

Rel: A Programming Language for Relational Data

Wim Martens

RelationalAI
University of Bayreuth

FMT 2025

Rel: A Programming Language for Relational Data

Molham Aref

Mary McGrath

Cristina Sirangelo

Liat Peterfreund

Allison Rogers

Leonid Libkin

Filip Murlak

Victor Marsault

Nathaniel Nystrom

David Zhao

Wim Martens

Paolo Guagliardo

George Kastrinis

Abdul Zreika

Domagoj Vrgoč

Special thanks:

Martin Bravenboer

FMT 2025

Rel: A Programming Language for Relational Data

RelationalAI

Molham Aref

Mary McGrath

Cristina Sirangelo

Liat Peterfreund

Allison Rogers

Leonid Libkin

Filip Murlak

Victor Marsault

Nathaniel Nystrom

David Zhao

Wim Martens

Paolo Guagliardo

George Kastrinis

Abdul Zreika

Domagoj Vrgoč

Special thanks:

Martin Bravenboer

FMT 2025

Rel: A Programming Language for Relational Data

RelationalAI

Academia

Molham Aref

Mary McGrath

Cristina Sirangelo

Liat Peterfreund

Allison Rogers

Leonid Libkin

Victor Marsault

Nathaniel Nystrom

Filip Murlak

David Zhao

Wim Martens

Paolo Guagliardo

George Kastrinis

Abdul Zreika

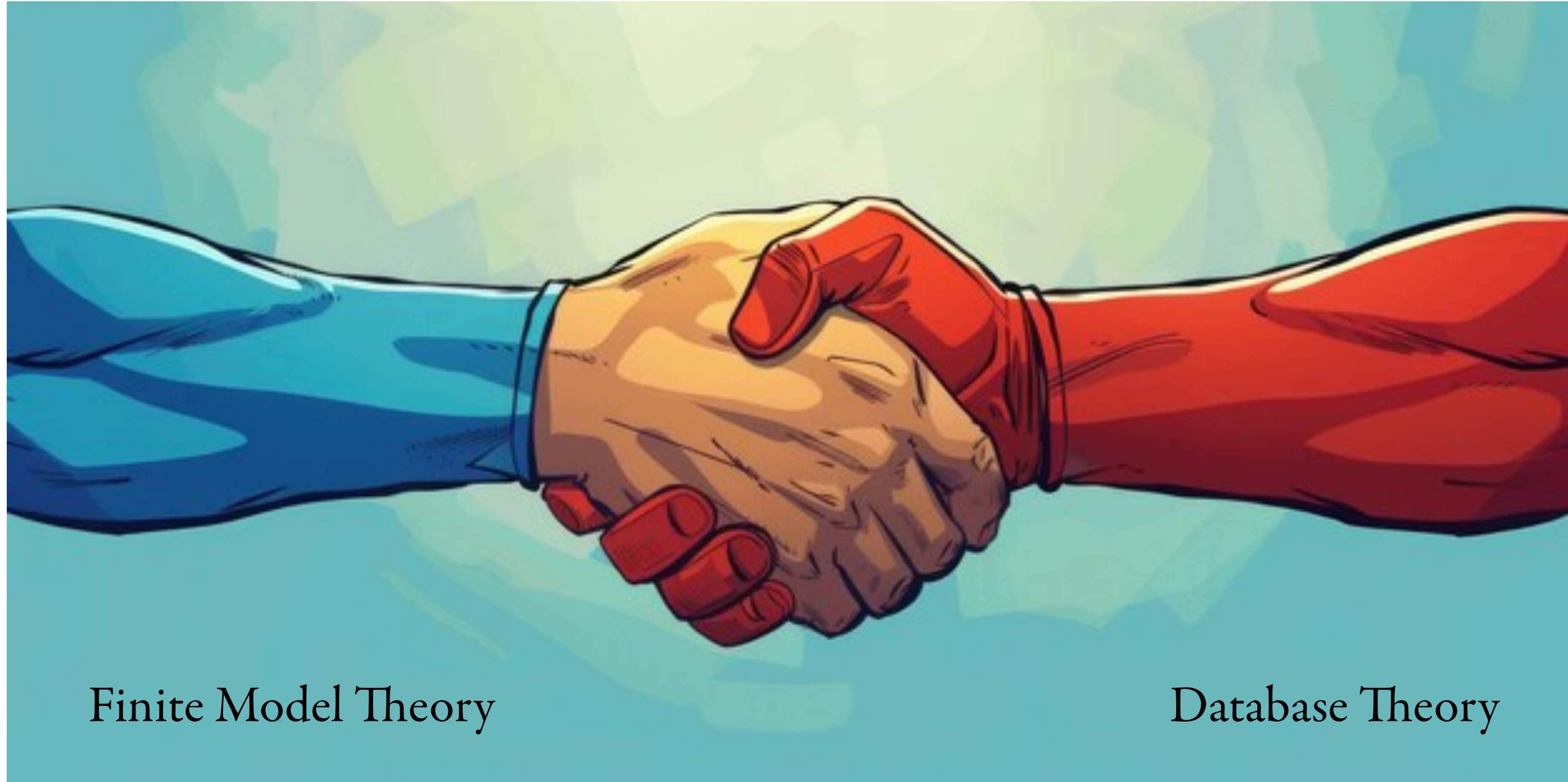
Domagoj Vrgoč

Special thanks:

Martin Bravenboer

FMT 2025

Why I Think You Should Be Excited



Finite Model Theory

Database Theory

Database Research Landscape

Theory

Systems



Database Research Landscape



Theory

Systems

divide

why?

Database Research Landscape

Theory

Sets

divide

why?

Systems

Bags
(Multisets)

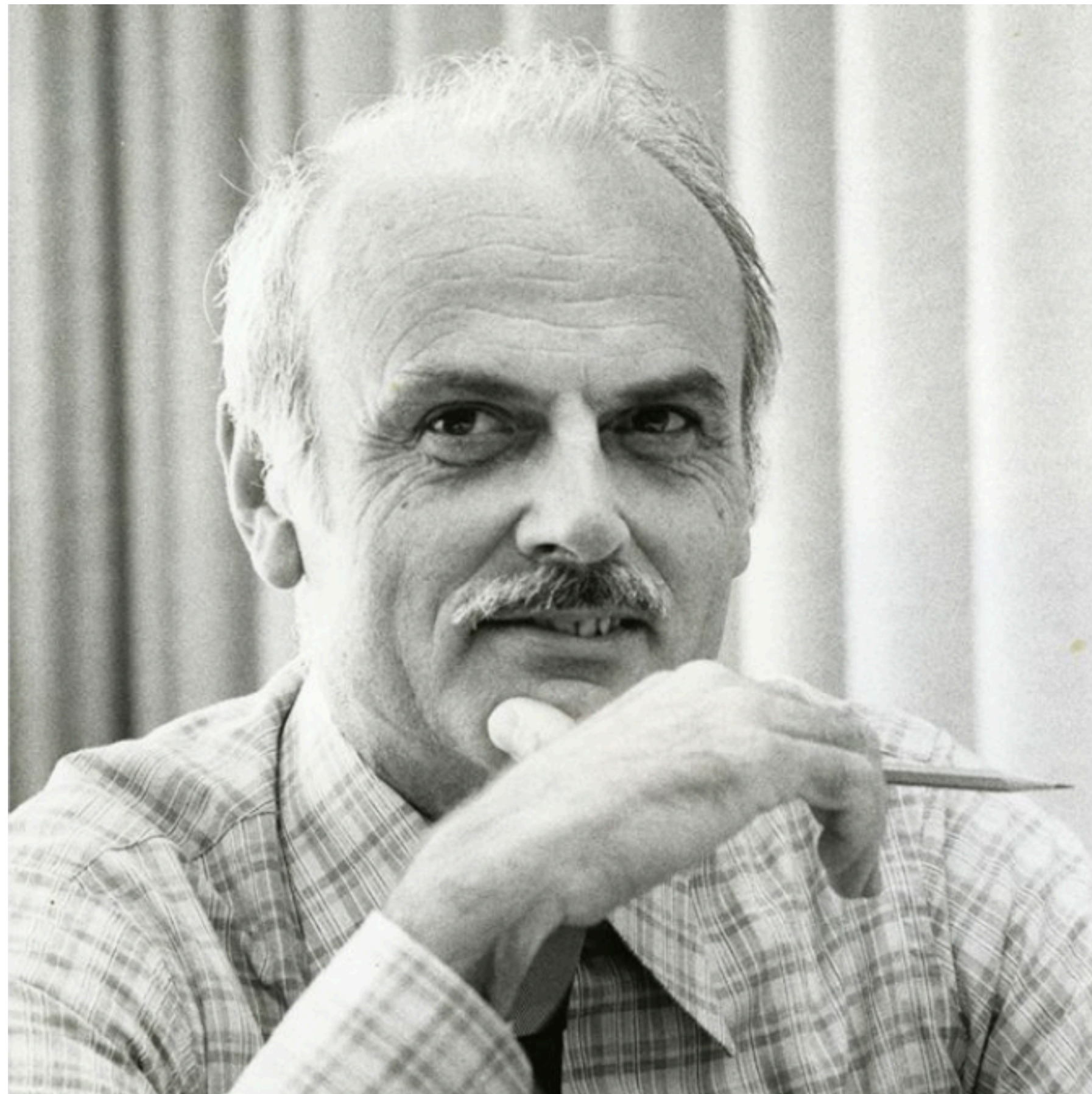
What Can You Do, If...

...You Build a Database (and More) Engine

- From the ground up
- From the principles that the theory community believes are the right ones
 - Set semantics!

???

Databases: The Origin Story



(Image: IBM, fair use)

1971:

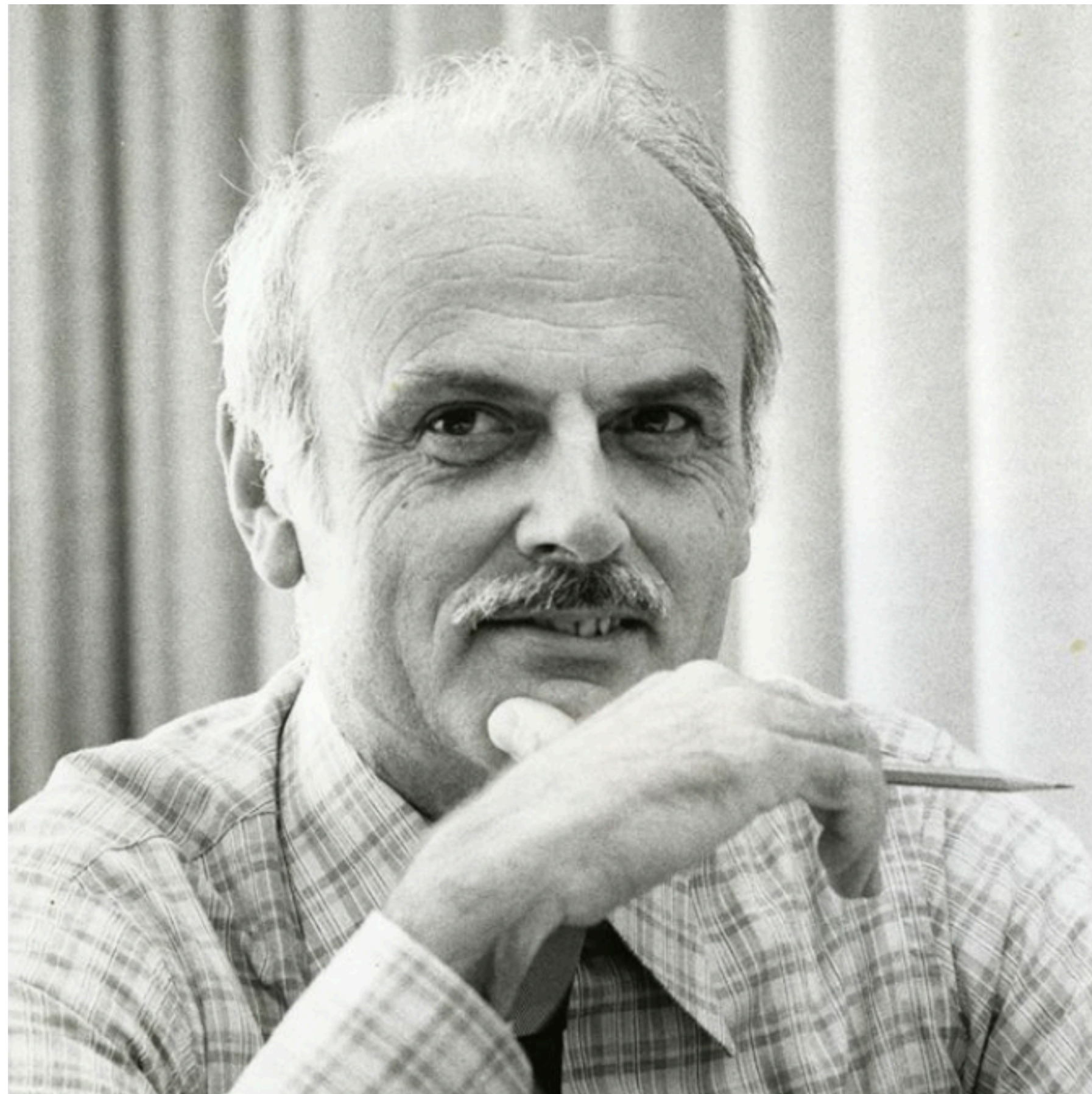
A DATA BASE SUBLANGUAGE FOUNDED ON
THE RELATIONAL CALCULUS

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



(Image: IBM, fair use)

1971:

A DATA BASE SUBLANGUAGE FOUNDED ON
THE RELATIONAL CALCULUS

by

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

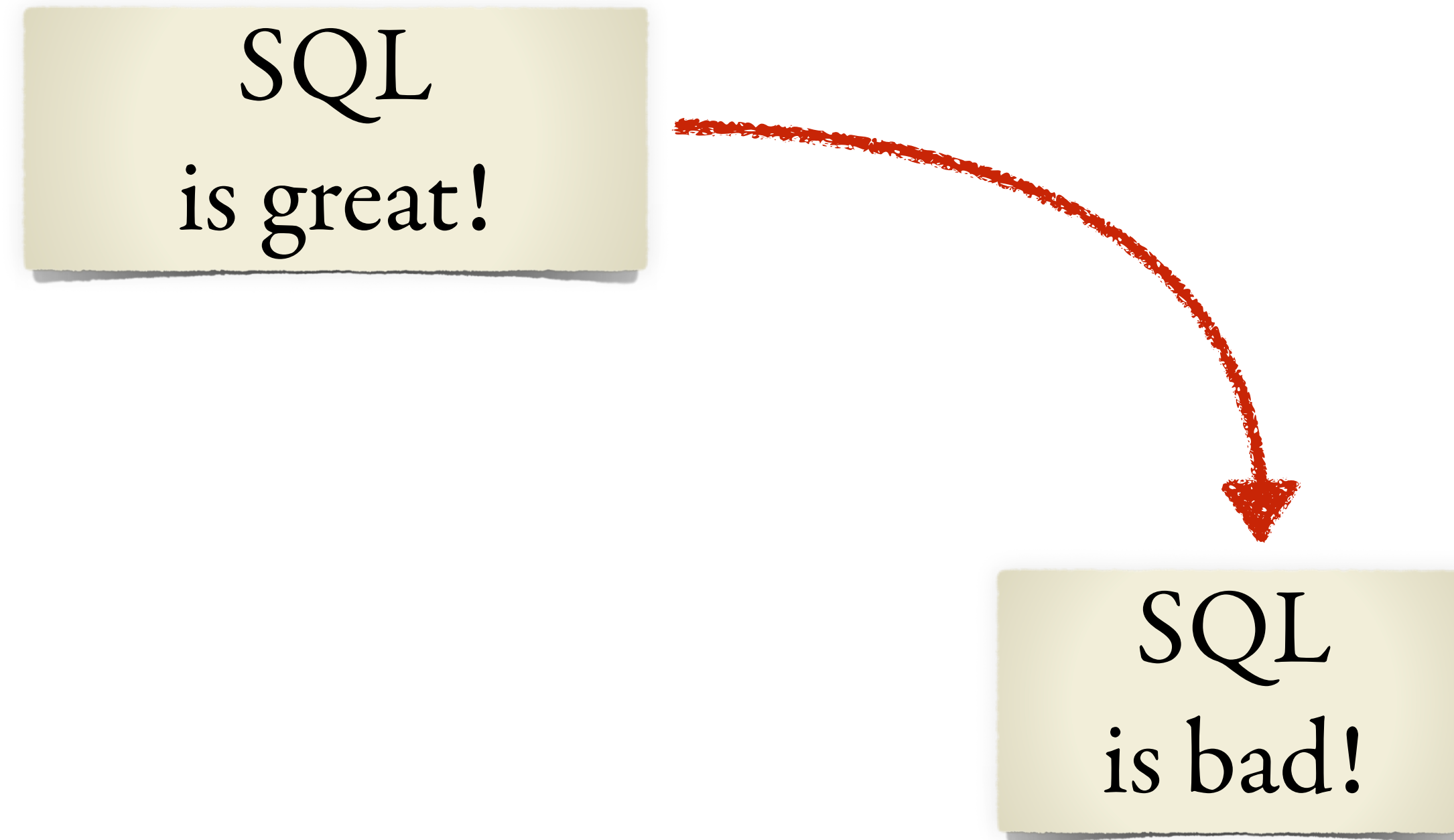
We use the term data sublanguage (rather than language) because we are not concerned with general processing (or the capability of computing all computable functions). Instead, we wish to focus on only those language components which support storage and retrieval of formatted data from large shared data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

What Goes Around Comes Around

relationalAI

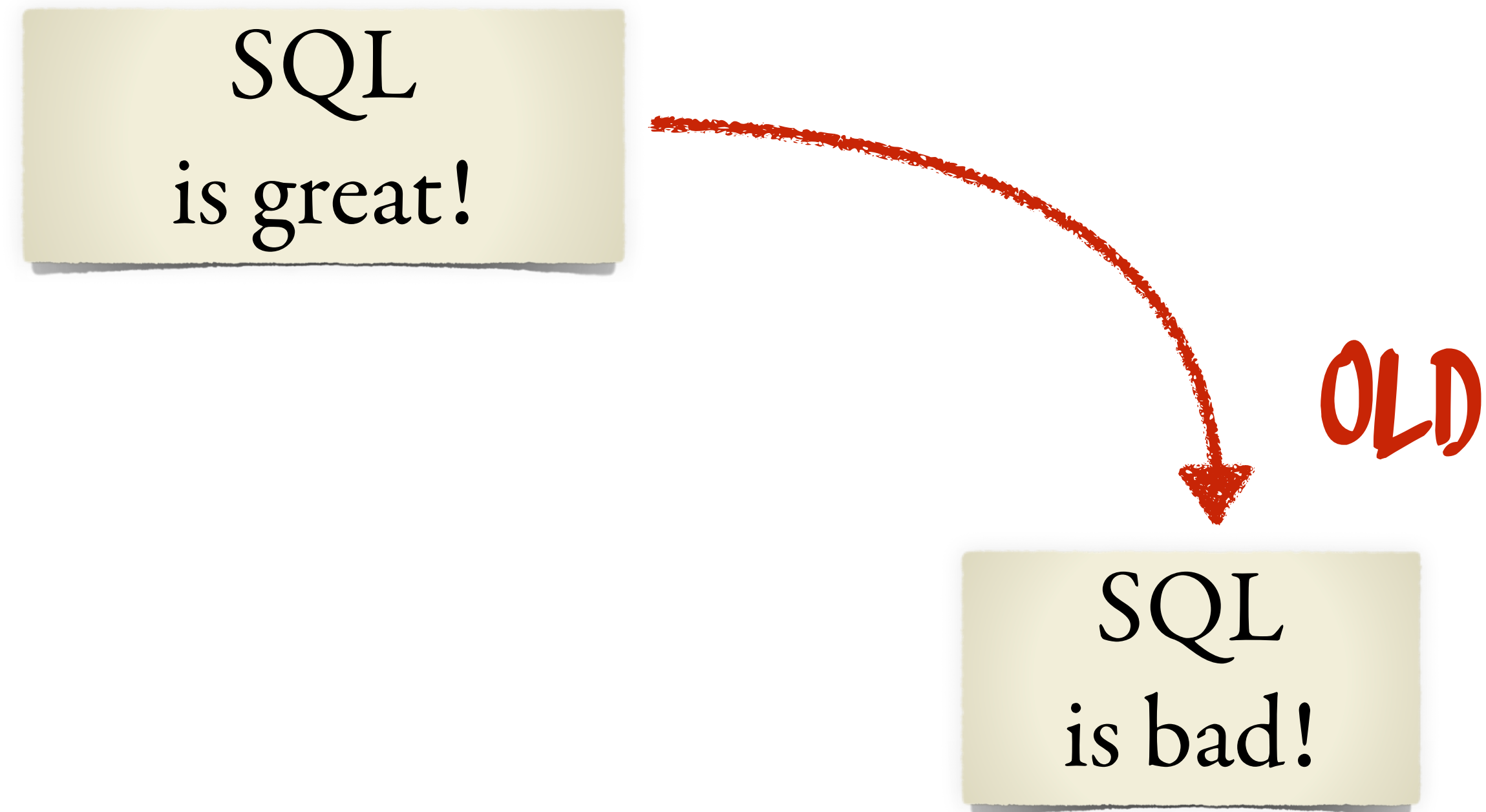
SQL
is great!

What Goes Around Comes Around



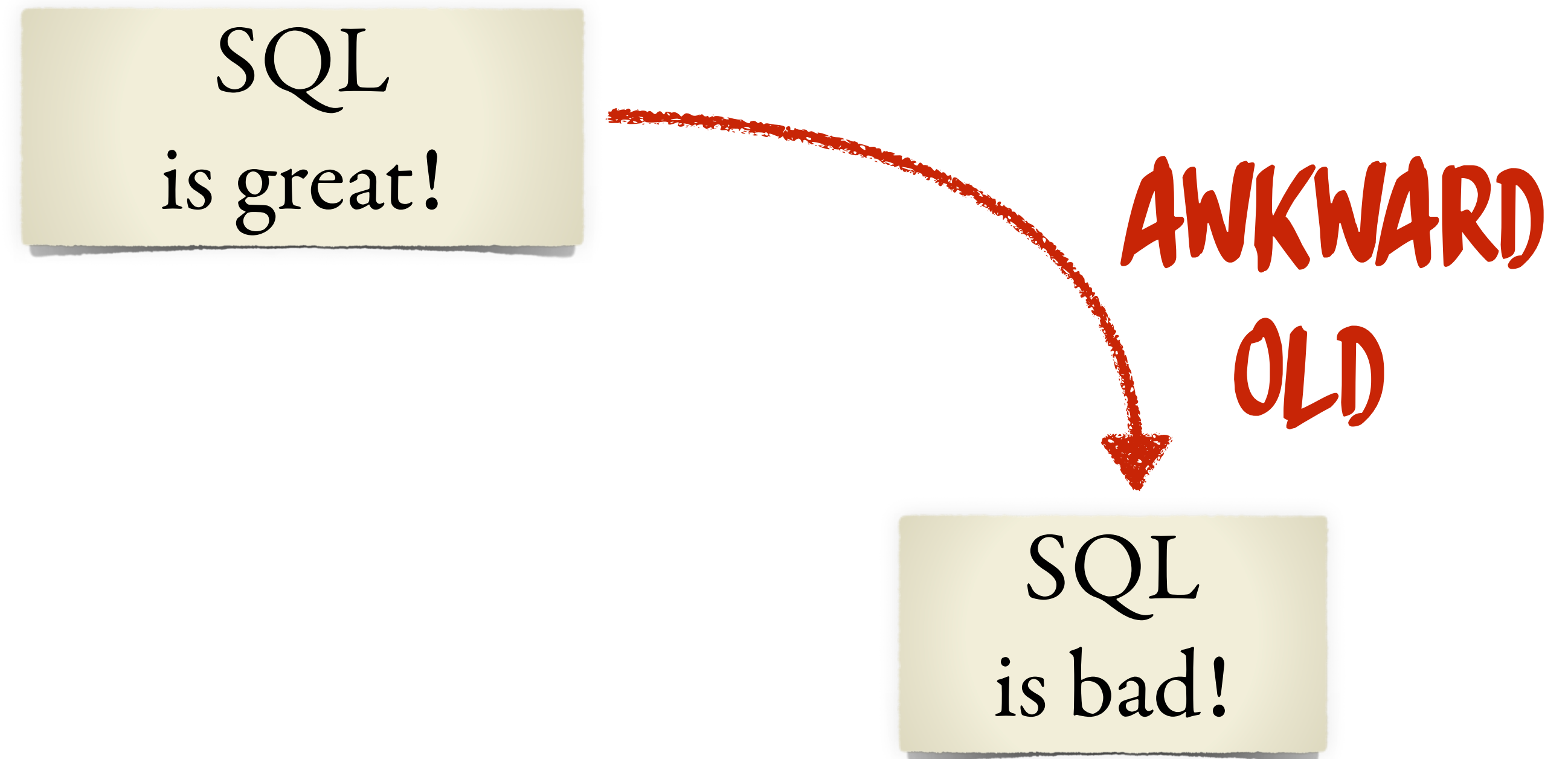
What Goes Around Comes Around

relationalAI



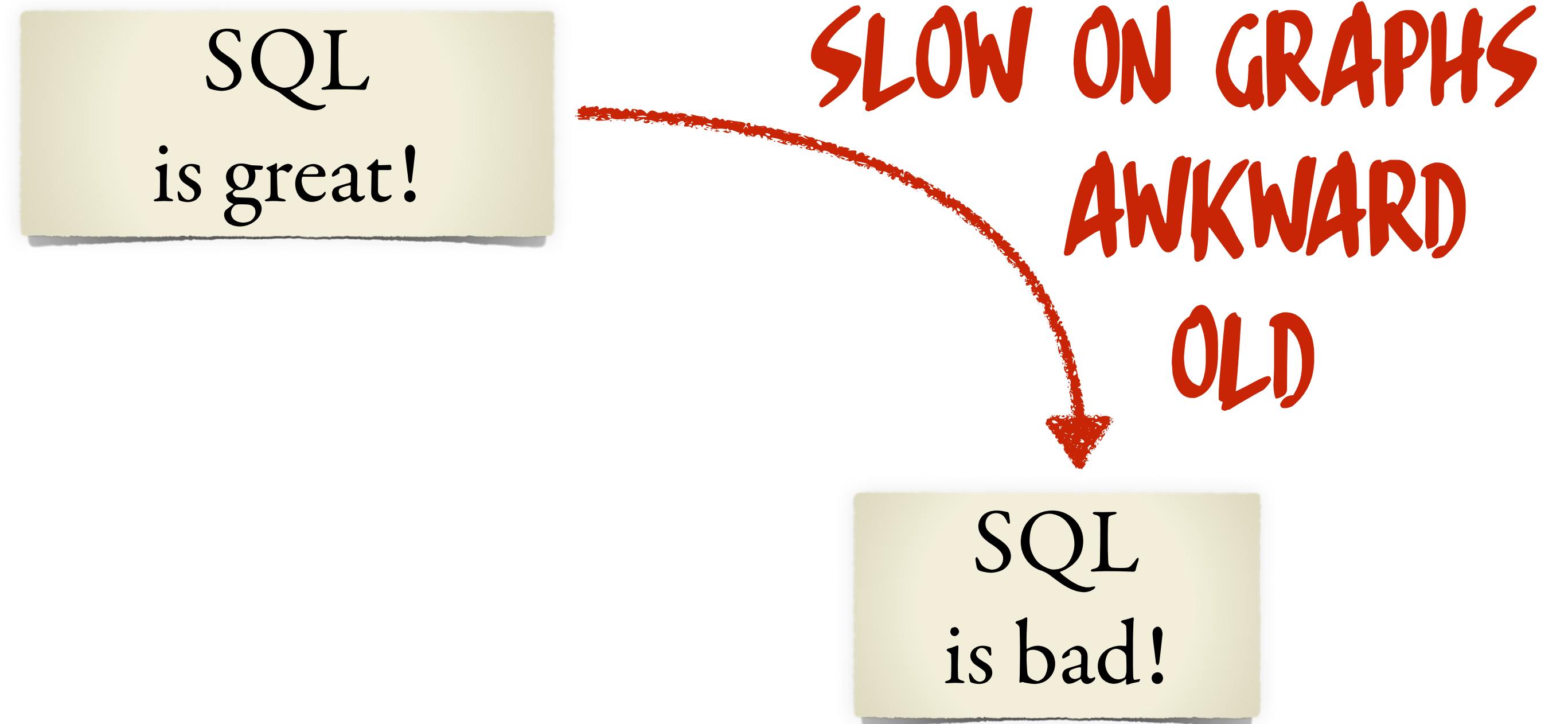
What Goes Around Comes Around

relationalAI



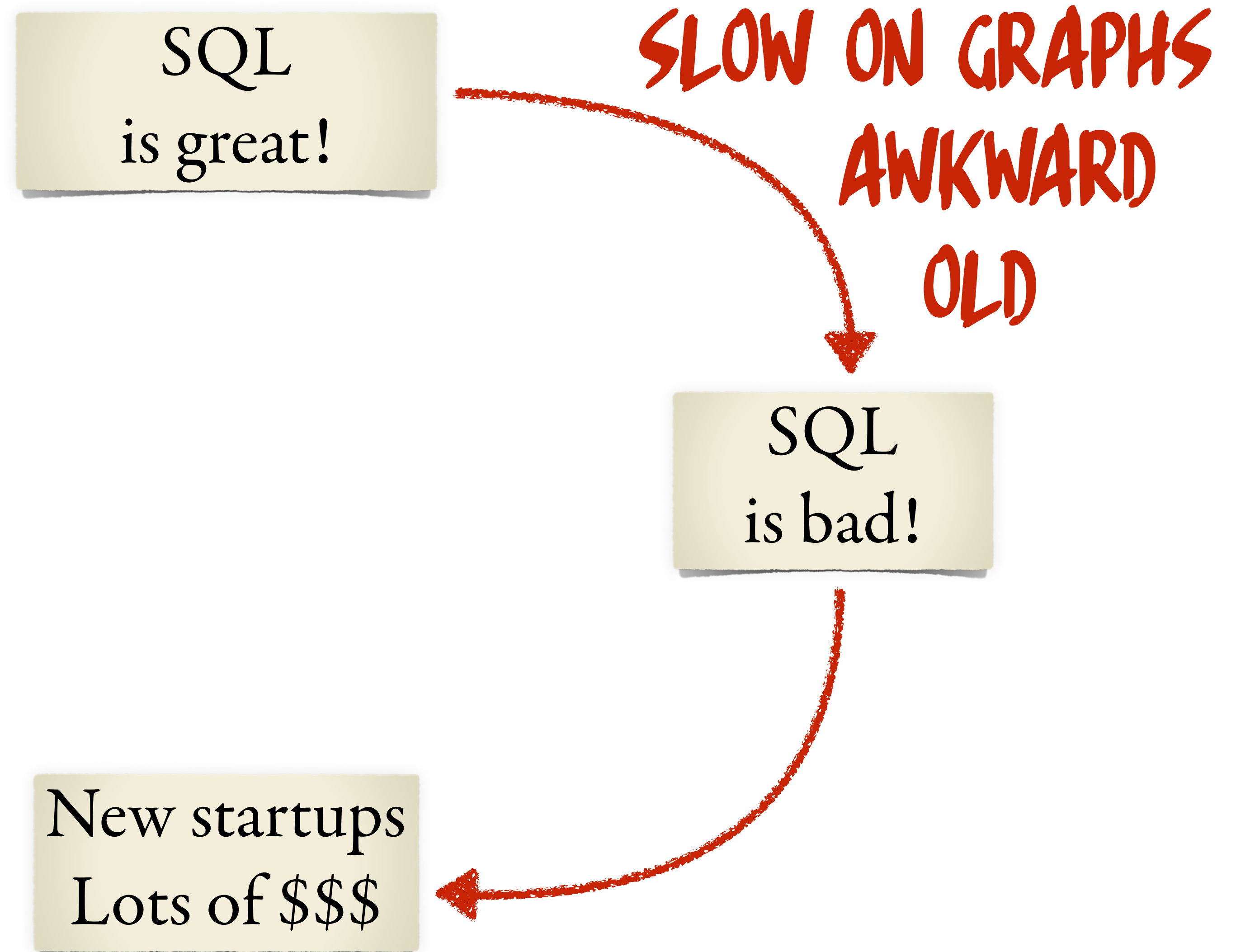
What Goes Around Comes Around

relationalAI



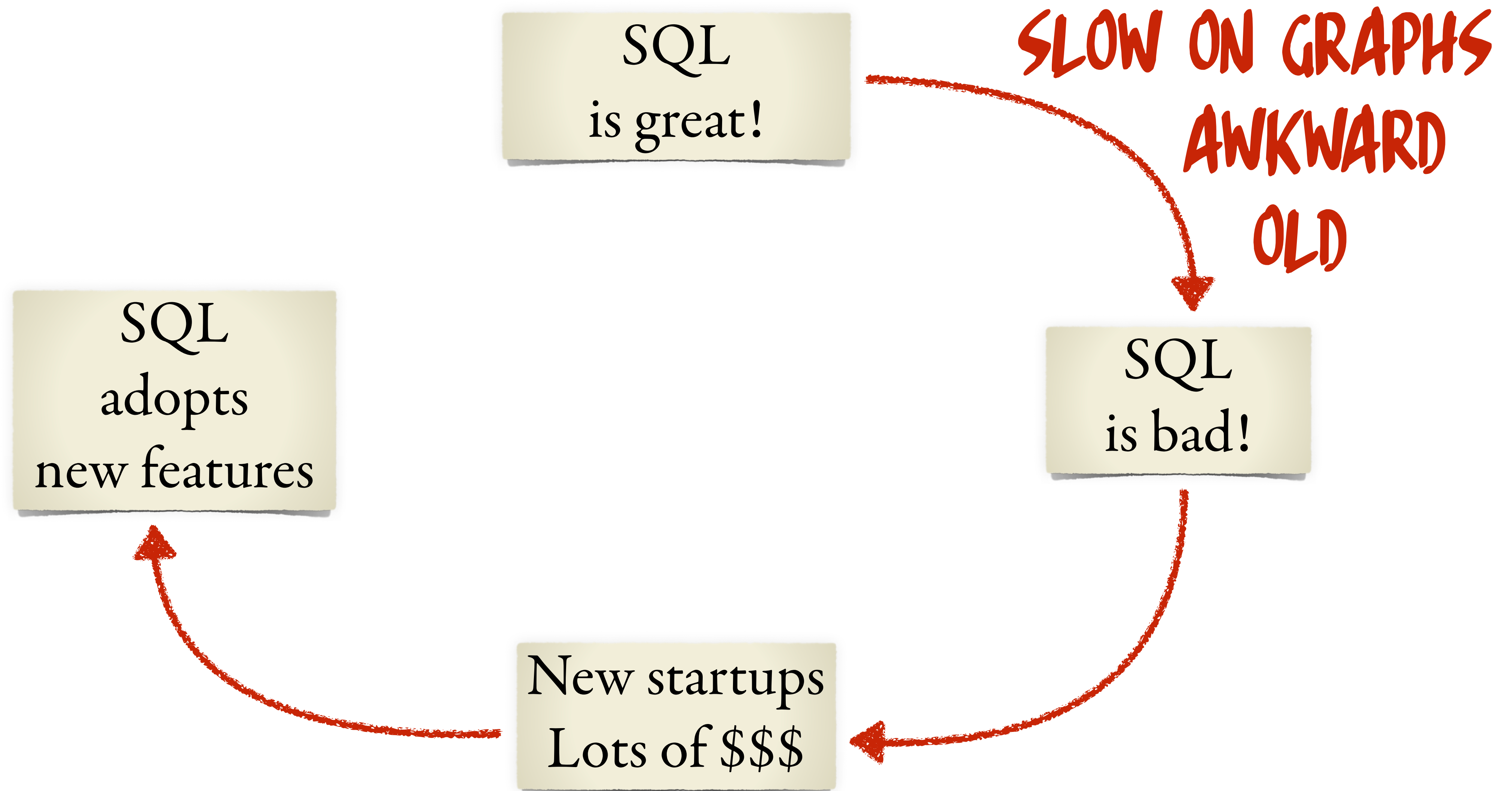
What Goes Around Comes Around

relationalAI

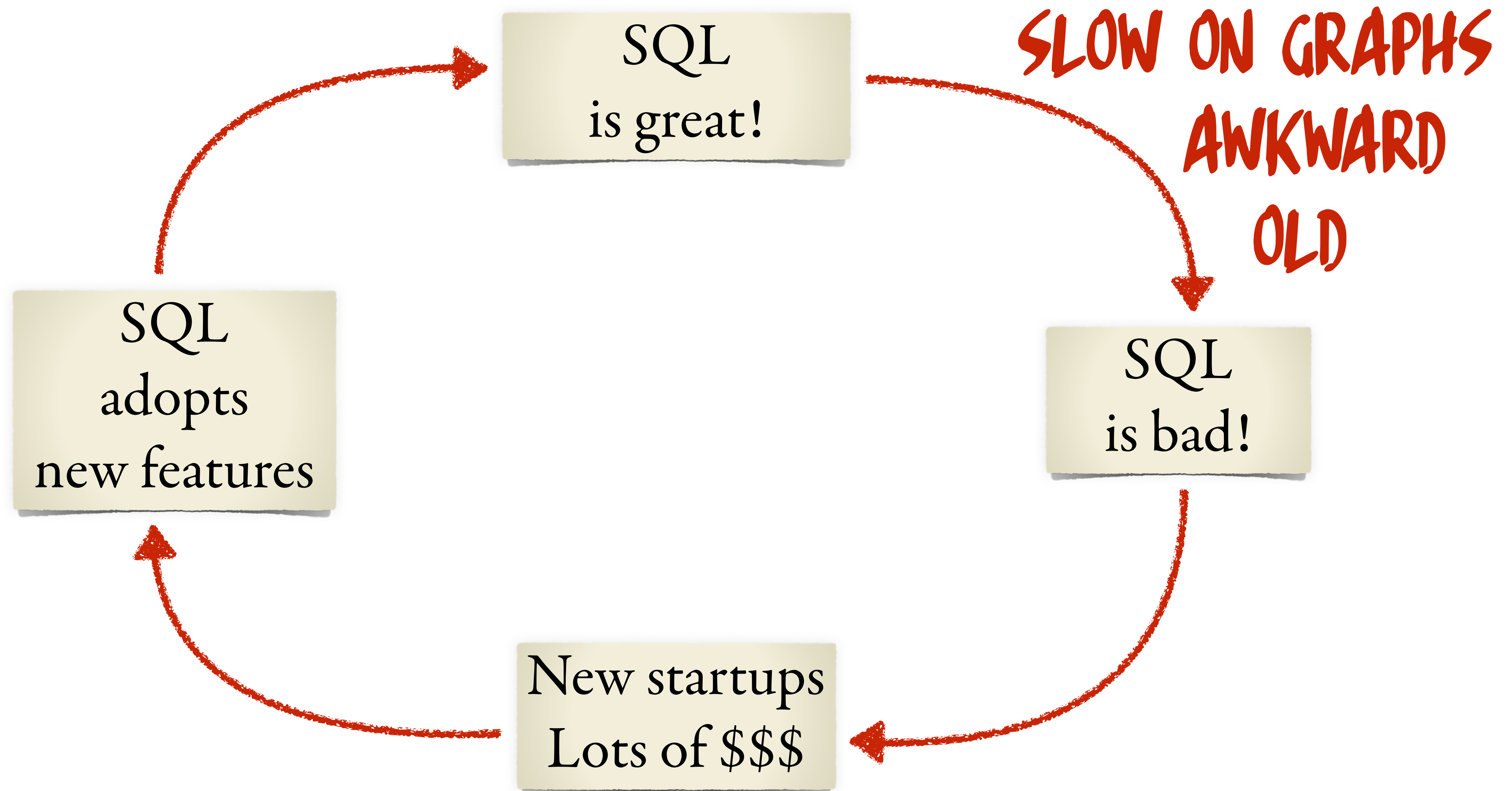


What Goes Around Comes Around

relationalAI



What Goes Around Comes Around relationalAI

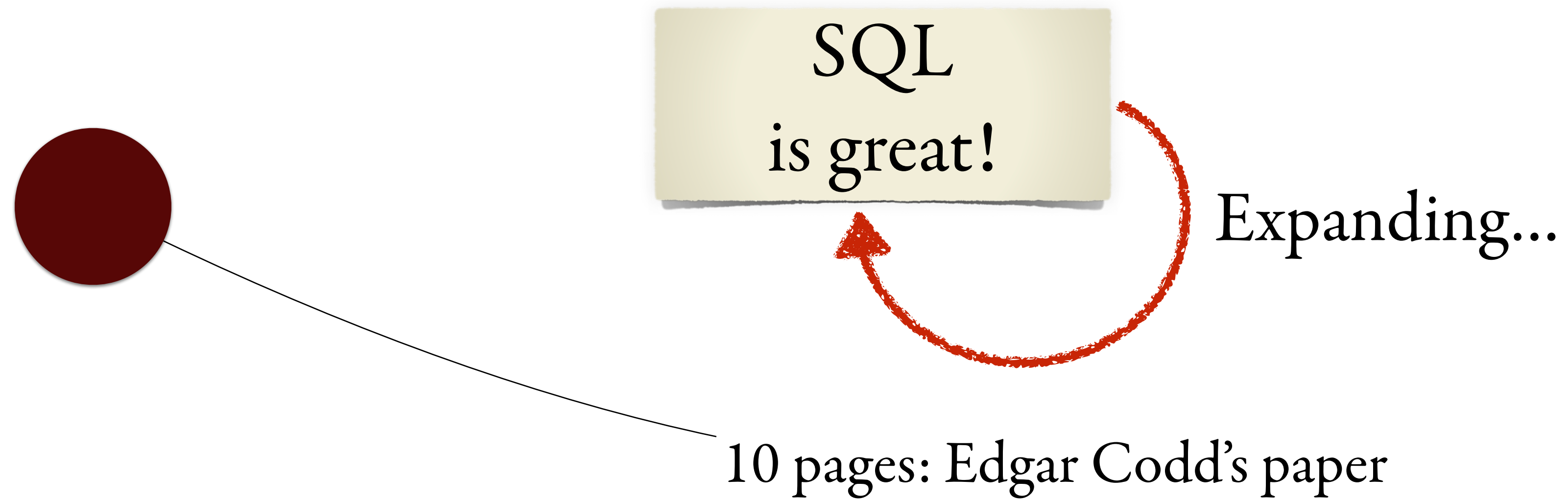


What Goes Around Comes Around

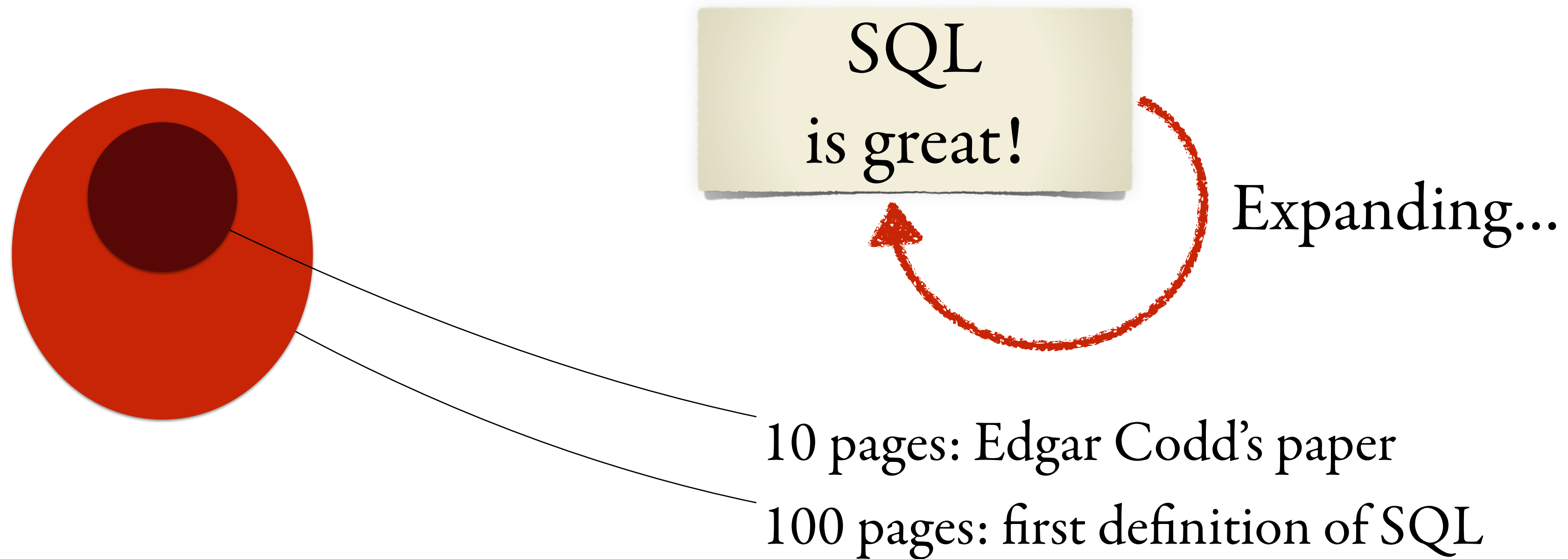
relationalAI



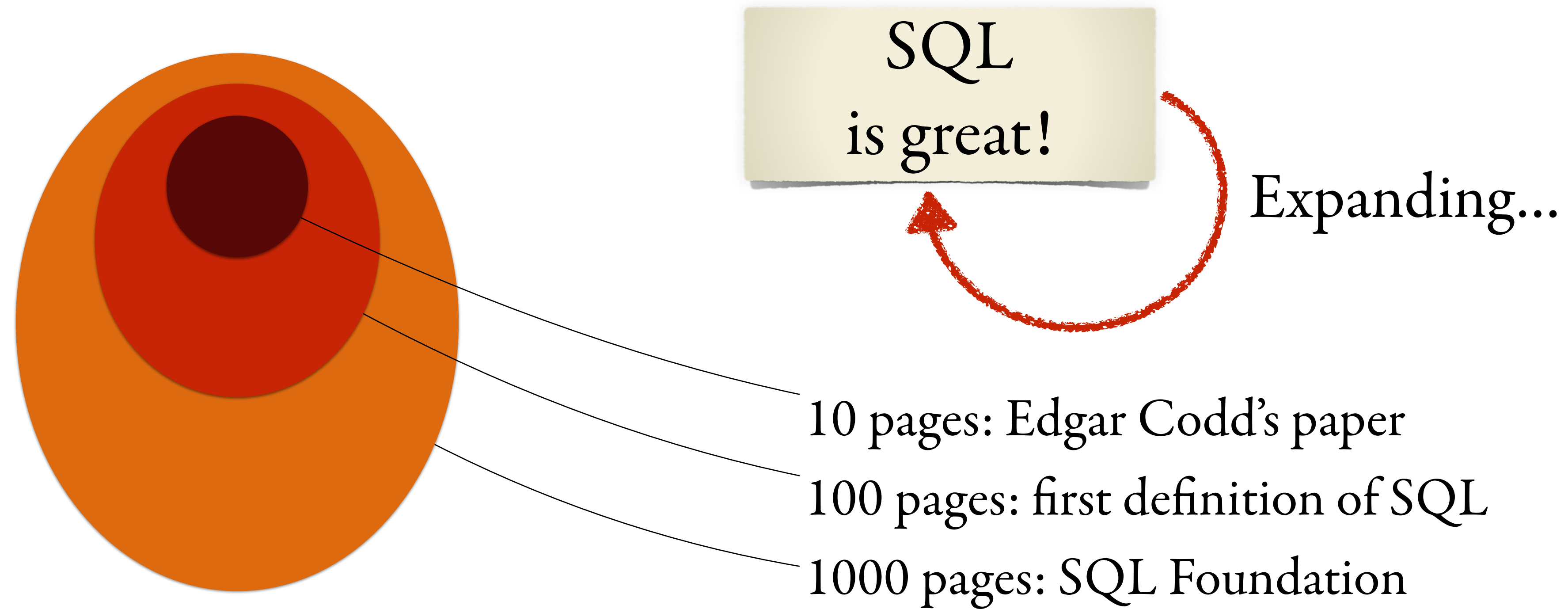
What Goes Around Comes Around



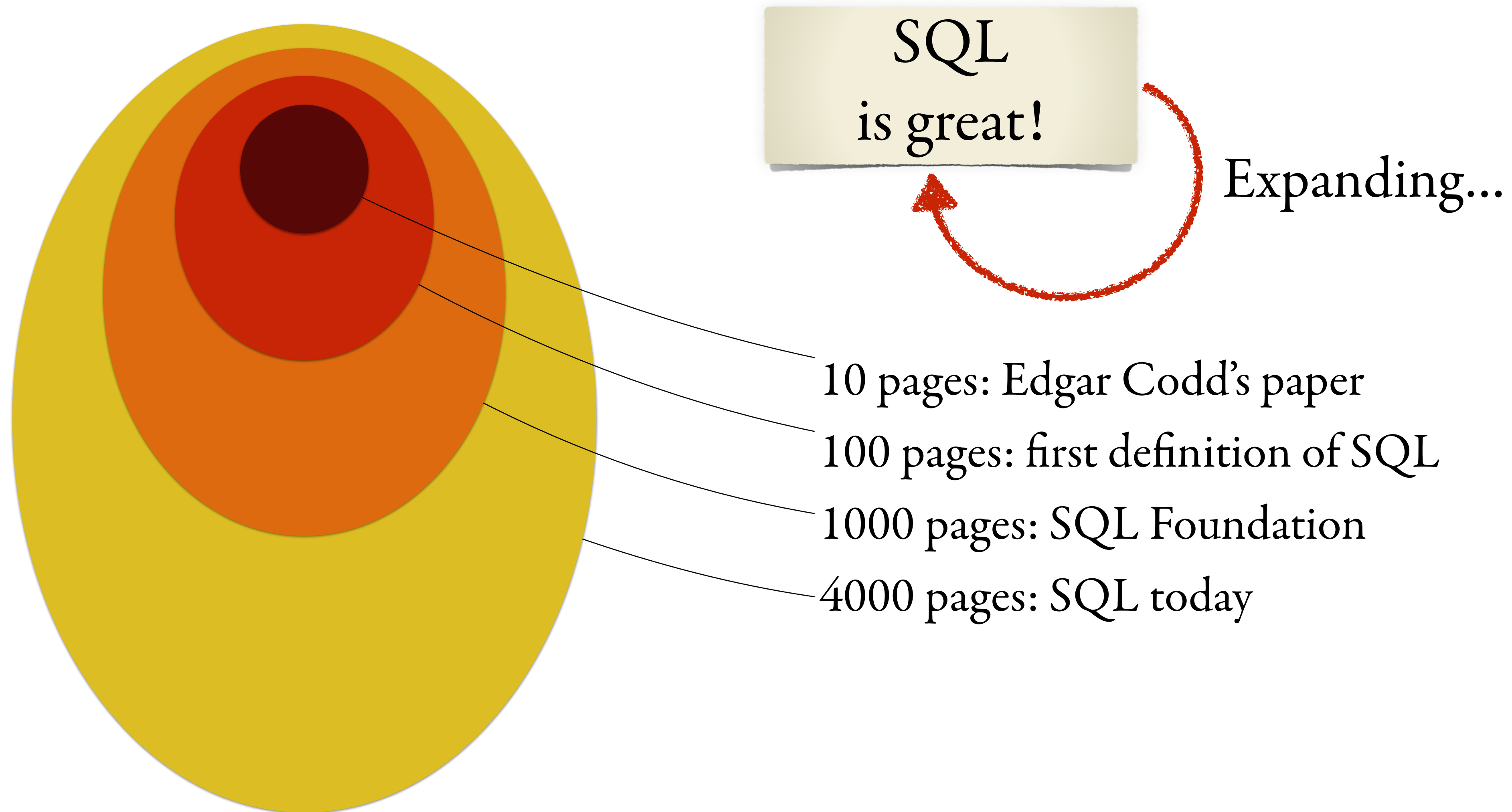
What Goes Around Comes Around



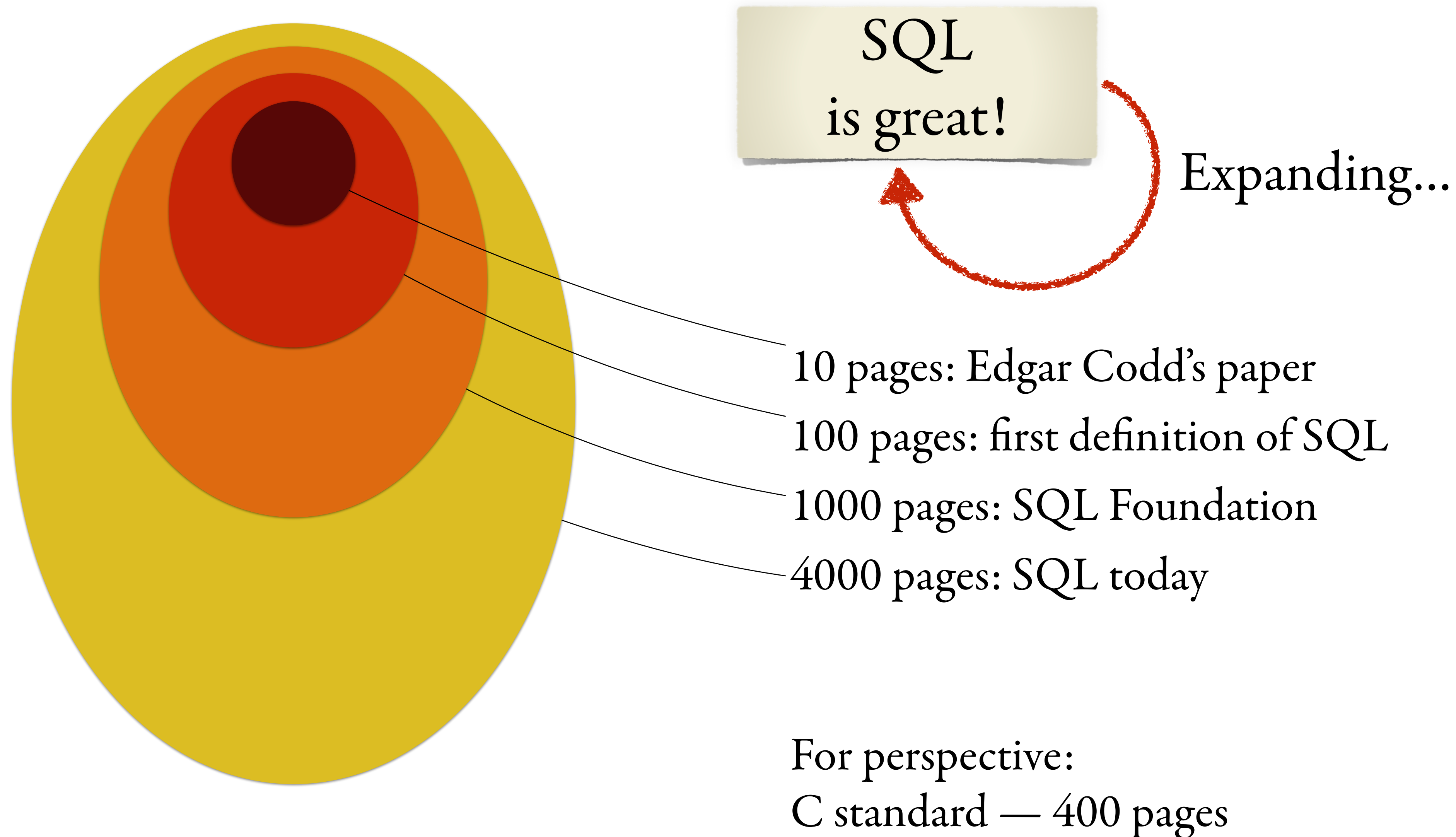
What Goes Around Comes Around



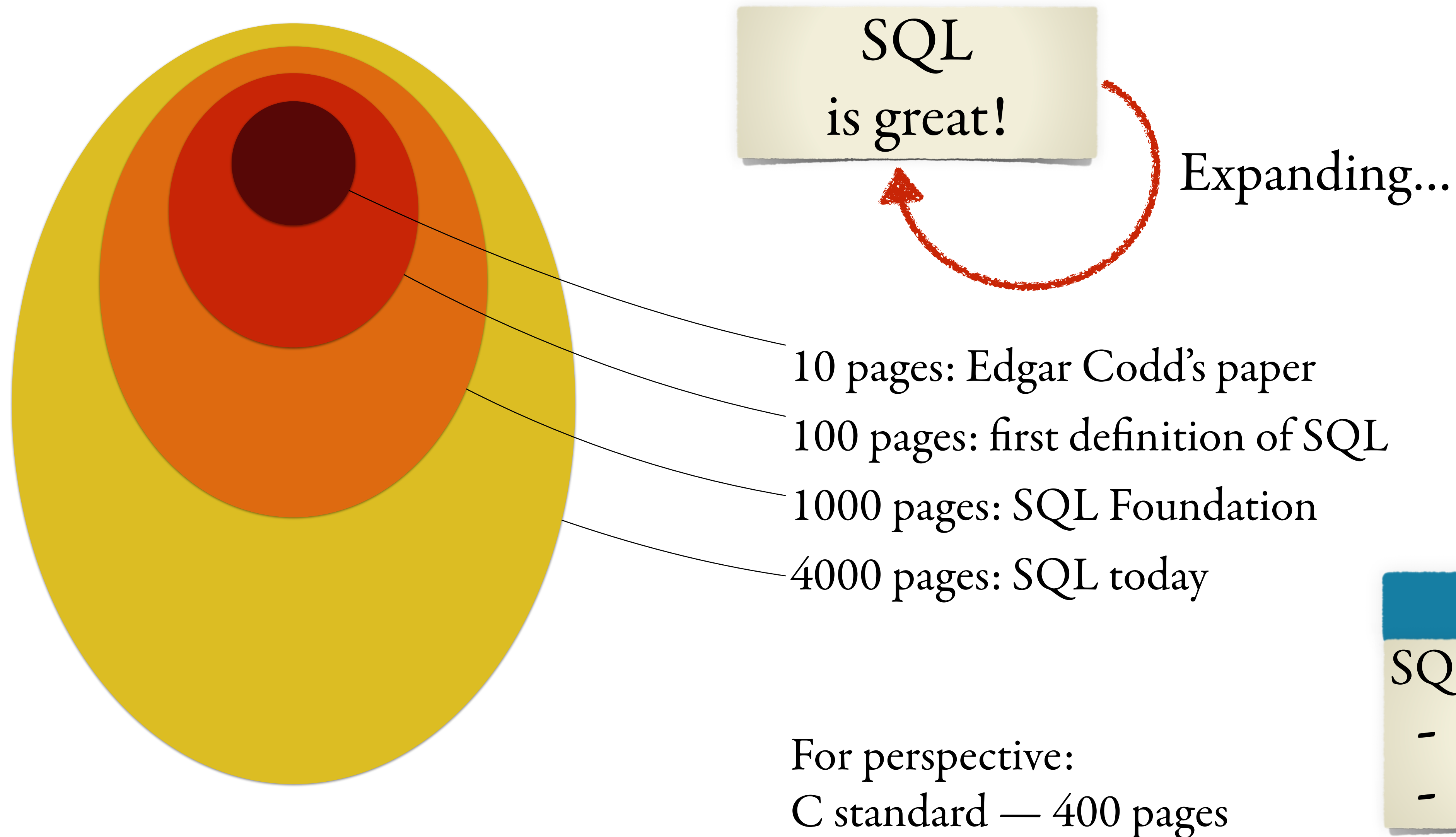
What Goes Around Comes Around



What Goes Around Comes Around



What Goes Around Comes Around relationalAI



Why?

SQL...

- is a sublanguage
- can't do **libraries**

So We Asked Ourselves

Can we design...

...a query / programming language based on the tried and trusted DB principles

- being declarative
- using the relational model
 - with set semantics (!)

but can do full-fledged programming, including libraries

So We Asked Ourselves

Can we design...

...a query / programming language based on the tried and trusted DB principles

- being declarative
- using the relational model
 - with set semantics (!)

but can do full-fledged programming, including libraries

This would be nice, because...

So We Asked Ourselves

Can we design...

...a query / programming language based on the tried and trusted DB principles

- being declarative
- using the relational model
 - with set semantics (!)

but can do full-fledged programming, including libraries

This would be nice, because...

Database engine

can do query optimization everywhere
(not just over the database part of applications)

So We Asked Ourselves

Can we design...

...a query / programming language based on the tried and trusted DB principles

- being declarative
- using the relational model
 - with set semantics (!)

but can do full-fledged programming, including libraries

This would be nice, because...

Database engine

can do query optimization everywhere
(not just over the database part of applications)

Why is this a real advantage?

Solving the Impedance Mismatch

Solving the Impedance Mismatch



(<https://www.instagram.com/uglybelgianhouses>)

Solving the Impedance Mismatch

Database

Business Logic

Solving the Impedance Mismatch

Database

Query Language

- declarative
- (serious!) query optimization
- automatic parallelization
- automatic out-of-core computation
- ...

Business Logic

Solving the Impedance Mismatch

Database

Query Language

- declarative
- (serious!) query optimization
- automatic parallelization
- automatic out-of-core computation
- ...

Business Logic

General Purpose Language

- usually imperative
- invisible to your query optimizer

Solving the Impedance Mismatch

Let's bring this together!

Database

Query Language

- declarative
- (serious!) query optimization
- automatic parallelization
- automatic out-of-core computation
- ...

Business Logic

General Purpose Language

- usually imperative
- invisible to your query optimizer

Rel:

A Programming Language for Relational Data



Rel Basics

Base Relations

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

the mother/father of x is y

Rel code:

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

Ingredients

- Datalog rules
- FO in the bodies



Rel Basics

Base Relations

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

the mother/father of x is y

Rel code:

```
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), ...`

Infinite Relations



Infinite Relations

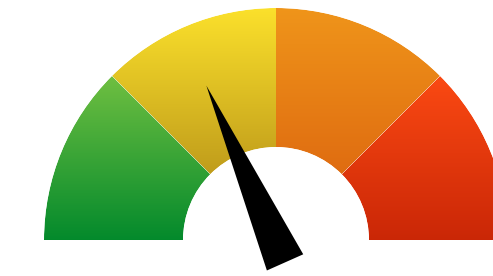
- `Int(x), ...`
- `>, =, >=, ...`
- `add(x,y,z), multiply(x,y,z), modulo(x,y,z), ...`

```
def add_inverse(x,y) : add(x,y,0)
```

Equivalent:

```
def add_inverse(x,y) : x + y = 0
```

Infinite Relations



Infinite Relations

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

```
def add_inverse(x,y) : add(x,y,0)
```

Equivalent:

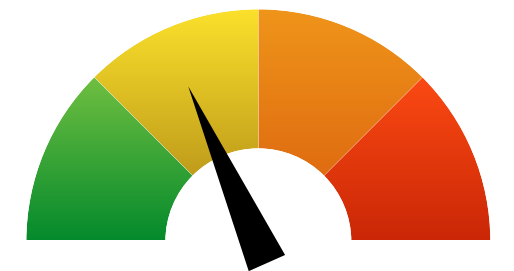
```
def add_inverse(x,y) : x + y = 0
```

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

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



What is added to enable programming in the large?

Four Extra Features

to enable Relational Programming

Four Extra Features

to enable Relational Programming

Tuple Variables

Four Extra Features

to enable Relational Programming

Tuple Variables

Relation Variables

Four Extra Features

to enable Relational Programming

Tuple Variables

Relation Variables

Relational Application

Four Extra Features

to enable Relational Programming

Tuple Variables

Relation Variables

Relational Application

Abstraction

Four Extra Features

to enable Relational Programming

Tuple Variables

Relation Variables

Relational Application

Abstraction

So, let's look at some code examples

Details \rightsquigarrow the paper

Relational Algebra as a Library

How do we write generally applicable code?

U

1	2
3	4
3	5

V

1	2
6	7

Cartesian 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

Relational Algebra as a Library

How do we write generally applicable code?

U

1	2
3	4
3	5

V

1	2
6	7

Cartesian 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 ProductUV {(a,b,c,d) : U(a,b) and V(c,d)}
```

Relational Algebra as a Library

How do we write generally applicable code?

U

1	2
3	4
3	5

V

1	2
6	7

Cartesian 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 ProductUV {(a,b,c,d) : U(a,b) and V(c,d)}
```

But what if *V* is ternary?

Relational Algebra as a Library

How do we write generally applicable code?

U

1	2
3	4
3	5

V

1	2
6	7

Cartesian 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 ProductUV {(a,b,c,d) : U(a,b) and V(c,d)}
```

But what if *V* is ternary?

```
def ProductUV {(a,b,c,d,e) : U(a,b) and V(c,d,e)}
```

Relational Algebra as a Library

How do we write generally applicable code?

U

1	2
3	4
3	5

V

1	2
6	7

Cartesian 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 ProductUV {(a,b,c,d) : U(a,b) and V(c,d)}
```

But what if *V* is ternary?

```
def ProductUV {(a,b,c,d,e) : U(a,b) and V(c,d,e)}
```

This is both tedious and not generally applicable. Solution:

Relational Algebra as a Library

How do we write generally applicable code?

U

1	2
3	4
3	5

V

1	2
6	7

Cartesian 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 ProductUV {(a,b,c,d) : U(a,b) and V(c,d)}
```

But what if *V* is ternary?

```
def ProductUV {(a,b,c,d,e) : U(a,b) and V(c,d,e)}
```

This is both tedious and not generally applicable. Solution:

```
def ProductUV {(x...,y...) : U(x...) and V(y...)}
```

Tuple variables bind to “sequences” of values (subtuples)

Relational Algebra as a Library

```
def ProductUV { (x..., y...) : U(x...) and V(y...) }
```


Relational Algebra as a Library

```
def ProductUV { (x..., y...) : U(x...) and V(y...) }
```

```
def ProductAB { (x..., y...) : A(x...) and B(y...) }
```

...

Relational Algebra as a Library

```
def ProductUV { (x... y...) : U(x...) and V(y...) }
```

```
def ProductAB { (x... y...) : A(x...) and B(y...) }
```

...

Relational Algebra as a Library

```
def ProductUV { (x..., y...) : U(x...) and V(y...) }
```

```
def ProductAB { (x..., y...) : A(x...) and B(y...) }
```

...

`Product[R, S]` $\rightsquigarrow R \times S$

Relational Algebra as a Library

```
def ProductUV { (x..., y...) : U(x...) and V(y...) }
```

```
def ProductAB { (x..., y...) : A(x...) and B(y...) }
```

...

`Product[R, S]` $\rightsquigarrow R \times S$

tuple variables

Cartesian Product

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

relation variables

Intermezzo: “Everything is a Relation”^{relationalAI}

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

Intermezzo: “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	0	
{(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

Intermezzo: “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	0	
{(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

Partial Application

Parent	
alice	cindy
john	debby
john	bob

Partial Application

Parent	
alice	cindy
john	debby
john	bob

```
Parent("alice", "cindy")
```

Partial Application

Parent	
alice	cindy
john	debby
john	bob

```
Parent("alice", "cindy")
```

 \rightsquigarrow true

Partial Application

Parent	
alice	cindy
john	debby
john	bob

```
Parent("alice", "cindy")
```

⇒ true

```
Parent["alice"]
```

Partial Application

Parent	
alice	cindy
john	debby
john	bob

```
Parent("alice", "cindy")
```

⇒ true

```
Parent["alice"]
```

⇒ {"cindy"}

Partial Application

Parent	
alice	cindy
john	debby
john	bob

```
Parent("alice", "cindy")
```

⇒ true

```
Parent["alice"]
```

⇒ {"cindy"}

```
Parent["john"]
```

Partial Application

Parent	
alice	cindy
john	debby
john	bob

```
Parent("alice", "cindy")
```

⇒ true

```
Parent["alice"]
```

⇒ {"cindy"}

```
Parent["john"]
```

⇒ {"debby", "bob"}

Partial Application

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	0	
{(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]

...

Partial Application

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	0	
{(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

Partial Application

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	0	
{(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]

...

Partial Application

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	0	
{(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 to the product of U and V

Partial Application

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	0	
{(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 to the product of U and V

Product[U][V]

...

Partial Application

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	0	
{(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 to the product of U and V

Product[U][V]

↪ the Cartesian product of U and V

...

Partial Application

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	0	
{(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 to the product of U and V

Product[U][V]

⇒ the Cartesian product of U and V

It looks like sugar...

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

Partial Application

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 0

{(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 to the product of U and V

Product[U][V]

↪ the Cartesian product of U and V

It looks like sugar...

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

but it's not:

Product[U] can occur as a subexpression with free variable u

All Pairs 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 ((j in Int) | j < k and APSP[V,E](x,y,j))
```

All Pairs 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 ((j in Int) | j < k and APSP[V,E](x,y,j))
```

.....▶ shortest path has length 0 :  
 x y

All Pairs 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 ((j in Int) | j < k and APSP[V,E](x,y,j))
```

.....▶ shortest path has length 0 : $\begin{matrix} \bullet & = & \bullet \\ x & & y \end{matrix}$

All Pairs 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 ((j in Int) | j < k and APSP[V,E](x,y,j))
```


▶ shortest path has length 0 : $\text{blue circle } x = \text{red circle } y$

▶ shortest path has length k : $\text{blue circle } x \quad \text{red circle } y$

All Pairs 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 ((j in Int) | j < k and APSP[V,E](x,y,j))
```

→ shortest path has length 0 : 

→ shortest path has length k : 

All Pairs 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 ((j in Int) | j < k and APSP[V,E](x,y,j))
```

→ shortest path has length 0 : $\text{blue circle } x = \text{red circle } y$

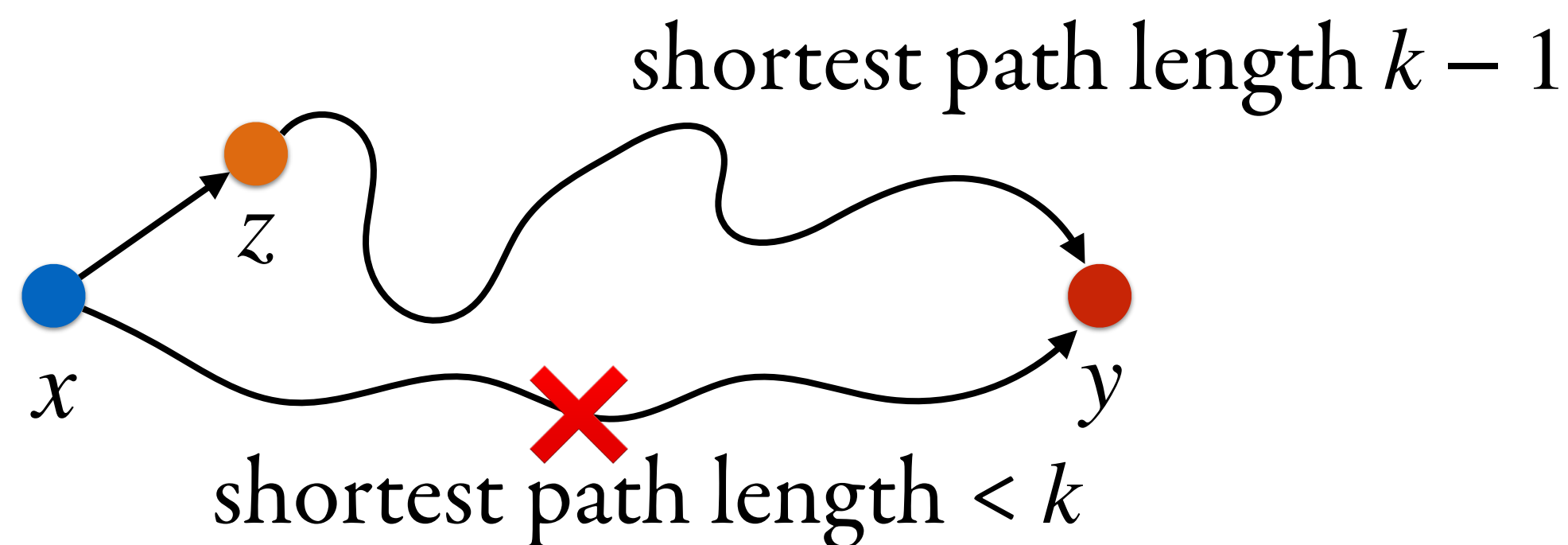
→ shortest path has length k : $\text{blue circle } x \xrightarrow{\text{orange circle } z} \text{red circle } y$ (shortest path length $k - 1$)

All Pairs 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 ((j in Int) | j < k and APSP[V,E](x,y,j))
```

shortest path has length 0 : $\text{blue circle} = \text{red circle}$
 $x = y$

shortest path has length k :



All Pairs 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 ((j in Int) | j < k and APSP[V,E](x,y,j))
```

This becomes more succinct with

- aggregates
- abstraction

All Pairs 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 ((j in Int) | j < k and APSP[V,E](x,y,j))
```

This becomes more succinct with

- aggregates
- abstraction

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

aggregate

abstraction

PageRank

Step 1: Matrix multiplication

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

PageRank

Step 1: Matrix multiplication

$$(A \cdot B)_{ij} = \sum_k A_{ik} \cdot B_{kj}$$

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


PageRank

Step 1: Matrix multiplication

$$(A \cdot B)_{ij} = \sum_k A_{ik} \cdot B_{kj}$$

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

Step 2: Prelims

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

PageRank

$$(A \cdot B)_{ij} = \sum_k A_{ik} \cdot B_{kj}$$

Step 1: Matrix multiplication

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

Step 2: Prelims

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

Step 3: PageRank

```
def PageRank[ {G}, 0 ] : vector[ numberOfRows[ G ] ]  
def PageRank[ {G}, k ] : MatrixMult[ G, PageRank[ G, k-1 ] ] where k > 0
```

PageRank

$$(A \cdot B)_{ij} = \sum_k A_{ik} \cdot B_{kj}$$

Step 1: Matrix multiplication

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

Step 2: Prelims

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

Step 3: PageRank

```
def PageRank[ {G}, 0 ] : vector[ numberOfRows[ G ] ]
def PageRank[ {G}, k ] : MatrixMult[ G, PageRank[ G, k-1 ] ] where k > 0
```

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

↪ 10 iterations of PageRank on matrix M

Rel is Handling Large Applications

Rel is Handling Large Applications

Rel in the Real World

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

Rel is Handling Large Applications

Rel in the Real World

- RelationalAI 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

Rel is Handling Large Applications

Rel in the Real World

- RelationalAI 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# \rightsquigarrow 15k lines of Rel
 - 205k lines of C++ \rightsquigarrow 9k lines of Rel

Rel is Handling Large Applications

Rel in the Real World

- RelationalAI 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# \rightsquigarrow 15k lines of Rel
 - 205k lines of C++ \rightsquigarrow 9k lines of Rel
 - Performance goes up
 - E.g. 1 month \rightsquigarrow a few hours of processing time

Rel is Handling Large Applications

Rel in the Real World

- RelationalAI 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# \rightsquigarrow 15k lines of Rel
 - 205k lines of C++ \rightsquigarrow 9k lines of Rel
 - Performance goes up
 - E.g. 1 month \rightsquigarrow a few hours of processing time

Application-wide optimization works!

Thanks!

Molham Aref, Paolo Guagliardo, George Kastrinis, Leonid Libkin, Victor Marsault,
Wim Martens, Mary McGrath, Filip Murlak, Nathaniel Nystrom, Liat Peterfreund,
Allison Rogers, Cristina Sirangelo, Domagoj Vrgoc, David Zhao, Abdul Zreika:

Rel: A Programming Language for Relational Data

To appear in SIGMOD 2025

ArXiv version:

