Database Theory

Relational Query Languages

Matthias Lanzinger, 2025

TU WIEN Informatics

The Relational Model

Setup

The relational database model is essentially first-order structures + names for attributes.

This extra information on attributes can be useful when describing queries.

Sometimes it is unnecessary and we just work on plain first-order structures.

Reminder First-order structures:

Relations $R_1, R_2, \ldots, R_n$ that each have an arity $\#R_i$.

Assigned meaning through an interpretation I over a domain Dom:

$$I(R_i) \subseteq Dom_n^{\#R_i}$$

Schemas

Let **Rel** and **Att** be (countably infinite) vocabularies of relation names and attribute names.

A database schema $\mathcal{S}$ is a partial function $\mathcal{S} : \mathsf{Rel} \to 2^{\mathsf{Att}}$ such that $Dom(\mathcal{S})$ is finite and every image under $\mathcal{S}$ is finite.

The arity of $R \in Dom(\mathcal{S})$ is defined as $|\mathcal{S}(R)|$.

Example:

In a database we have a table $Student$ with columns for id, name and birthdate.

Formally, there is a relation name $Student \in \mathsf{Rel}$ and we use the schema $\mathcal{S}(Student) = (id, name, brithdate)$ where these are names in **Att**.

Relation Instances

Each attribute $A \in \textsf{Att}$ has a domain $Dom(A)$.

A tuple for relation name R (under schema $\mathcal{S}$) is an element of
$Dom(A_1) \times Dom(A_2) \times \cdots \times Dom(A_n)$
where $\mathcal{S}(R) = (A_1, A_2, \ldots, A_n)$.

A relation (instance) for R (under $\mathcal{S}$) is a *finite* set of tuples for R

A database is a finite set of relations under some schema $\mathcal{S}$.

Continuing our example:

$Dom(id) = \mathbb{N}$,
$Dom(name) =$ all strings
(Σ^* for some alphabet Σ)
$Dom(birthdate)$ = e.g., all strings of certain format

Example tuple:

$(1,$ Bob, 12-03-4567$) \in Dom(id) \times Dom(name) \times Dom(birthdate)$

Some Helpful Notation

1. Relation Names vs. Relation Instances

   When we have a database D we use R^D to denote the relation instance for relation name R

2. Attributes of Tuples

   For a relation R^D with schema $\mathcal{S}(R) = (A_1, \ldots, A_k)$, we use $A_i(t)$ to denote the i-th position of tuple $t \in R^D$.

Id	Name	DoB
13	Student A	14.06.19
22	Student B	23.06.19
...	...	..

$ID(t_1) = 13, \quad Name(t_1) =$ Student A

Relational Query Languages

Formal Query Languages

As in any discussion of formal languages, query languages always have two core parts.

Syntax
How do terms of the language look like. Can be analogous to logic or more operational.

Semantics
How are expressions of the languages evaluated. The formal definition of what answers we want from the data.

```
SELECT MIN(s_acctbal), MAX(s_acctbal)
FROM part, partsupp, supplier,
     nation, region
WHERE p_partkey   = ps_partkey
  AND s_suppkey   = ps_suppkey
  AND n_nationkey = s_nationkey
  AND r_regionkey = n_regionkey
  AND p_price >
      (SELECT avg (p_price) FROM part)
  AND r_name IN ('Europe', 'Asia')
```

$$\exists y\ x \xrightarrow{a^*b} y \wedge x \xrightarrow{(a+b)^*c} y$$

$$\pi_{\{\text{pid},\text{pname}\}}(\text{Person}) - \pi_{\{\text{pid},\text{pname}\}}(\text{Person} \bowtie \text{City})$$

```
SELECT ?capital
       ?country
WHERE
  {
    ?x  ex:cityname       ?capital
        ex:isCapitalOf    ?y
    ?y  ex:countryname    ?country
        ex:isInContinent  ex:Africa
  }
```

$$(\forall x)(\exists yz)(y \neq z \wedge E(x,y) \wedge E(x,z) \wedge (\forall w)(E(x,w) \to (w = y \vee w = z)))$$

$$\begin{aligned} \exists z \mathsf{Drive}(z,x,y) &\leftarrow (\Diamond_{[0,\infty]} \mathsf{Depart}(x,y))\, \mathcal{U}_{[0,\infty)} \mathsf{Arrive}(x,y), \\ \mathsf{Working}(z) &\leftarrow \mathsf{Drive}(z,x,y), \\ \mathsf{Dangerous}(x) &\leftarrow \boxminus_{[0,8]} \mathsf{Working}(z) \wedge \mathsf{Drive}(z,x,y). \end{aligned}$$

Example Schema & Database

Course

Name	Sem	Lecturer
Logic	W24	L1
Complexity	S24	L2
Logic	W23	L1

Student

Id	Name	DoB	Active
13	Student A	14.06.1903	TRUE
22	Student B	23.06.1912	FALSE
...	...	..	...

Enrolled

Course	Graded	Student
Logic	FALSE	13
Logic	TRUE	22
Complexity	TRUE	13

Relational Algebra

Overview

Syntax:

Relational Algebra (RA) expressions e are formed inductively:

- every relation name R is an RA expression

- If e_1, e_2 is an RA expressions, so are:

$$\sigma_\theta(e_1) \qquad \pi_\alpha(e_1) \qquad \rho_{A\to B}(e_1)$$
$$e_1 \times e_2 \qquad e_1 \cup e_2 \qquad e_1 - e_2$$

Semantics:

An expression e applied to a database D evaluates to a new relation, we write $e(D)$.

If e is relation name R, then $e(D) = R^D$.

Semantics of other operators are defined on the following slides.

σ

Selection

Syntax:

$e = \sigma_\theta(e_1)$ where

- e_1 is a RA expression of sort U
- θ is a propositional formula over attributes in U, $=$, and constants.

Semantics:

$e(D) = \{t \in e_1(D) \mid \theta(t) \text{ is true }\}$

with schema $\mathcal{S}(e) = \mathcal{S}(e_1)$

σ

Selection

$\sigma_{Sem='W24'}(Course) \longrightarrow$

Name	Sem	Lecturer
Logic	W24	L1
Complexity	S24	L2
Logic	W23	L1

$$\pi$$

Projection

Syntax:

$e = \pi_\alpha(e_1)$ where

- e_1 is a RA expression of sort U
- α is sequence of attributes in U

Semantics:

$$e(D) = \{(\alpha_1(t), \alpha_2(t), \ldots, \alpha_{|\alpha|}) \mid t \in e_1(D)\}$$

with schema $\mathcal{S}(e) = \alpha$

$$\pi$$

Projection

$$\pi_{Active,ID}(Student) \longrightarrow$$

Active	Id
TRUE	13
FALSE	22
...	...

ρ
Renaming

Syntax:

$e = \rho_{A \to B}(e_1)$ where

- e_1 is a RA expression of sort U
- $A \in U$, and $B \in \mathsf{Att} \backslash U$

Semantics:

$e(D) = e_1(D)$

with schema $\mathcal{S}(e) = \mathcal{S}(e_1)[A/B]$

Replace A with B

$$\rho$$

Renaming

$$\rho_{Course \to CID, Student \to SID}(Enrolled) \longrightarrow$$

CID	**Graded**	**SID**
Logic	FALSE	13
Logic	TRUE	22
Complexity	TRUE	13

$\times$

Product

Syntax:

$e = e_1 \times e_2$ where

- e_1, e_2 are RA expressions with schema $(A_1, \ldots, A_n)$ and $(B_1, \ldots, B_m)$, respectively.

Semantics:

$$e_1(D) = \{(a_1, \ldots, a_n, b_1, \ldots, b_m) \mid (a_1, \ldots, a_n) \in R, (b_1, \ldots, b_m) \in S\}$$

with schema $\mathcal{S}(e) = (A_1, \ldots, A_n, B_1, \ldots, B_n)$

$\times$ Product

$$Enrolled \times \pi_{ID,Name}(Student) \longrightarrow$$

Course	Graded	Student	ID	Name
Logic	FALSE	13	13	Student A
Logic	FALSE	13	22	Student B
Logic	TRUE	22	13	Student A
Logic	TRUE	22	22	Student B
Complexity	TRUE	13	13	Student A
Complexity	TRUE	13	22	Student B
...	...	...	...	...

$-$

Difference

Syntax:

$e = e_1 - e_2$ where

- e_1, e_2 are RA expressions of sort U

Semantics:

$e(D) = e_1(D) \backslash e_2(D)$

$\cup$

Union

Syntax:

$e = e_1 \cup e_2$ where

- e_1, e_2 are RA expressions of sort U

Semantics:

$e(D) = e_1(D) \cup e_2(D)$

$\bowtie$
(Natural) Join

Very common in database queries.
Can be expressed via other operators:

$$e_1 \bowtie e_2 := \sigma_{A=A'}(e_1 \times \rho_{A\to A'}(e_2))$$

Where A is the only shared attribute between e_1, e_2.
(generalisation to more shared attributes is straightforward)

Example Query

List lectures together with the students that attend them in WS24:

$$\overbrace{\pi_{Course,Student}\left(\overbrace{\overbrace{\pi_{Course,Student}(Enrolled)}^{\text{Remove Grade attribute}} \bowtie \overbrace{\rho_{Name\to Course}\,\overbrace{\sigma_{Sem=WS24}(Course)}^{\text{Limit to WS24}}}^{\text{Rename for Join}} \bowtie \overbrace{\rho_{Id\to Student}(Student)}^{\text{Rename for Join}}}^{\text{Connect student data to course data via enrollment}}\right)}^{\text{Keep only the two attributes that we want}}$$

The natural database theory question:
Do we need all of these operators?

Yes 👍 — but how do we show this?

(Except for ⋈ of course)

Example: Renaming

Relation R

A
1
2
3

Relation S

B
3
4
5

Consider the expression

$R \cup \rho_{B \to A}(S)$

A
1
2
3
4
5

We can show that ρ is necessary by showing that in RA without renaming there is no expression that gives the same output on these inputs!

Relational Domain Calculus

= First-Order Queries

Queries from Logic

Let φ be a formula with free variables $x_1, \ldots, x_k$, then

$$\{(x_1, x_2, \ldots, x_k) \in \mathsf{Dom}^k \mid D \vDash \varphi(x_1, x_2, \ldots, x_k)\}$$

is a k-ary query. It returns a set of k-tuples that represent "solutions" for φ on database D.

Logics can be seen as query languages!

Relational Domain Calculus

The query language induced by *first-order logic* is called relational (domain) calculus.

Quick reminder — satisfaction of first-order sentences:

$$
\begin{array}{lcl}
I \models R(t_1, \ldots, t_n) & \iff & R(t_1, \ldots, t_n) \in I \\
I \models t_1 = t_2 & \iff & t_1 \text{ and } t_2 \text{ are the same object.} \\
I \models \neg\phi & \iff & I \not\models \phi \\
I \models \phi_1 \wedge \phi_2 & \iff & I \models \phi_1 \text{ and } \phi_2 \\
I \models \exists x \,.\, \phi & \iff & I \models \phi[x/c] \text{ for some } c \in Dom
\end{array}
$$

with $\forall x \,.\, \phi := \neg \exists x \,.\, \neg\phi$ and $\phi_1 \vee \phi_2 := \neg(\neg\phi_1 \wedge \neg\phi_2)$

Example Query

Recall our example query:
list lectures together with the students that attend them in WS24.

$$\{(c, sname) \mid \exists sem, grade, sid, l, dob, active\ .\ Enrolled(c, grade, sid) \land Course(c, sem, z) \land sem = \text{WS24} \land Student(sid, sname, dob, active)\}$$

Course

Name	Sem	Lecturer
Logic	W24	L1
Complexity	S24	L2
Logic	W23	L1

Student

Id	Name	DoB	Active
13	Student A	14.06.1903	TRUE
22	Student B	23.06.1912	FALSE
...	...	..	...

Enrolled

Course	Graded	Student
Logic	FALSE	13
Logic	TRUE	22
Complexity	TRUE	13

$$\{(c, sname) \mid \exists sem, grade, sid, l, dob, active \,. \\ Enrolled(c, grade, sid) \wedge Course(c, sem, z) \wedge \\ sem = \mathsf{WS24} \wedge Student(sid, sname, dob, active)\}$$

(Logic, Student A) is an answer to the query:

$$sem \mapsto W24, \quad sid \mapsto 13, \quad dob \mapsto 14.06.1903$$

$$grade \mapsto \mathsf{TRUE}, \quad l \mapsto L1, \quad active \mapsto \mathsf{TRUE}$$

SQL

SQL Overview

The standard language for relational database systems.

Originally developed in the 1970s inspired by the relational model and especially relational algebra.

SQL Query Syntax...

We are not going to formally define SQL.

- Syntax changes between implementations.
- Contains constructs that specify details of the actual execution of the query, e.g.,
  WITH … AS MATERIALIZED
  which makes it challenging to specify formal semantics.

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ { * | expression [ [ AS ] output_name ] } [, ...] ]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY [ ALL | DISTINCT ] grouping_element [, ...] ]
    [ HAVING condition ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } { ONLY | WITH TIES } ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF from_reference [, ...] ] [ NOWAIT | SKIP LOCKED ] [.

where from_item can be one of:

    [ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
                [ TABLESAMPLE sampling_method ( argument [, ...] ) [ REPEATABLE ( seed ) ] ]
    [ LATERAL ] ( select ) [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] )
                [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] ) [ AS ] alias ( column_definition [, ...] )
    [ LATERAL ] function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
    [ LATERAL ] ROWS FROM( function_name ( [ argument [, ...] ] ) [ AS ( column_definition [, ...] ) ] [, ...] )
                [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    from_item join_type from_item { ON join_condition | USING ( join_column [, ...] ) [ AS join_using_alias ] }
    from_item NATURAL join_type from_item
    from_item CROSS JOIN from_item

and grouping_element can be one of:

    ( )
    expression
    ( expression [, ...] )
    ROLLUP ( { expression | ( expression [, ...] ) } [, ...] )
    CUBE ( { expression | ( expression [, ...] ) } [, ...] )
    GROUPING SETS ( grouping_element [, ...] )

and with_query is:

    with_query_name [ ( column_name [, ...] ) ] AS [ [ NOT ] MATERIALIZED ] ( select | values | insert | update |
        [ SEARCH { BREADTH | DEPTH } FIRST BY column_name [, ...] SET search_seq_col_name ]
        [ CYCLE column_name [, ...] SET cycle_mark_col_name [ TO cycle_mark_value DEFAULT cycle_mark_default ] USI

TABLE [ ONLY ] table_name [ * ]
```

Image source: https://www.postgresql.org/docs/current/sql-select.html

Core SQL Queries

$$Q_1, Q_2 := \textbf{SELECT} \text{ <select\_list>} \; \textbf{FROM} \text{ <from\_list>} \; \textbf{WHERE} \text{ <condition>}$$

$$\mid \; Q_1 \; \textbf{UNION} \; Q_2$$

$$\mid \; Q_1 \; \textbf{EXCEPT} \; Q_2$$

Core SQL Queries

$Q_1, Q_2 :=$ **SELECT** <select_list>
FROM <from_list>
WHERE <condition>

Constants or attributes of relation names from the <from_list>

| Q_1 **UNION** Q_2

| Q_1 **EXCEPT** Q_2

Core SQL Queries

$Q_1, Q_2 :=$ **SELECT** <select_list>
FROM <from_list>
WHERE <condition>

| Q_1 **UNION** Q_2

| Q_1 **EXCEPT** Q_2

List of relation names and (core SQL) subqueries. Can be aliased (renamed) to use relation repeatedly.

Core SQL Queries

$$Q_1, Q_2 :=$$ **SELECT** <select_list>
FROM <from_list>
WHERE <condition>

| Q_1 **UNION** Q_2

| Q_1 **EXCEPT** Q_2

An *expression* consisting of:

- Equalities between attributes, e.g., R.a = S.a.
- Equalities between attributes and constants, e.g., R.a = 7
- Combinations of expressions using **AND**, **OR**, and **NOT**.

Core SQL Queries

Theorem

For every Core SQL query q, there exists an RA query q' such that $q(D) = q'(D)$ for every database D, and vice versa.

For details, see Arenas, et al. "Database Theory.", Section 5.

Informally, this means that we can focus our theoretical analysis only on one of these languages!

SQL — Example

List lectures together with the students that attend them in WS24:

```sql
SELECT course.name, student.name
FROM course, student, enrolled
WHERE course.name = enrolled.course AND
      course.sem = enrolled.sem AND
      student.id = enrolled.student AND
      enrolled.sem = 'WS24';
```

Warning: Bag vs. Set Semantics

Set Semantics: Answers to queries are *sets* of tuples. That is, there is no repetition in answers and operations ignore repeating tuples.

Bag Semantics: Answers to queries are *bags (or multisets)* of tuples. Repetition matters!

$\pi_A\Big($

A	B
1	2
1	3

$\Big) =$

A
1

Set Semantics

/

A
1
1

Bag Semantics

Warning: Bag vs. Set Semantics

Set Semantics: Answers to queries are *sets* of tuples. That is, there is no repetition in answers and operations ignore repeating tuples.

Bag Semantics: Answers to queries are *bags (or multisets)* of tuples. Repetition matters!

Our definition of relational algebra and relational calculus uses set semantics.
In the statement on the previous slide we assume set semantics for core SQL queries.
However, SQL in practical systems usually uses bag semantics.

Not a problem, it is also straightforward to define relational algebra with bag semantics.
But it is important to always keep in mind which type of semantics we are talking about.

SQL is More than SELECT

Data Description

CREATE: To create new tables, databases, views, or indexes.

ALTER: To modify existing database objects (e.g., add columns to a table).

Data Manipulation

GRANT: To provide specific privileges (e.g., SELECT, INSERT) to users or roles.

REVOKE: To remove previously granted privileges.

Data Control

INSERT: Adds new rows (records) to a table.

UPDATE: Modifies existing rows in a table based on certain conditions.

DELETE: Removes rows from a table based on specified conditions.

Which is Best?

How can we compare query languages?

Two Proposals

For two query languages $\mathscr{L}$ and $\mathscr{M}$:

Are there more efficient algorithms to answer queries in language $\mathscr{L}$ than for queries in $\mathscr{M}$? → ***Complexity*** later lectures

Are there queries in language $\mathscr{L}$ that I cannot equivalently write in language $\mathscr{M}$? → ***Expressivity***

Expressivity of Query Languages

We say that $\mathscr{L}$ can be *expressed by* $\mathscr{M}$ if for every query $\phi \in \mathscr{L}$, there is a query $\psi \in \mathscr{M}$ such that $\phi(D) = \psi(D)$ on every database D

Easy example — Core SQL can be expressed by RA

SELECT $A_1, \ldots, A_\ell$ FROM $T_1, \ldots, T_k$ WHERE C

⬇️

$$\pi_{A_1,\ldots,A_\ell}\, \sigma_C\, (T_1 \times \cdots \times T_k)$$

Expressivity of Query Languages

We say that $\mathscr{L}$ can be expressed by $\mathscr{M}$ if for every query $\phi \in \mathscr{L}$, there is a query $\psi \in \mathscr{M}$ such that $\phi(D) = \psi(D)$ on every database D

We say that $\mathscr{L}$ and $\mathscr{M}$ have the *same expressive power* if $\mathscr{L}$ can be expressed by $\mathscr{M}$ and vice versa.
Intuitively, you can write the exact same queries in both languages.

Codd's Theorem

Theorem (Codd 1972, informal)
Relational algebra, relational domain calculus, and Core SQL Queries have the same expressive power.

Technical caveat: relational calculus queries have to be "safe"

Overview of Proof Idea

Explore the details in a theory exercise!

Relational Algebra

(Domain-Indep.) Relational Calculus

Active Domain Relational Calculus

RC where quantifiers range only over values in the database.

Range-restricted Calculus

RC where every variable tied to some relation.

We move through multiple languages to get back to RA!

Qualitative Comparison

As a consequence of the languages being equivalent we can switch between them depending on the task.

Relational Algebra

Operational semantics are well suited for topics where we care about the steps taken to execute a query.

Well-suited for the study of query optimisation and execution.

Relational Calculus

Declarative language with cleanest semantics. Direct connection to extensive body of work on logic.

Well-suited for theoretical study of complexity and expressivity.

SQL

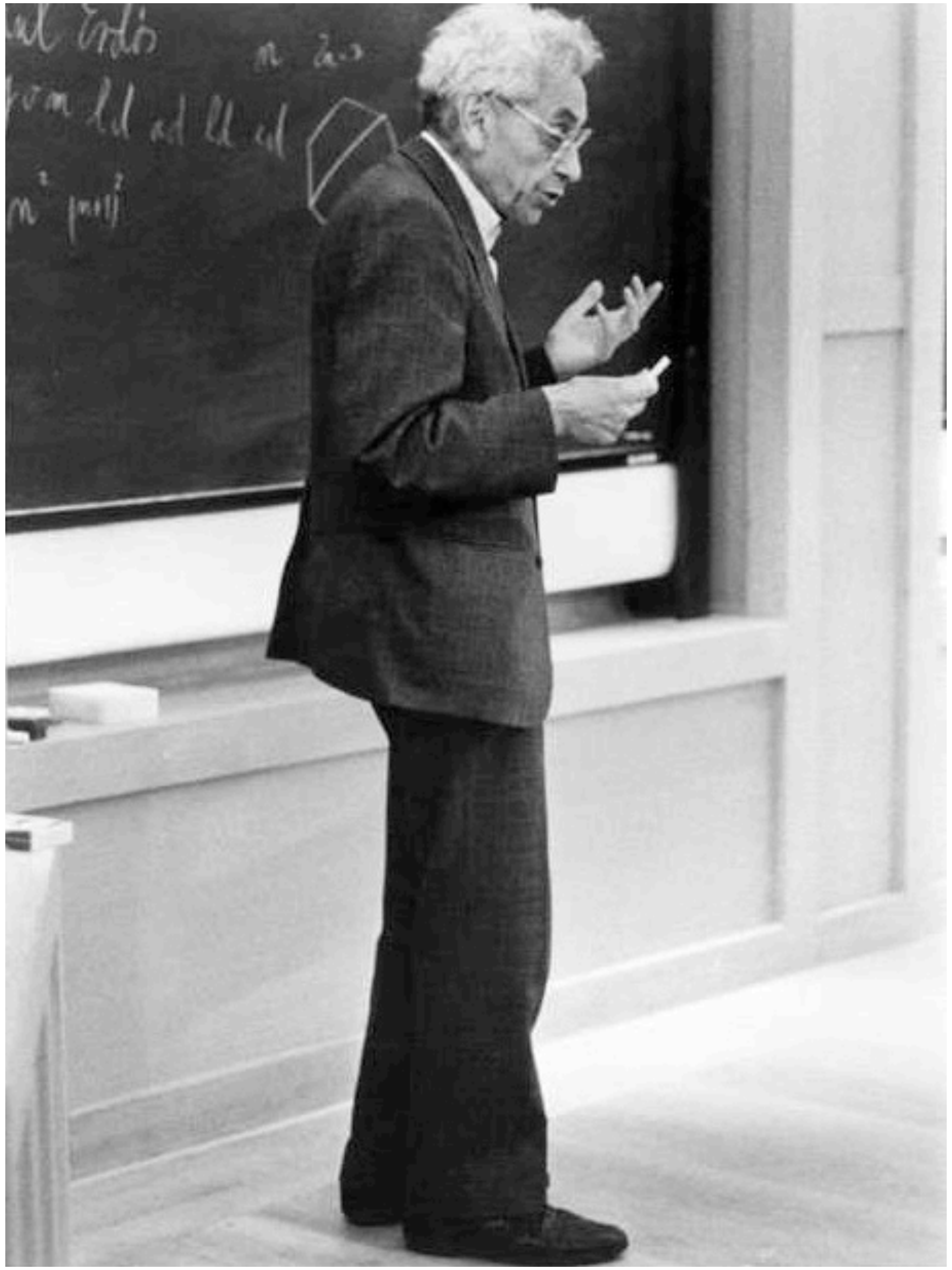
"User-friendly" language aimed at end-users of actual systems. Extremely wide-spread in the real world.

Can these languages do everything?

Limitations?

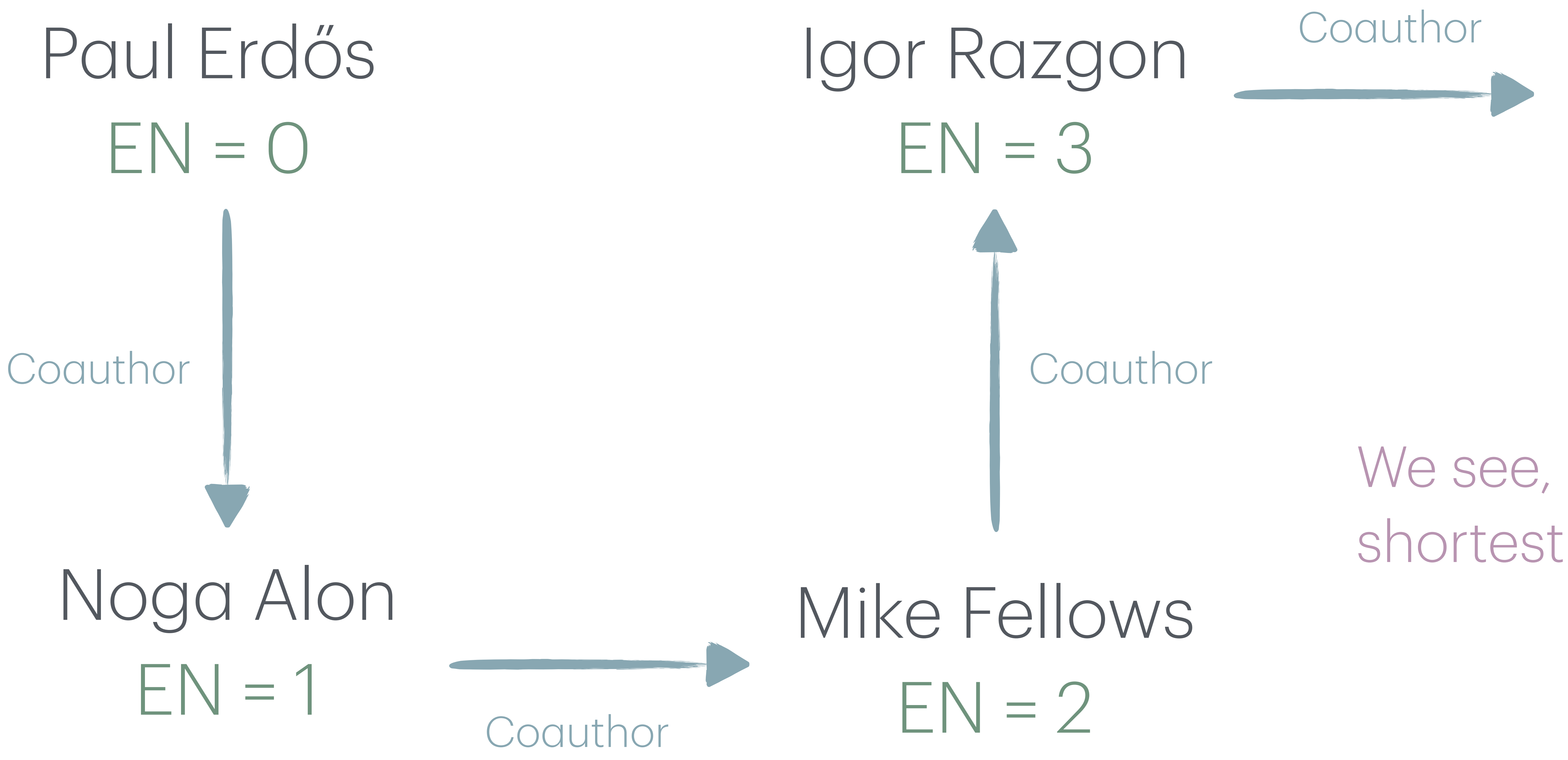
- Paul Erdős was one of the most prolific mathematicians of all time. He wrote over 1500 articles, many of them highly influential.
  He had 509 direct collaborators!

- The Erdős Number is a way of describing the "collaboration distance" Paul Erdős.
  - Erdős has an Erdős number of 0
  - The Erdős Number of author M is the minimum among the Erdős Numbers of all the coauthors of M, plus 1



https://sites.math.rutgers.edu/~sg1108/People/Math/Erdos

Example



We see, it's simply a fun way of describing shortest path in the coauthor graph.

Isaac Asimov

EN = ∞

Querying the Erdős Number

Assume a database with schema:

Author(aid, name), **Paper**(pid, title), **Wrote**(aid, pid)

We can query the authors with EN ≤ 1 easily:

$$P := \pi_{pid}\left(\sigma_{name=\text{'Paul Erdos'}}(Author) \bowtie Write\right)$$ get the ids of Erdős' papers

$$Q := \pi_{aid}(P \bowtie Author)$$ get the authors of those papers

Can we also get the authors with EN = 1?

Querying the Erdős Number

Assume a database with schema:

Author(aid, name), **Paper**(pid, title), **Wrote**(aid, pid)

We can query the authors with EN $\leq$ 1 easily:

$$P := \pi_{pid}\left(\sigma_{name=\text{'Paul Erdos'}}(Author) \bowtie Write\right)$$ get the ids of Erdős' papers

$$Q := \pi_{aid}(P \bowtie Author)$$ get the authors of those papers

Can we also get the authors with EN = 1?

Yes: $Q - \pi_{aid}\,\sigma_{name=\text{'Paul Erdos'}}(Author)$

Querying the Erdős Number

Assume a database with schema:

Author(aid, name), **Paper**(pid, title), **Wrote**(aid, pid)

We can query the authors with EN $\leq$ 2 just as easily:

$P_0 := \pi_{pid}\left(\sigma_{name=\text{'Paul Erdos'}}(Author) \bowtie Write\right)$ get the ids of Erdős' papers

$Q_1 := \pi_{aid}(P_0 \bowtie Author)$ get the authors with EN at most 1

$P_1 := \pi_{pid}(Q_1 \bowtie Author)$ get their papers

$Q_2 := \pi_{aid}(P_1 \bowtie Author)$ and get those papers' coauthors

Querying the Erdős Number

Assume a database with schema:

Author(aid, name), **Paper**(pid, title), **Wrote**(aid, pid)

Let's be more ambitious. Can we write RA queries for the following questions:

- AIDs of authors with EN < ∞, i.e., those with finite EN?
- AIDs of authors with no EN?

Querying the Erdős Number

Assume a database with schema:

Author(aid, name), **Paper**(pid, title), **Wrote**(aid, pid)

Let's be more ambitious. Can we write RA queries for the following questions:

- AIDs of authors with EN $< \infty$, i.e., those with finite EN?
- AIDs of authors with no EN?

No

Equal expressive power also means that all languages that we've discussed so far share the same limitations!

🙁

Looking Forward

How do we know this?

How can we prove that there cannot be a RA query for these questions?

We use Codd's Theorem in combination with results from logic, e.g., Ehrenfeucht-Fraïsse Games or the Compactness Theorem.

Solutions

Are there query languages that can answer these queries?

Yes! Datalog, a prominent example of such languages will be the topic of a future lecture.

More Bad News

Finite Satisfiability

One natural piece of information for query optimization and automated query analysis is to know whether it is impossible for part of the query to have an answer. In other words, is part of the query always empty over any database?

Formally we say, that query ϕ is *finitely satisfiable* if there exists a (finite) database D such that $\phi(D) \neq \emptyset$.

Remember, databases
are always finite by defintion!

Finite Satisfiability — Example

$$Q(x) := \underbrace{\neg \exists y\, E(y,x)}_{x \text{ has no predecessor}} \wedge \underbrace{\forall z\, \exists w\, (E(z,w) \wedge \forall w' (E(z,w') \to w' = w))}_{\text{is every node has exactly 1 successor}} \wedge \underbrace{\forall w\, \forall z_1\, \forall z_2\, ((F(z_1,w) \wedge F(z_2,w)) \to z_1 = z_2)}_{\text{every node has at most 1 predecessor}}.$$

Ideally a query optimizer would notice this and instantly answer with ∅!

Intuitively, Q asks for those nodes x that are the start of an infinite chain. It is therefore empty for every finite database.

Trakhtenbrot's Theorem

Explore the details in a theory exercise!

Theorem (Trakhtenbrot 1950)
Finite satisfiability of first-order logic is undecidable. That is, given a FO query ϕ, it is undecidable whether $\phi(D) \neq \emptyset$ for some database D.

Two important consequences for us:

- Perfect query optimisation is impossible for FO queries!
- Via Codd's Theorem this applies just as well to RA or even core SQL.

Summary

We have learned how to define relational databases as mathematical objects. This will form the basis for future mathematical arguments about their properties.

Query languages can be defined in many ways. Operational and declarative languages are both important and have their individual strengths. Interestingly, the natural languages of relational algebra and first-order logic turn out to be equivalent (and the same as the core of SQL).

Finally, we saw some first discussion about the limitations of query languages. Both in terms of model decidability and expressivity.