{name}} \ (\sigma_{\text{passed>=2}}(R2))$

- Can simplify expression writing a lot, especially on paper
- Helps the thought process when incrementally solving problems
- Names are not part of the algebra, just a convenience for writing expressions
  - Like saying "let x = min(y,z) in x*(x+1)", x can be substituted for its definition
  - The names can <u>not</u> be used as qualified name (unless you use ρ)
- Remember to still do the sanity check! (What attributes do R1 and R2 have?)

Expression trees

- The best way to understand an expression in any algebra, is as a syntax tree





- Each node in the tree can be computed into a value (or a schema), bottom up

All basic operators (a few more on next slide)

- Selection, "Sigma": $\sigma_{\text{<selection condition>}}(R)$
- Projection, "Pi": $\pi_{\text{<attribute list>}}(R)$
- Cartesian product: $R1 \times R2$
- Other set operations: $R1 \cup R2$, $R1 \cap R2$, $R1 - R2$
- Grouping, "Gamma": $\gamma_{\text{<attributes/aggregates>}}(R)$
- Join: $R1 \bowtie_{\text{<condition>}} R2$
- Renaming, "Rho": $\rho_{\text{<Relation name>(<optional attribute names>)}}(R)$

Additional operators

- Apart from the operators we have seen so far there are several extensions to match various features of SQL
- NATURAL JOIN: R1 $\bowtie$ R2 (Just omit the Join-condition)
- JOIN USING: R1 $\bowtie_{idnr}$ R2 (replace Join-condition with attribute)
- Outer joins:
  - Full outer join: R1 $\bowtie^{o}_{\text{<join condition>}}$ R2
  - Left/right join: R1 $\bowtie^{oL}_{\text{<join condition>}}$ R2 and R1 $\bowtie^{oR}_{\text{<join condition>}}$ R2
- DISTINCT: δ (delta), for converting from a bag to a set
  e.g. R1 U R2 is UNION ALL in SQL, δ (R1 U R2) is UNION
- τ (tau), for ORDER BY on an expression. Examples:
  τ_{grade}(Grades) for SELECT * FROM Grades ORDER BY grade ASC
  τ_{-grade}(Grades) for SELECT * FROM Grades ORDER BY grade DESC

Is it OK if I just learn SQL and translate that to RA?

- Yes!
- But the translation is not always trivial
- Relational algebra is not just SQL in Greek!

Translating a single query

- A query with almost everything:

```
SELECT a1, MAX(a2) AS mx
 FROM T1, T2
 WHERE a3=5
 GROUP BY a1,a3
 HAVING COUNT(*) > 10
 ORDER BY a1 ASC;
```

- A relational algebra expression for it:

$\tau_{a1}(\pi_{a1,mx}(\sigma_{temp>10}(\gamma_{a1,a3,MAX(a2)\rightarrow mx,COUNT(*)\rightarrow temp}(\sigma_{a3=5}(T1 \times T2)))))$

Some things, like HAVING requires new names to be introduced

- The sanity check is even more important when "blindly" translating

Translating correlated queries

Correlation: subquery refers to outer query

- Consider a query like

```
SELECT name FROM Students AS S
   WHERE 4<(SELECT AVG(grade) FROM Grades WHERE student=S.idnr);
```

- This is very easy to mistranslate (if you don't sanity check!)
- The correlation needs to be replaced with a join:

$\pi_{name}(\sigma_{4<average}(\gamma_{student,\ AVG(grade) \rightarrow average}(Grades) \bowtie_{idnr=student} Students))$

Never put RA-expressions in your RA-conditions!
Conditions should be simple boolean expressions

What about things like NOT IN and NOT EXISTS?

- Set subtraction can often (always?) be used to replace NOT IN
- Example: Select students that have no grades

```
SELECT idnr,name FROM Students
  WHERE idnr NOT IN (SELECT student FROM Grades);
```

- In relational algebra (one of many possible solutions):

$R1 = \rho_{NoGrades(s)}(\pi_{idnr}(Students) - \pi_{student}(Grades))$

$Result = \pi_{idnr,name}(Students \bowtie_{s=idnr} R1)$

- Use set subtraction to get the ID of all students without grades, then join back with Students to recover names
  (uses renaming to avoid having two Students.idnr for the join)