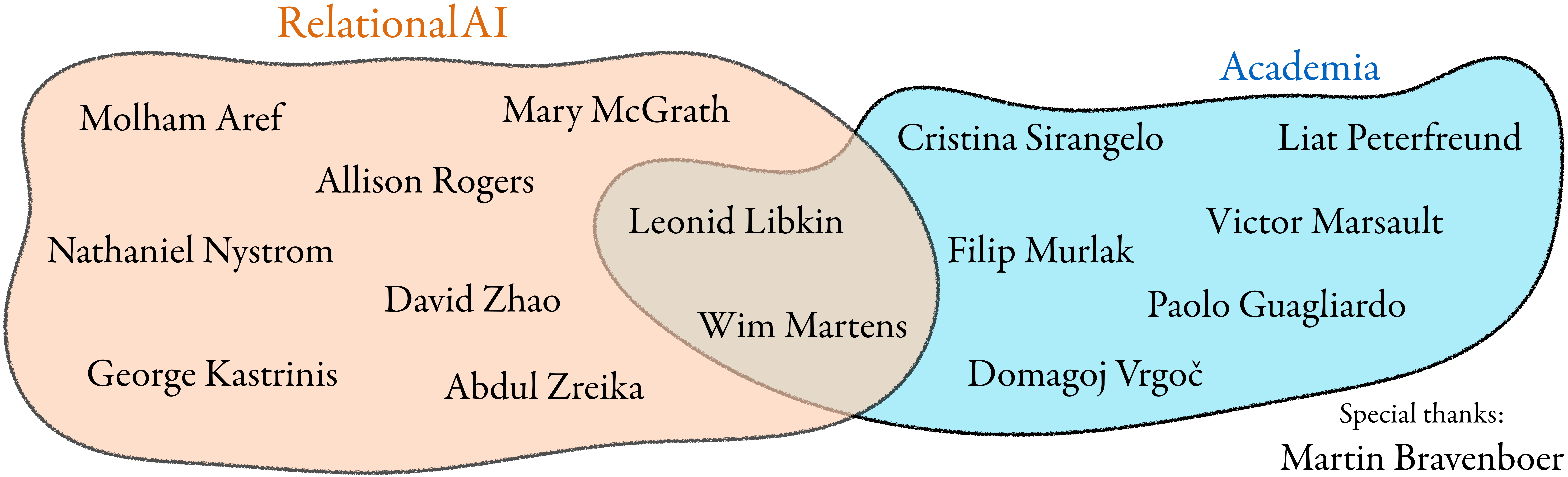
relationalAI

Programming in the Large with Data



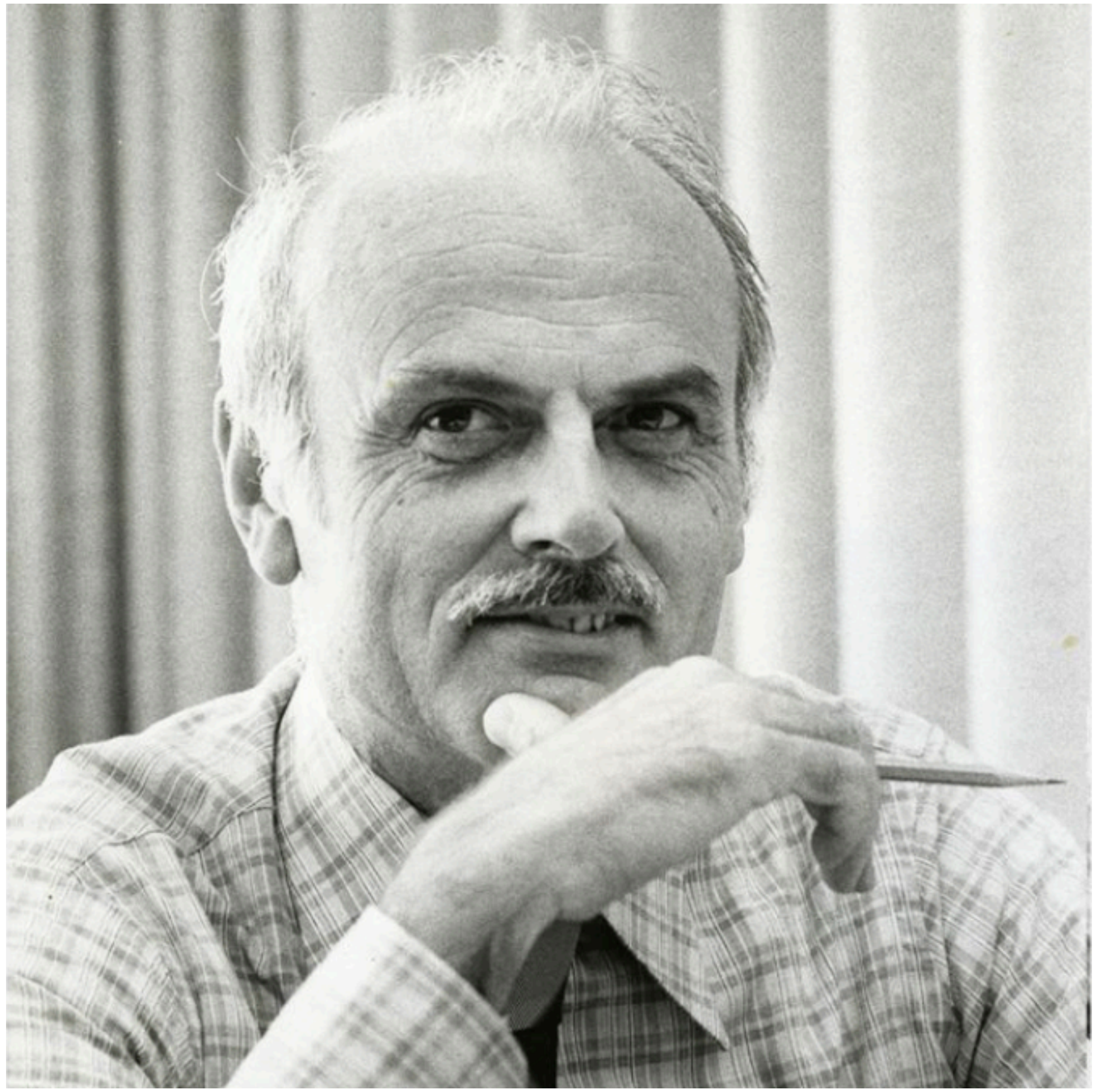
Special thanks:

Martin Bravenboer

Foundations of Databases and AI

TU Vienna

Databases: The Origin Story



(Image: IBM, fair use)

A DATA BASE SUBLANGUAGE FOUNDED ON
THE RELATIONAL CALCULUS

by

E. F. Codd
IBM Research Laboratory
San Jose, California

We use the term data sublanguage (rather than language) because we are not concerned with general processing (or the capability of computing all computable functions). Instead, we wish to focus on only those language components which support storage and retrieval of formatted data from large shared data bases.

Databases: The Origin Story

Why do we have a sublanguage for databases?

It was a brilliant idea in the early 1970s:

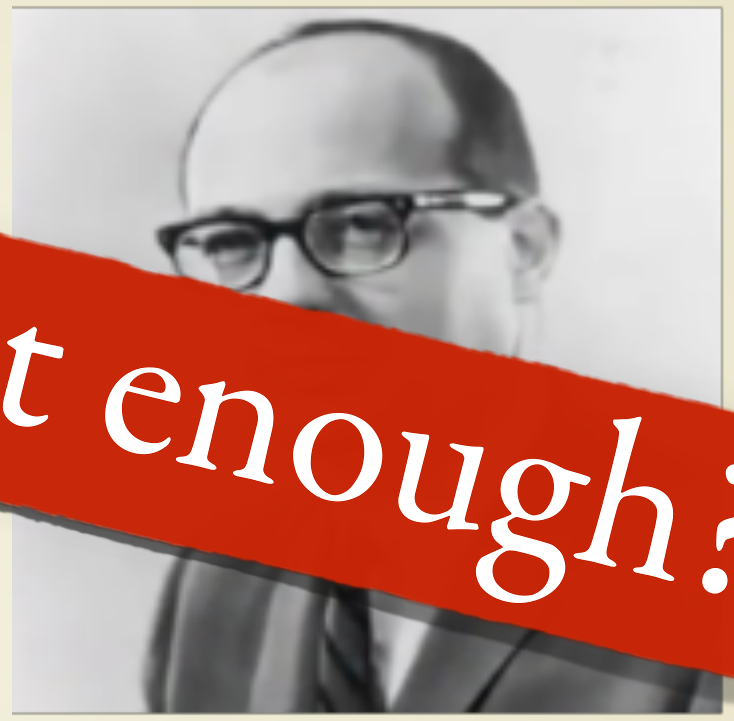
- Querying databases is a nice domain-specific use case
- FO as a natural yardstick
- Declarative programming why tackle the entire problem at once?

Holy grail of programming languages

This was still before the famous SIGFIDET 1974 debate between

Will it ever be efficient enough?

Ted Codd



Charles Bachman

where the question was if a declarative language was even going to work for databases

In the beginning, things were clean

- First-order logic
- Set semantics
- No nulls

SQL today

On top of first-order logic, we have

- bag semantics
- nulls
- arrays
- windowing functions
- XML-related specs
- graph pattern matching
- …

SQL standard today: 4000 pages!

This growth exists because SQL is a sublanguage (not powerful enough to do libraries)

Large-scale applications use a query language and a host language

This causes the **impedance mismatch**:

- different runtimes
- different programming paradigms
- no automatic optimization
- no automatic parallelization
- no automatic …

This is Where We Want to Improve

We want to revisit the sublanguage paradigm

Rel:
Programming in the Large with Data

Rel: Relational Programming

Guy Steele, OOPSLA '98
"Growing a Language"

How do you grow a programming language?
You build a small core of powerful operations

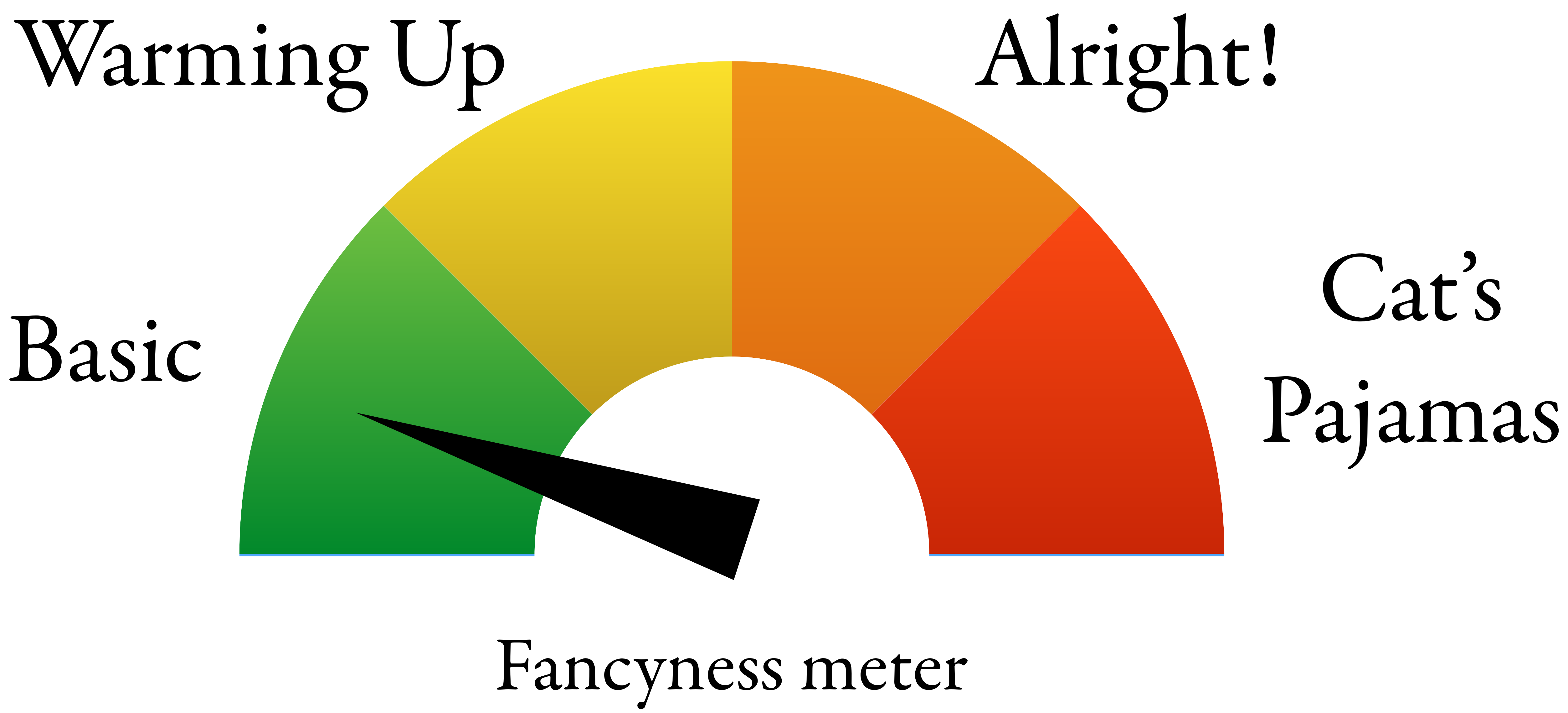
This core should be powerful enough
to build libraries

Why?
You cannot build everything that everyone wants

SQL doesn't have this:
this is why it keeps getting extended

A Crash Course on Rel

"Everything is a relation"


Warming Up
Alright!
Basic
Cat's
Pajamas
Fancyness meter

Rel Basics

relationalAI

Base Relations

- `person(x)`
- `mother(x,y)`
- `father(x,y)`
- `alive(x)`

the mother/father of x is y

```
def parent(x,y) : mother(x,y) or father(x,y)
```

Ingredients

- Datalog rules
- FO in the bodies

Quantifiers:

```
def grandparent(x,y) :
      exists ((z) | parent(x,z) and parent(z,y))

def orphan(x) :
      person(x) and forall ((p) | parent(x,p) implies not alive(p))
```

Infinite Relations

Infinite Relations

- `Int(x), …`
- `>, =, >=, …`
- `add(x,y,z), multiply(x,y,z), modulo(x,y,z), …`

```
def positive_int(x) : Int(x) and x > 0
```

```
def absolute(x,y) : (x >= 0 and y = x) or (x < 0 and y = -x)
```

```
def T(x,y,z) : R(x) and S(y) and add(x,y,z)
```

In SQL:

```
SELECT R.a + S.b from R, S
```

Safety of Rel is non-trivial [Guagliardo et al. ICDT'25]

Rel Basics

relationalAI

Core Operators and Features

Relations: finite and infinite

First-Order Logic

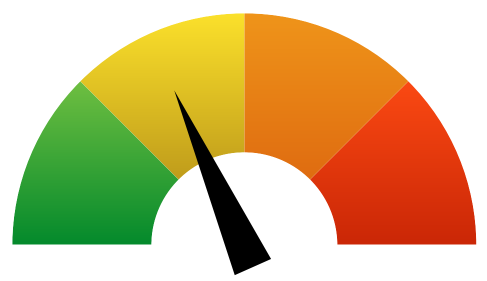
- and, or, not
- forall, exists

Recursion

- Datalog-style

Rel Recursion

```
def ancestor(x,y) : parent(x,y)
def ancestor(x,y) : exists ((z) | parent(x,z) and ancestor(z,y))
```



(Also: non-linear recursion)

Warming up

Fancyness meter

How do we go to Programming in the Large?

Two Extra Features

to enable Relational Programming

Tuple Variables

Higher-Order

Tuple Variables

How do we write generally applicable code?

U	
1	2
3	4
3	5

V	
1	2
6	7

product

1	2	1	2
1	2	6	7
3	4	1	2
3	4	6	7
3	5	1	2
3	5	6	7

```
def Product(a,b,c,d) : U(a,b) and V(c,d)
```

But what if V is ternary?

```
def Product(a,b,c,d,e) : U(a,b) and V(c,d,e)
```

This is both tedious and not generally applicable. Solution:

```
def Product(x…,y…) : U(x…) and V(y…)
```

Tuple Variables

relationalAI

By the way, tuple variables don't need to bind to entire tuples

```
def DotJoin(x…,y…) : exists((v) | U(x…,v) and V(v,y…))
```

Higher-Order Relations

cat's pajamas

relationalAI

We would like something like `Product[U,V]` to return the product of `U` and `V`

and `Product[A,B]` to return the product of `A` and `B`

using relations as parameters

This is done with higher-order variables:

```
def Product({A},{B},x…,y…) : A(x…) and B(y…)
```

Now how do we go to `Product[U,V]` ? ⇝ We need some sugar (will appear soon)

“Everything is a Relation”

relationalAI

```
def Product({A},{B},x…,y…) :
                        A(x…) and B(y…)
```

Product					
{}	{}				
{0}	{}	0			
{0}	{0}	0	0		
…					
{(0,0),(0,1)}	{(1,2)}	0	0	1	2
{(0,0),(0,1)}	{(1,2)}	0	1	1	2
…					

A second-order relation with

- infinitely many rows
- infinitely many columns

Observations

- Users are not exposed to higher-order relations
  - the output is always first-order
- Relations in Rel don’t need a uniform arity

Rel Basics

relationalAI

Core Operators and Features

Relations: finite and infinite

First-Order Logic
- and, or, not
- forall, exists

Recursion
- Datalog-style

Tuple Variables
Higher Order

That's essentially it!

Small core ✓

Powerful ??

Let's Do Some Examples

Partial Application (sugar)

parent	
alice	cindy
john	debby
john	bob

```
parent(“alice”,”cindy”)
```
⇝ true

```
parent[“alice”]
```
⇝ {“cindy”}

```
parent[“john”]
```
⇝ {“debby”, “bob”}

Product					
{}	{}				
{0}	{}	0			
{0}	{0}	0	0		
…					
{(0,0),(0,1)}	{(1,2)}	0	0	1	2
{(0,0),(0,1)}	{(1,2)}	0	1	1	2
…					

```
Product[U,V]
```
⇝ the Cartesian product of U and V

```
Product[U]
```
⇝ maps any V on the product of U and V

```
Product[U][V]
```
⇝ the Cartesian product of U and V

It’s sugar:

```
def ProductU({V},x…) : Product(U,V,x…)
```

Shortest Path

relationalAI

```
def APSP({V},{E},x,y,0) : V(x) and V(y) and x = y
def APSP({V},{E},x,y,k) :
        exists ((z in V) | E(x,z) and APSP[V,E](z,y,k-1)) and
        and not exists ((l in Int) | l < k and APSP[V,E](x,y,l))
```

This becomes more succinct with

- abstraction
- aggregates

Abstraction (sugar)

```
{(x,y) : mother(x,y) or father(x,y)}
```

⇝ defines an anonymous relation

You can give it a name if you want:

```
def parent
        {(x,y) : mother(x,y) or father(x,y)}
```

Aggregation and Reduce

relationalAI

The Standard Library has `reduce`

It has tuples (F, R, v) such that v is obtained by "aggregating" the values in the last column of R using the function F

Now we can define aggregation!

```
def sum[{A}]    : reduce[add,A]
def count[{A}] : reduce[add,(A,1)]
def min[{A}]    : reduce[minimum,A]
def max[{A}]    : reduce[maximum,A]
def avg[{A}]    : sum[A] / count[A]
```

relationalAI

Shortest Path

```
def APSP({V},{E},x,y,0) : V(x) and V(y) and x = y
def APSP({V},{E},x,y,k) :
         exists ((z in V) | E(x,z) and APSP[V,E](z,y,k-1)) and
         and not exists ((l in Int) | l < k and APSP[V,E](x,y,l))
```

With abstraction and aggregates:

```
def APSP({V},{E},x,y,0) : V(x) and V(y) and x = y
def APSP({V},{E},x,y,k) :
         k = min[{l : exists ((z in V) | E(x,z) and APSP[V,E](z,y,l-1))}]
```

aggregate

abstraction

Defining Relational Algebra

Again, let's have some sugar to improve readability

We already know:

```
def grandparent(x,y) :
      exists ((z) | parent(x,z) and parent(z,y))
```

"*y* is a grandparent of *x* if ..."

We can also write:

```
def grandparent[x] :
      {y : exists ((z) | parent(x,z) and parent(z,y))}
```

"the set of grandparents of *x* is ..."

Defining Relational Algebra

relationalAI

```
def Product[{A},{B}] :
          {(x…,y…) : A(x…) and B(y…)}
```

```
def (,)[{A},{B}] :
          {(x…,y…) : A(x…) and B(y…)}
```

⇝

```
R,S
```

```
def Minus[{A},{B}] :
          {(x…) : A(x…) and not B(x…)}
```

```
def Union[{A},{B}] :
          {(x…) : A(x…) or B(x…)}
```

(Select and project we can already do with abstraction)

PageRank

Step 1: Matrix multiplication

```
def MatrixMult[{A},{B},i,j] : { sum[[k] : A[i,k]*B[k,j]] }
```

Step 2: Prelims

```
def dimension[{M}] : max[(k) : M(k,_,_)]
def vector[d,i,j]  : 1/d, range(1,d,1,i), j = 1
```

Step 3: PageRank

```
def PageRank[{G},0,i,j] : vector[dimension[G]]
def PageRank[{G},k,i,j] : k>0, MatrixMult[G,PageRank[G,k-1],i,j]
```

```
def output {PageRank[M,10]}
```

⇝ 10 iterations of PageRank on matrix *M*

Rel is Already Handling Large Applications

relationalAI

Rel in the Real World

- RAI is actively using Rel with about a dozen customers
- Hundreds are inline

- Rel models the **semantics of the whole domain**
  - It is replacing arbitrary Java / C# code
- Codebase becomes 20 - 50x smaller
  - E.g. 800k lines of C# ⇝ 15k lines of Rel
  - 205k lines of C++ ⇝ 9k lines of Rel
- Performance goes up
  - E.g. 1 month ⇝ a few hours of processing time

It can be efficient enough!

What I Skipped

Inserts / deletes

Standard library

Type system

Code structuring features

...

Integrity constraints

Bonus Motivation
(for Theoreticians)

Database Research Landscape

Theory

divide

why?

Systems

Bag Semantics

PODS 1993

Optimization of *Real* Conjunctive Queries

Surajit Chaudhuri Moshe Y. Vardi

Abstract

The optimization problem for conjunctive queries has been studied extensively. Unfortunately, this research almost invariably assumes set-theoretic semantics (i.e., duplicates are eliminated). In contrast, SQL queries have bag-theoretic semantics (i.e., in general duplicates are not eliminated). In this paper we study the optimization problems for conjunctive queries under bag-theoretic semantics. We show that optimization techniques from the set-theoretic setting do not carry over to the bag-theoretic setting.

But *why* do we have bag semantics?

Prompt:
Why do databases use bag semantics instead of set semantics?

ChatGPT gave me 11 reasons:

- 8x: bad modeling / schema design
- 2x: efficiency (inserts, union, projection)
- 1x: historic (first SQL database had it)

Summary:

- Bag semantics align more closely with the nature of real world data ⇝ false
- Bag semantics enable efficient union and projection ⇝ true, but it kills optimization
- SQL does it ⇝ circular argument

How Rel Closes This Gap

The design of Rel goes back to first principles

- Relations are fully normalized (6NF) ⇝ actually, Graph Normal Form
- Set semantics!
- No nulls! ⇝ Hoare: “My billion dollar mistake”

Great for research:

- this model is much more elegant than the alternative
- we like to study clean and elegant models!

The Big Challenge:

- make this efficient!

Set semantics allows for more optimization than bag semantics!

Questions?