



# Building Intelligent Data Apps in Rel using Reasoning and Probabilistic Modelling

**Stefan Pabst**

Documentation & Training Team Lead

**Cassi Hunt**

Product Analysis & Benchmarking Team Lead

Declarative AI 2022



# RelationalAI



Founded in 2017, Cloud-native  
Recently out of stealth with first enterprise customers



Mission Driven team with track record of success



Active board of directors includes  
Bob Muglia, former CEO of Snowflake  
Soma Somasegar, Madrona



Team with broad expertise  
Databases, languages, and algorithms to machine learning and operations research

**\$122M in Funding**

Tier 1 firms in Silicon Valley,  
NYC and Seattle

---

**Berkeley HQ - Global team**

180+ people  
130+ computer scientists

---

**50+ PhDs**

2,000+ publications with  
100K+ citations and  
37 awards and counting

# What we are building at RelationalAI

A cloud-native relational database platform

designed for structured and semi-structured dynamic data

with data and application logic together in one place



# What we are building at RelationalAI

1. A cloud-native relational database platform
2. designed for structured and semi-structured dynamic data
3. with data and application logic together in one place

*This is our  
outline!*

*...Almost!*





# Outline

1. A cloud-native relational database platform
2. designed for structured and semi-structured dynamic data
3. with data and application logic together in one place
4. Example: Intelligent data application with Rel
5. The future of intelligent databases:
  - Probabilistic modeling, optimization, relational ML

*That's better.*



# What we are building at RelationalAI

1. A cloud-native relational database platform
2. designed for structured and semi-structured dynamic data
3. with data and application logic together in one place

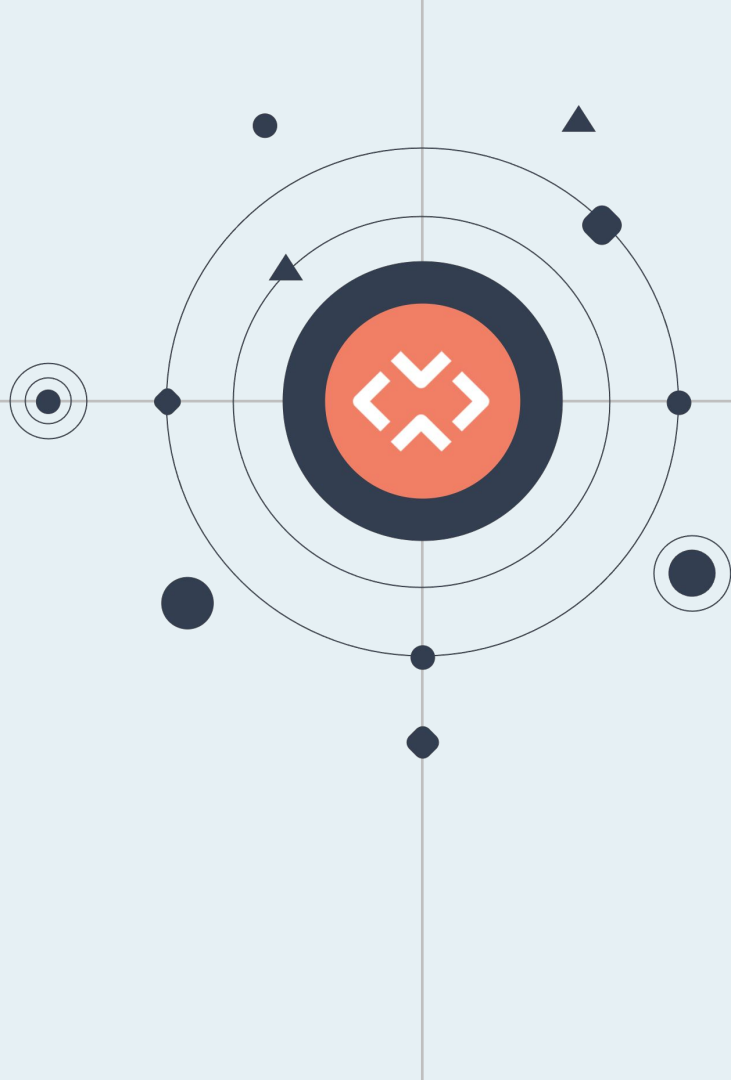
*First, what motivates us...*





# A Cloud-native Relational Database Platform

The RAI Knowledge Graph  
Management System (RKGMS)



# A Cloud-native Relational Database Platform

## Overview



- Motivation
- Architecture

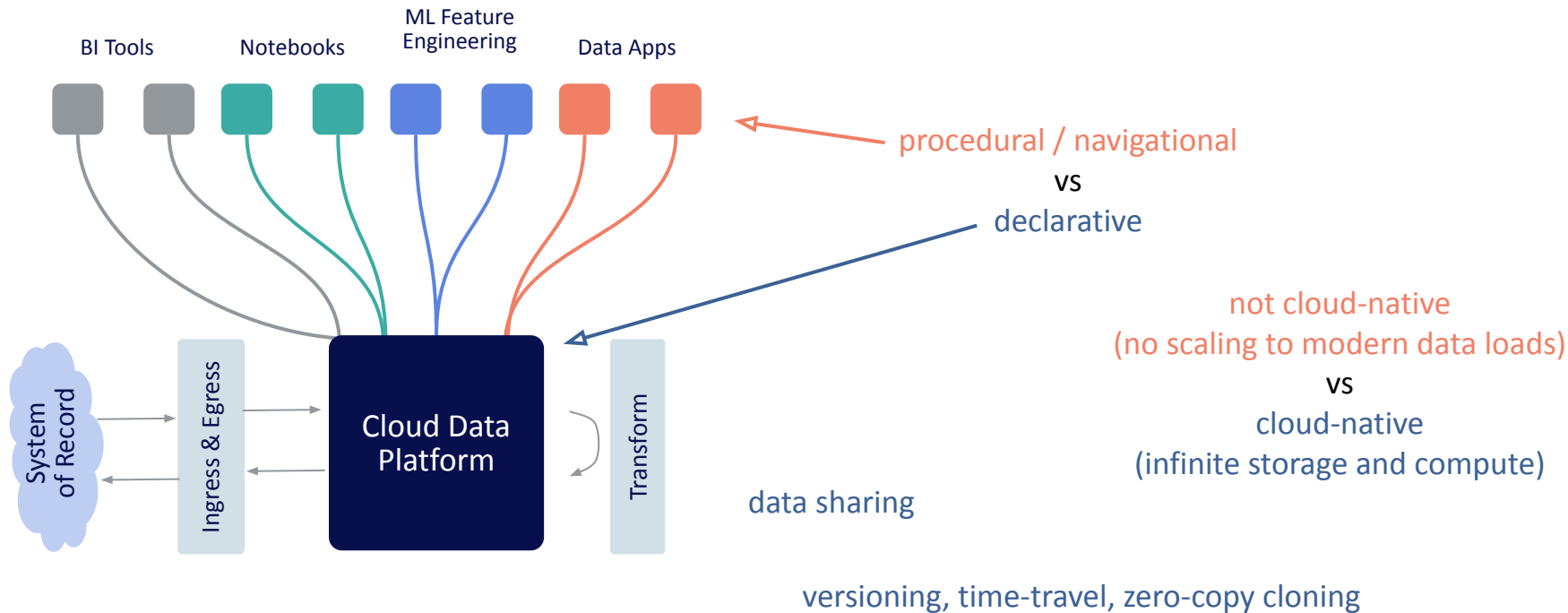
# A Cloud-native Relational Database Platform

## Overview

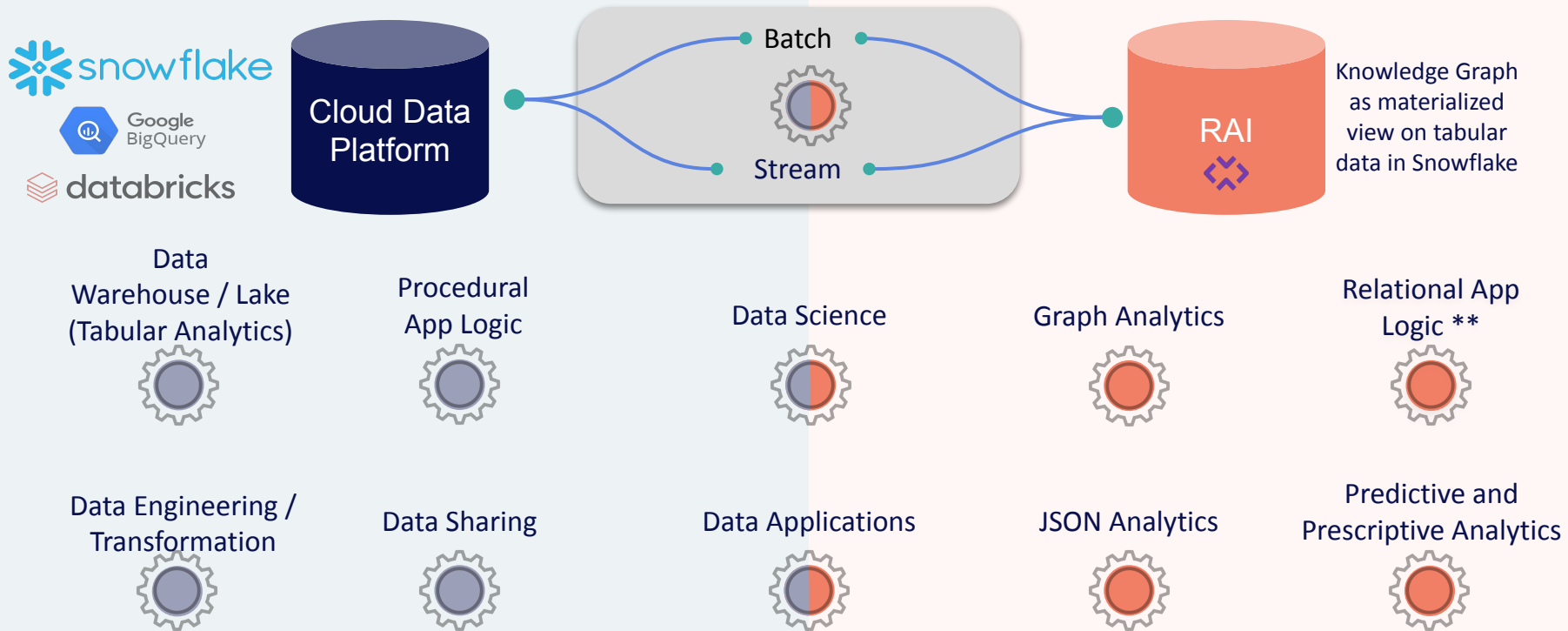


- Motivation
- Architecture

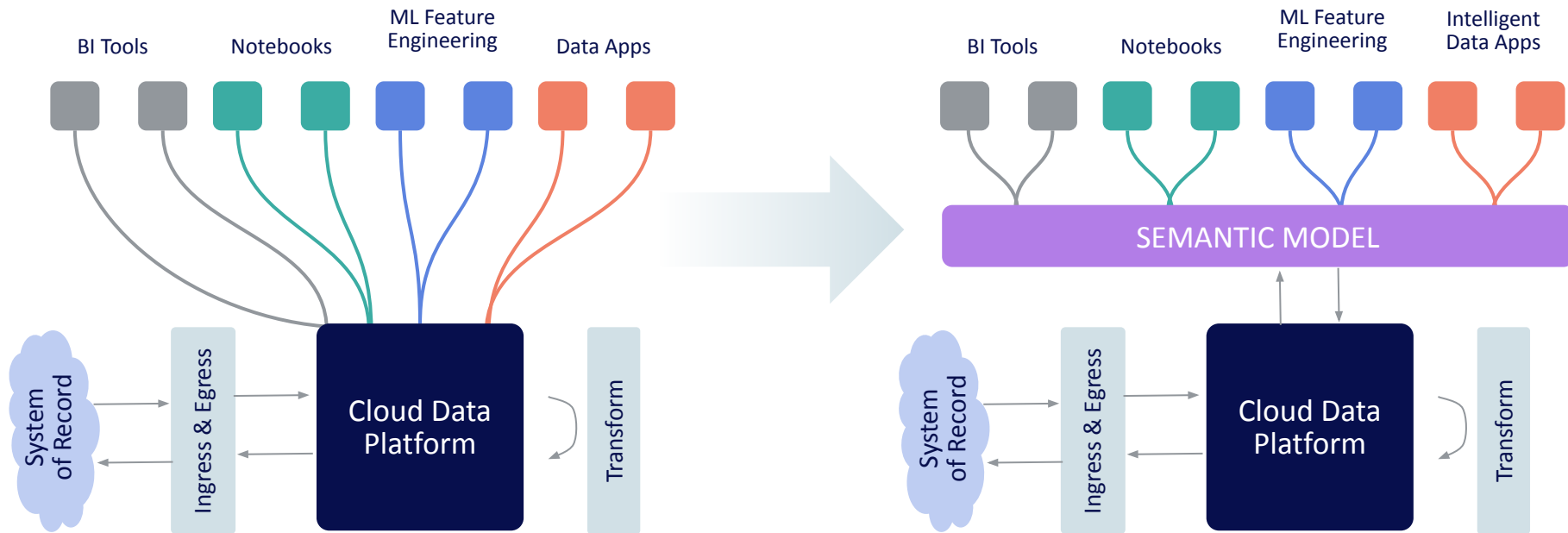
# The Modern Data Stack



# RAI Complements CDP for New Workloads

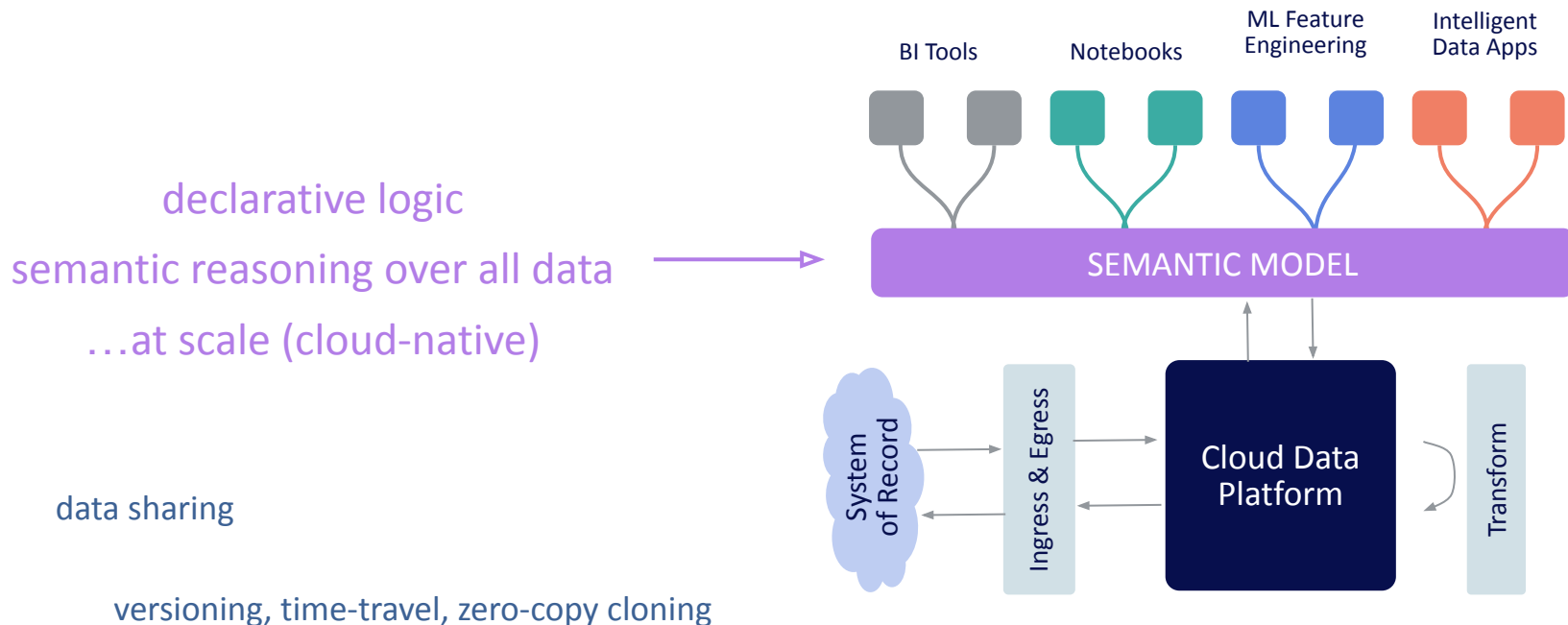


# Knowledge and Semantics in the Modern Data Stack





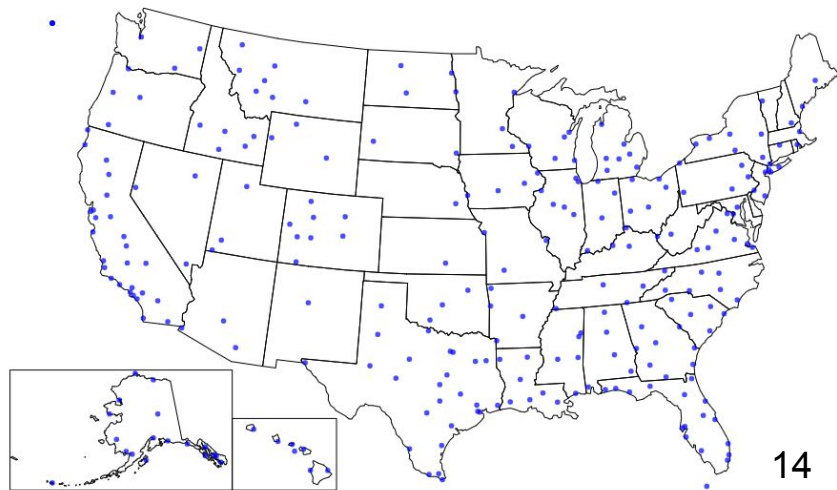
# Knowledge and Semantics in the Modern Data Stack



# Flight Data: How to build an intelligent data app

Opportunities in such an application

- Performance of carriers, airports, aircraft models and individual aircraft
- Discover relationships between seasons, time of day, weather and delays
- Planning and optimization
- Historical trends
- If live, act on issues that are arising



# Ex: flight data table

Period?  
Minutes?  
Hours?

Miles?  
Kilometers?

Are these  
exclusive?

Primary key?

Hours?  
Minutes?

Not a delay

Risk of  
messing up  
aggregates

Include helicopters?

Possible values?

carrier	origin	destination	flight_num	flight_time	tail_num	dep_time	arr_time	dep_delay	arr_delay	taxi_out	taxi_in	distance	cancelled	diverted	id2
F9	MCI	DEN	818	89	N916FR	2005-09-26 08:23:00 UTC	2005-09-26 09:07:00 UTC	-7	-8	9	6	533	N	N	36606824
WN	ELP		819	41	N708SW	2000-08-18 07:30:00 UTC	2000-08-18 07:20:00 UTC	0	-5	7	2	295	N	N	4369021
			819	5		-11-16 07:15:00 UTC	2001-11-16 07:29:00 UTC	-5	-9	19	3	332	N	N	11838308
			819	4		-12-04 22:12:00 UTC	2001-12-04 23:53:00 UTC	7	4	49	4	291	N	N	12060416
WN	PHX	SAN	819	51	N391SW	2001-12-05 09:11:00 UTC	2001-12-05 09:17:00 UTC	11				304	N	N	12383068
WN	LAS	AUS	819	135	N519SW	2002-04-05 08:18:00 UTC	2002-04-05 12:47:00 UTC	8				1085	N	N	13763279
WN	SJC	LAS	819	63	N528SW	2002-07-14 06:30:00 UTC	2002-07-14 07:47:00 UTC	0	-3	11	3	386	N	N	15284777
WN	LAS	AUS	819	137	N502SW	2002-09-16 08:20:00 UTC	2002-09-16 12:52:00 UTC	0	-8	11	4	1085	N	N	16027516
DL	MSP	SLC	819	149	N3754A	2002-09-29 19:34:00 UTC	2002-09-29 21:35:00 UTC	9	15	25	7	991	N	N	16399211
DL	MSP	SLC	819	145	N3745B	2002-12-06 19:27:00 UTC	2002-12-06 21:16:00 UTC	-3	-9	18	6	991	N	N	17417961
WN	SJC	LAS	819	54	N730SW	2003-06-26 06:30:00 UTC	2003-06-26 07:44:00 UTC	0	-11	15	5	386	N	N	20460576
WN	SJC	LAS	819	60	N501SW	2003-09-15 06:30:00 UTC	2003-09-15 07:50:00 UTC	0	0	18	2	386	N	N	22113592
NW	ATL	MEM	819	51	N785NC	2003-11-20 07:11:00 UTC	2003-11-20 07:15:00 UTC	-9	-23	10	3	332	N	N	23383534
DL	MSP	ATL	819	120	N906DE	2004-02-10 09:59:00 UTC	2004-02-10 13:36:00 UTC	-6	0	30	7	906	N	N	25206983
US	CHS	CLT	820	38	N592US	2002-07-01 19:32:00 UTC	2002-07-01 20:54:00 UTC	37	64	9	35	168	N	N	15142411
FL	TPA	ATL	820	64	N955AT	2003-01-31 11:29:00 UTC	2003-01-31 12:55:00 UTC	-6	-5	13	9	406	N	N	17949329
WN	RDU	PHL	820	73	N382SW	2004-12-02 11:10:00 UTC	2004-12-02 12:31:00 UTC	0	-19	5	3	336	N	N	30796766
HP	PHX	BOS	820	0	N826AW	2005-10-25 00:00:00 UTC	2005-10-25 00:00:00 UTC	0	0	0	0	2300	Y	N	37174931
US	PBI	CLT	821	90	N624AU	2002-05-18 06:35:00 UTC	2002-05-18 07:00:00 UTC	-5	-8	10	6	590	N	N	14247814
US	PBI	CLT	821	87	N624AU	2002-05-18 06:50:00 UTC	2002-05-18 07:43:00 UTC	-7	-7	16	7	590	N	N	20289351
DL	GSO	CVG	821	70	N943DL	2002-05-18 08:12:00 UTC	2002-05-18 08:12:00 UTC	1	11	14	7	330	N	N	21396171

# Reasoning manages app logic with the data

**Reasoning** subsumes business logic now implemented procedurally in languages like Java, C#, Python, Scala, PL/SQL, T/SQL etc.

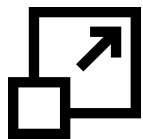
Fixing the **“split brain”** problem where the data is managed in one layer and knowledge/semantics in another will have huge impact.

Bringing the app logic to the data makes it possible for one (cloud native) system to manage the semantics, integrity, and resources needed for the application.



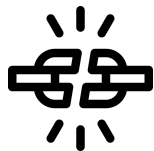
With thanks to Peter Bailis...

# Core requirements



## Scalability

Separate data storage from compute



## Data independence

Separate application logic from data representation



## Semantic reasoning

Data and application logic together in database



## Live Programming

Incremental for changes to data and logic



## Key innovations

- ## 1. Immutability - Cloud native architecture

*Data + metadata is versioned and immutable*

- ## 2. Expressive relational language (Rel)

*Designed for complex business logic, abstraction, schema abstraction, libraries*

- ### 3. Join algorithms

*Worst-case optimal join algorithms specifically suitable for knowledge graphs*

- ## 4. Semantic optimization

*Use knowledge to apply deep and structural optimization*

- ## 5. Vectorized and just-in-time (JIT) compilation of WCOJ

Compile complex joins to native code. Vectorized engine for simpler joins

- ## 6. Incremental computation

Maintain computations incrementally  
Compile model incrementally



# Learn More

From the  
Modern Data Stack  
to Knowledge Graphs

Bob Muglia  
Board Member, RelationalAI



**"KGC Bob Muglia"** for modern data stack and relational knowledge graph

[Youtube](#)

Carnegie Mellon University



**VACCINATION**  
Database Talks  
★ **BOOSTER** ★

**09** Martin Bravenboer  
RelationalAI

**"CMU RelationalAI"** for RAI system overview

[Youtube](#)

**Algorithms for  
Relational Knowledge Graphs**

The Dutch Seminar on Data Systems Design  
May 13, 2022

Martin Bravenboer  
VP Engineering



**"DSDSD Bravenboer"** for different RAI system overview

[Youtube](#)



<https://twitter.com/RelationalAI>

# A Cloud-native Relational Database Platform

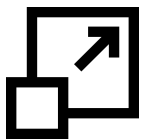
## Overview



- Motivation
- Architecture



# Core requirements



## Scalability

Separate data storage from compute



## Data independence

Separate application logic from data representation



## Semantic reasoning

Data and application logic together in database

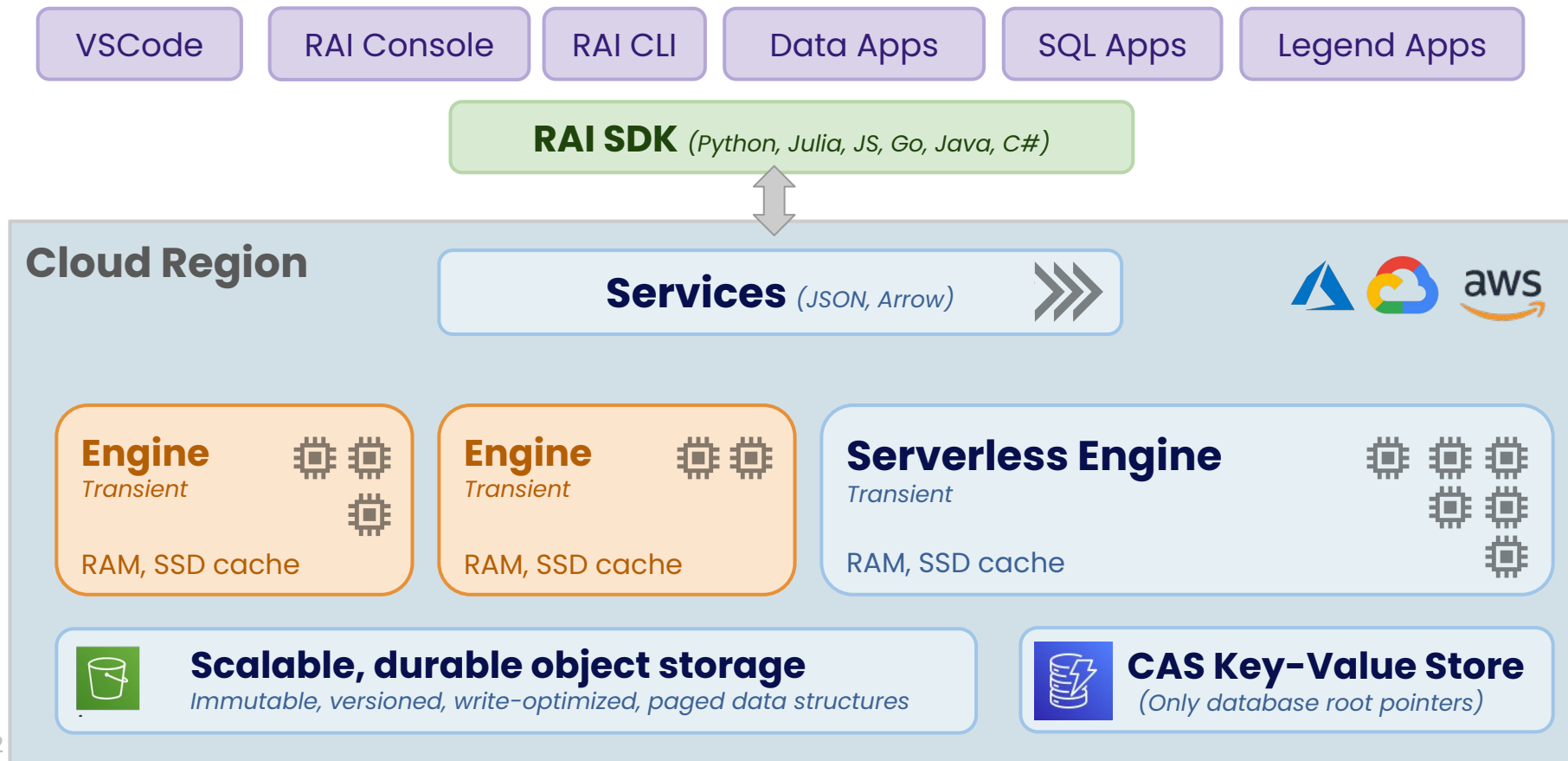


## Live Programming

Incremental for changes to data and logic



# Cloud Native Deployment Architecture



## Key innovations

## 1. Immutability - Cloud native architecture

*Data + metadata is versioned and immutable*

## 2. Expressive relational language (Rel)

*Designed for complex business logic, abstraction, schema abstraction, libraries*

### 3. Join algorithms

*Worst-case optimal join algorithms specifically suitable for knowledge graphs*

## 4. Semantic optimization

*Use knowledge to apply deep and structural optimization*

## 5. Vectorized and just-in-time (JIT) compilation of WCOJ

Compile complex joins to native code. Vectorized engine for simpler joins

## 6. Incremental computation

Maintain computations incrementally  
Compile model incrementally



# Storage Management: Influences and Resources

## Elastic Storage Management

- **The Snowflake Elastic Data Warehouse**  
Dageville et al., SIGMOD 2016
- **Building an Elastic Query Engine on Disaggregated Storage**  
Vuppalapati et al., NSDI 2020

## Write Optimization


- **Lower Bounds for External Memory Dictionaries**  
Brodal et al., SODA 2003
- **An Introduction to B $\epsilon$ -trees and Write-Optimization**  
Bender et al., :login: magazine, 2015
- **Design and Implementation of the LogicBlox System**  
Aref et al. SIGMOD 2015

## In-Memory Performance

- **LeanStore: In-Memory Data Management Beyond Main Memory**  
Leis et al., ICDE 2018
- **Umbra: A Disk-Based System with In-Memory Performance**  
Neumann et al., CIDR 2020



# Coexist as One Happy Relational Family

 Supported Languages &  
Data Formats

CSV  
RDF  
SQL RDBMS CDC  
Binary objects  
Tensor data  
Parquet, Iceberg  
LPG  
JSON



Rel



SQL



Legend

GQL

SPARQL

GraphQL



Core Rel IR



Relational Knowledge Graph System



LLVM

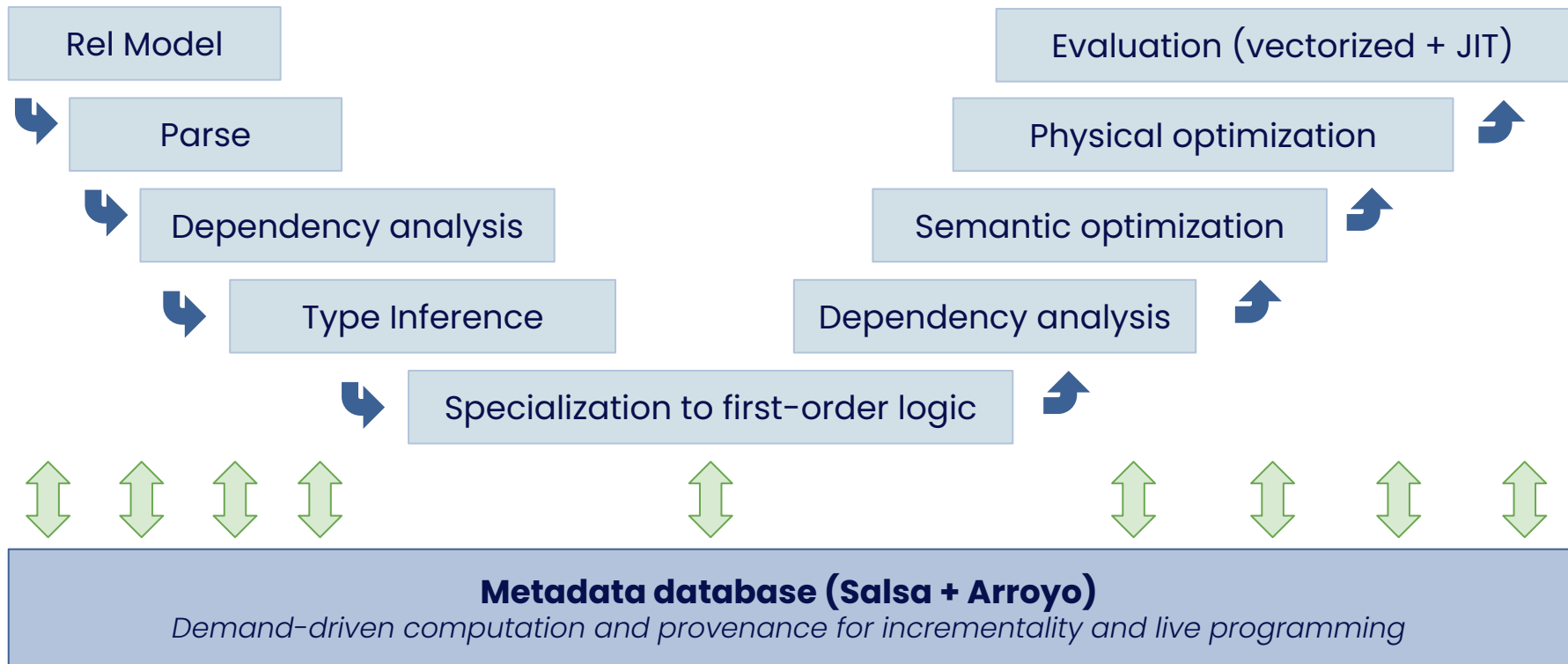


Specialized  
Solvers



Future

# Internal Engine Architecture

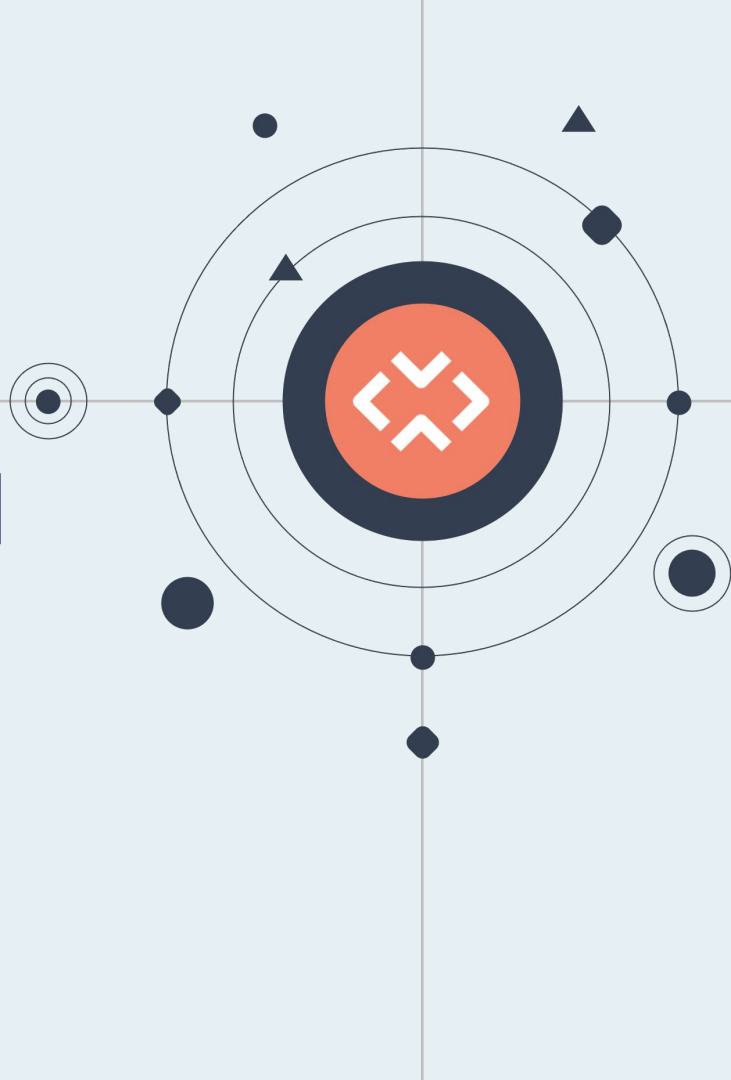




A Cloud-native Relational Database Platform

# Designed for structured and semi-structured dynamic data

The RAI Knowledge Graph  
Management System (RKGMS)



# Designed for structured and semi-structured dynamic data

## Overview

- The relational model
- Views on relations
- Performance





# Designed for structured and semi-structured dynamic data

## Overview



- **The relational model**
- Views on relations
- Performance

The **relational model** represents all data using first-order relations

The purpose is independence of application logic  
from changes in data representation:

**data independence**

**SQL is not the relational model**

*(inadequacy of SQL is often used as motivation for new data models)*

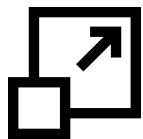
core requirement

# Data Independence

Separate application logic from data  
representation

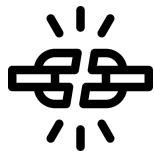


# Core requirements



## Scalability

Separate data storage from compute



## Data independence

Separate application logic from data representation



## Semantic reasoning

Data and application logic together in database



## Live Programming

Incremental for changes to data and logic



# Data independence with RAI relations

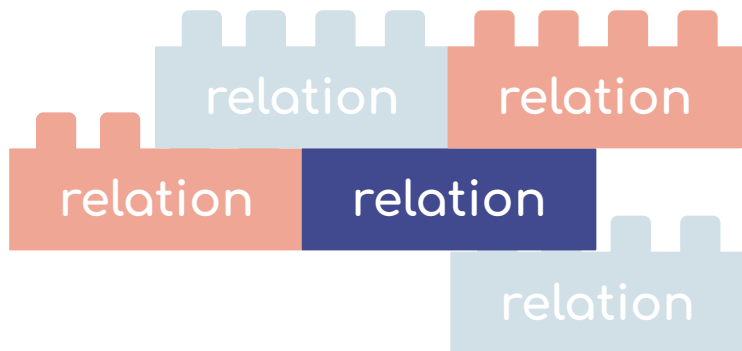
- All data and logic stored as RAI *relations*
- Application logic (*also*) written as declarative relation definitions
- Tables, graphs, etc are composed *views* over relations

*What is a RAI relation?*



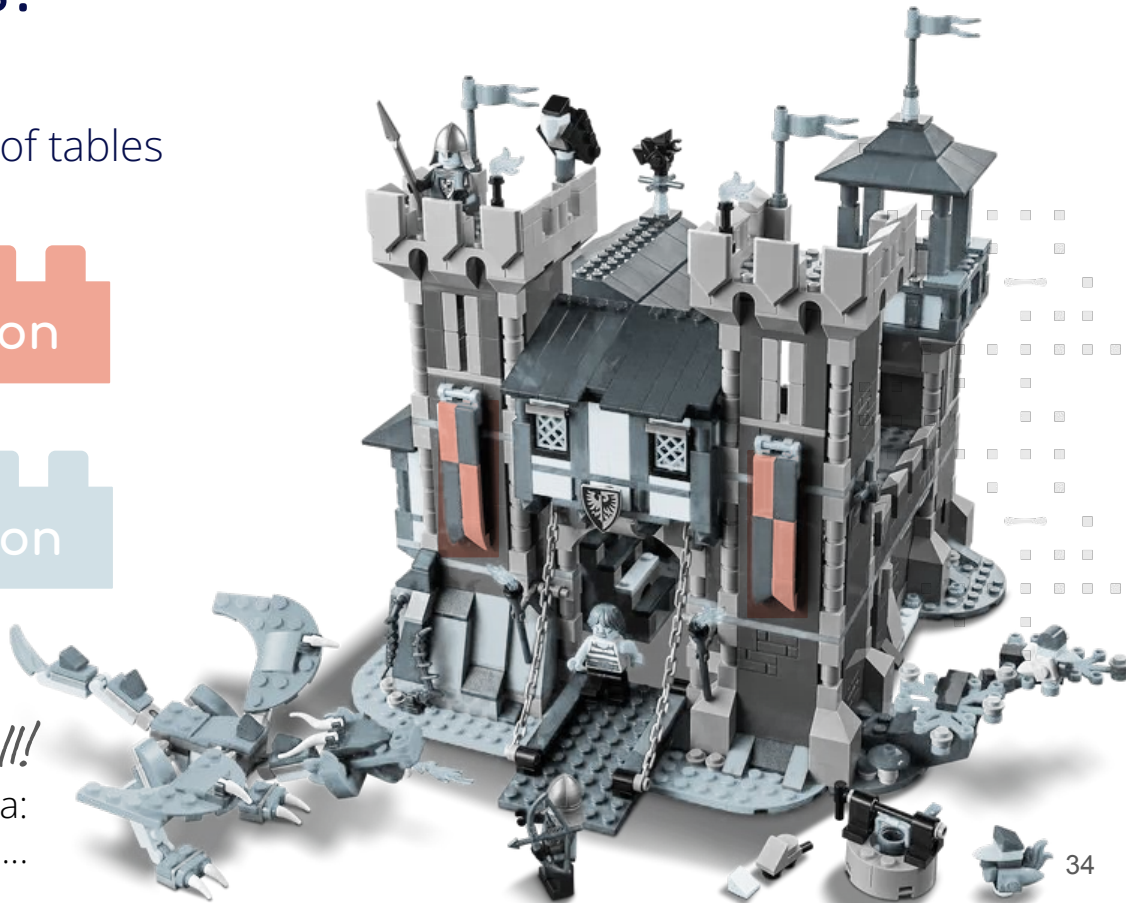
# Relations aren't tables!

Relations are the **building blocks** of tables



*But that's not all!*

Relations enable many other views on data:  
graph, json, RDF, ...



# Wide vs Narrow Data

*Relations are narrow*

Ex: Flight data table:

#	Carrier	Flight Number	Origin	Destination	Departure
1	AL	712	SEA	STL	2022-09-16
2	DL	2224	ATL	SEA	2022-09-16
3	DL	751	ATL		2022-09-16
4	DL	2224	ATL	SEA	2022-09-18

*Wide!*

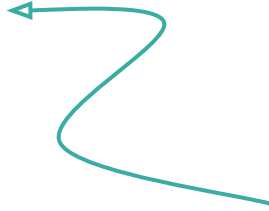
Departure	
1	2022-09-16
2	2022-09-16
3	2022-09-16
4	2022-09-18

*Narrow!*



# Wide vs Narrow Data

Carrier	
1	AL
2	DL
3	DL
4	DL



Flight Number	
1	712
2	2224
3	751
4	2224



Origin	
1	SEA
2	ATL
3	ATL
4	ATL



Destination	
1	STL
2	SEA
4	SEA



#	Carrier	Flight Number	Origin	Destination	Departure
1	AL	712	SEA	STL	2022-09-16
2	DL	2224	ATL	SEA	2022-09-16
3	DL	751	ATL		2022-09-16
4	DL	2224	ATL	SEA	2022-09-18

Departure	
1	2022-09-16
2	2022-09-16
3	2022-09-16
4	2022-09-18

*(This is called 6th Normal Form or 6NF)*





# Narrow Data Advantages

Carrier	
1	AL
2	DL
3	DL
4	DL

Total Delay	
1	10
2	0
3	37
4	14

Origin	
1	SEA
2	ATL
3	ATL
4	ATL

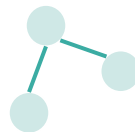
Destination	
1	STL
2	SEA
4	SEA

Departure	
1	2022-09-16
2	2022-09-16
3	2022-09-16
4	2022-09-18

*Relations are ...*

- Extensible + dynamic
  - Add additional base data or derived logic on data
- Free from nulls
  - The data is there or it isn't!
- Composable
  - Create tables, subgraphs, and other views on data

Flight Number	
1	712
2	2224
3	751
4	2224






# Designed for structured and semi-structured dynamic data

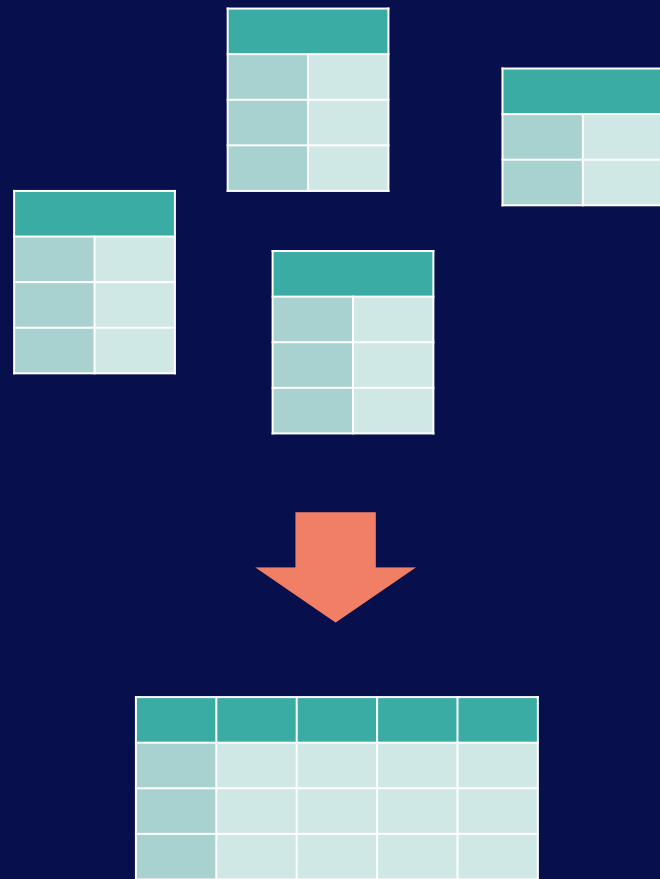
## Overview



- The relational model
- **Views on relations**
- Performance

# Table

A view composed of a module of relations with the same keys



# Module: relation of relations

A module groups relations together as a single relation.

Ex: The relations related to flights can be grouped into a table view by creating the module `flight`.

flight:carrier		flight:origin		flight:destination	
1	AL	1	SEA	1	STL
2	DL	2	ATL	2	SEA
3	DL	3	ATL	4	SEA
4	DL	4	ATL		



flight					
carrier		origin		destination	
1	AL	1	SEA	1	STL
2	DL	2	ATL	2	SEA
3	DL	3	ATL	4	SEA
4	DL	4	ATL		



# Module: relation of relations

The individual relations are accessed by name just the same as using keys.

*Under the hood:* the keys are *specialized* into separate relations, distinguishing schema from data.

carrier	
1	AL

carrier(1, "AL")

flight:carrier	
1	AL

flight(:carrier, 1, "AL")  
flight:carrier(1, "AL")

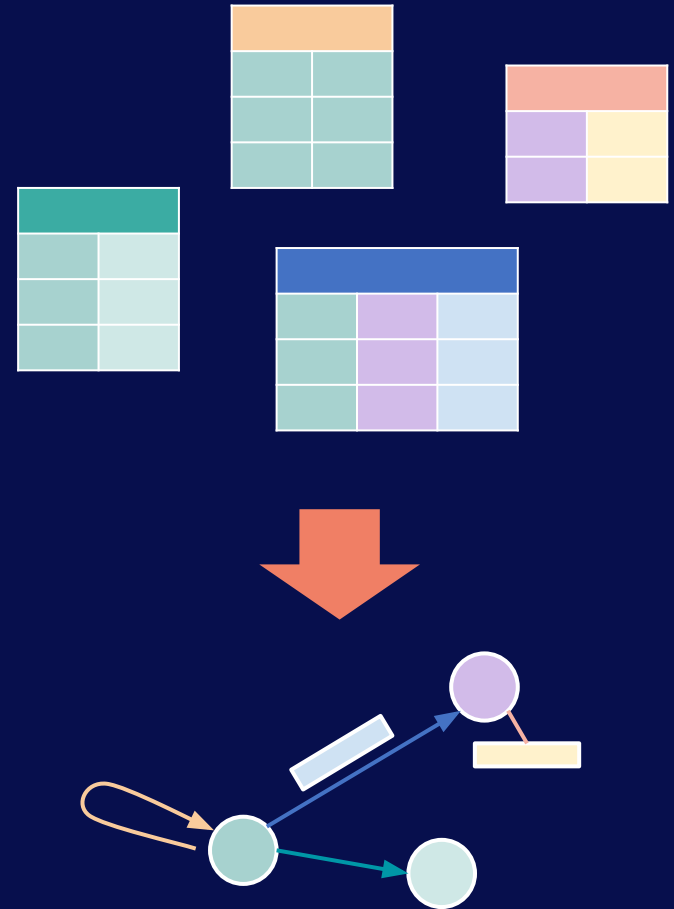
## Table view

flight(:carrier, 1, "AL")  
↕  
table(column, row, value)



# Graph

A view composed of a set of GNF relations



# Relational Graph

Carrier	
1	AL
2	DL
3	DL
4	DL

Flight Number	
1	712
2	2224
3	751
4	2224

Origin	
1	SEA
2	ATL
3	ATL
4	ATL

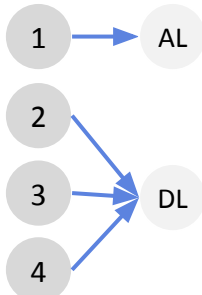
Destination	
1	STL
2	SEA
4	SEA

Departure	
1	2022-09-16
2	2022-09-16
3	2022-09-16
4	2022-09-18

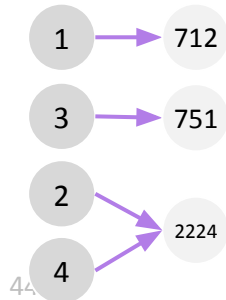


# Relational Graph

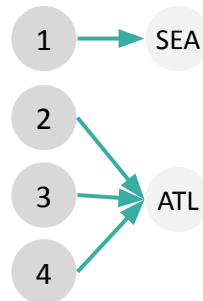
## Carrier



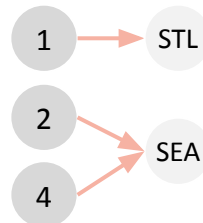
## Flight Number



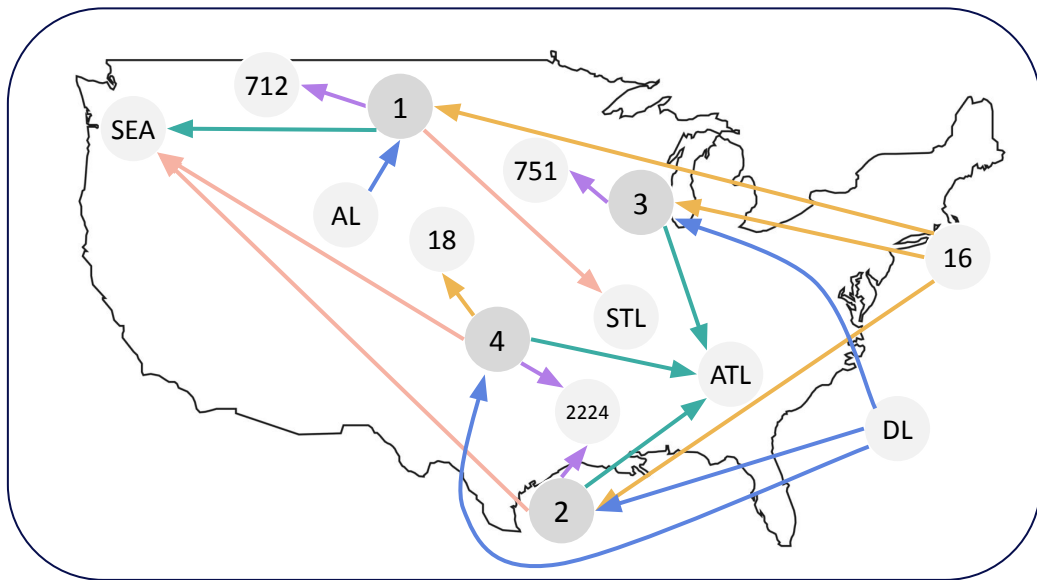
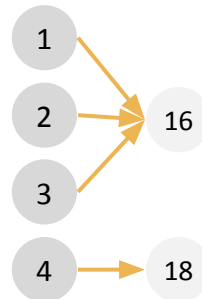
## Origin



## Destination

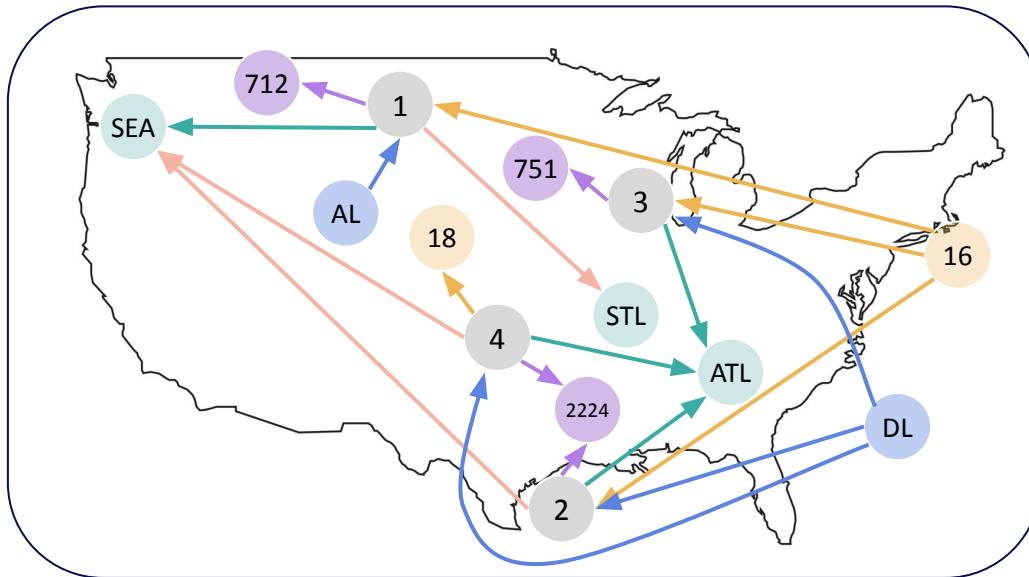
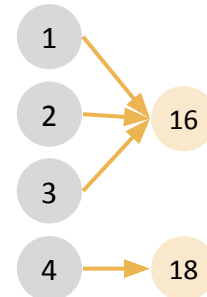
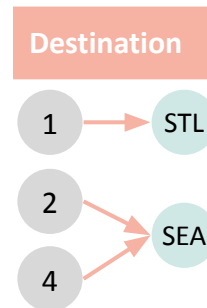


## Departure





## 45

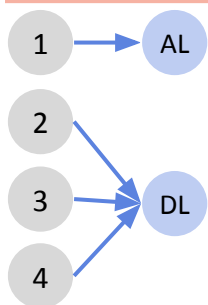


#

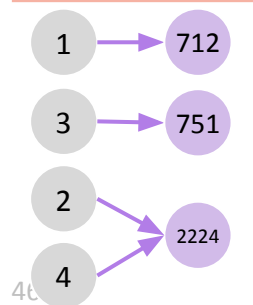
# Relational Graph

Graph Normal Form =  
6NF + "Things not Strings"

## Carrier



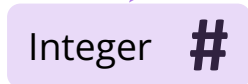
## Flight Number



## Carrier

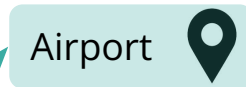


## Integer

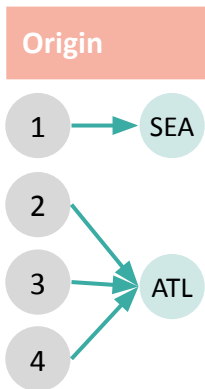
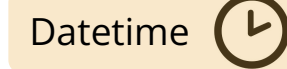


## Flight

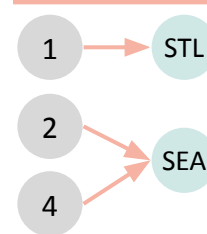
## Airport



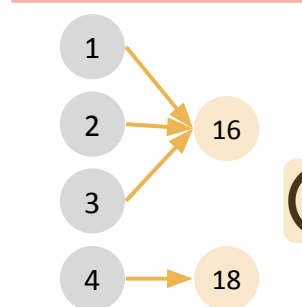
## Datetime



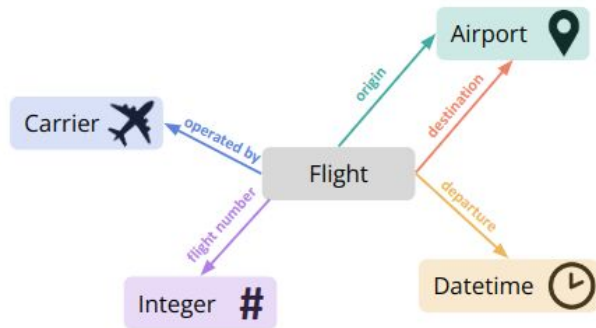
## Destination



## Departure



# Things Not Strings



**Thing** = annotated data that adds semantic context

- Value types, Ex:



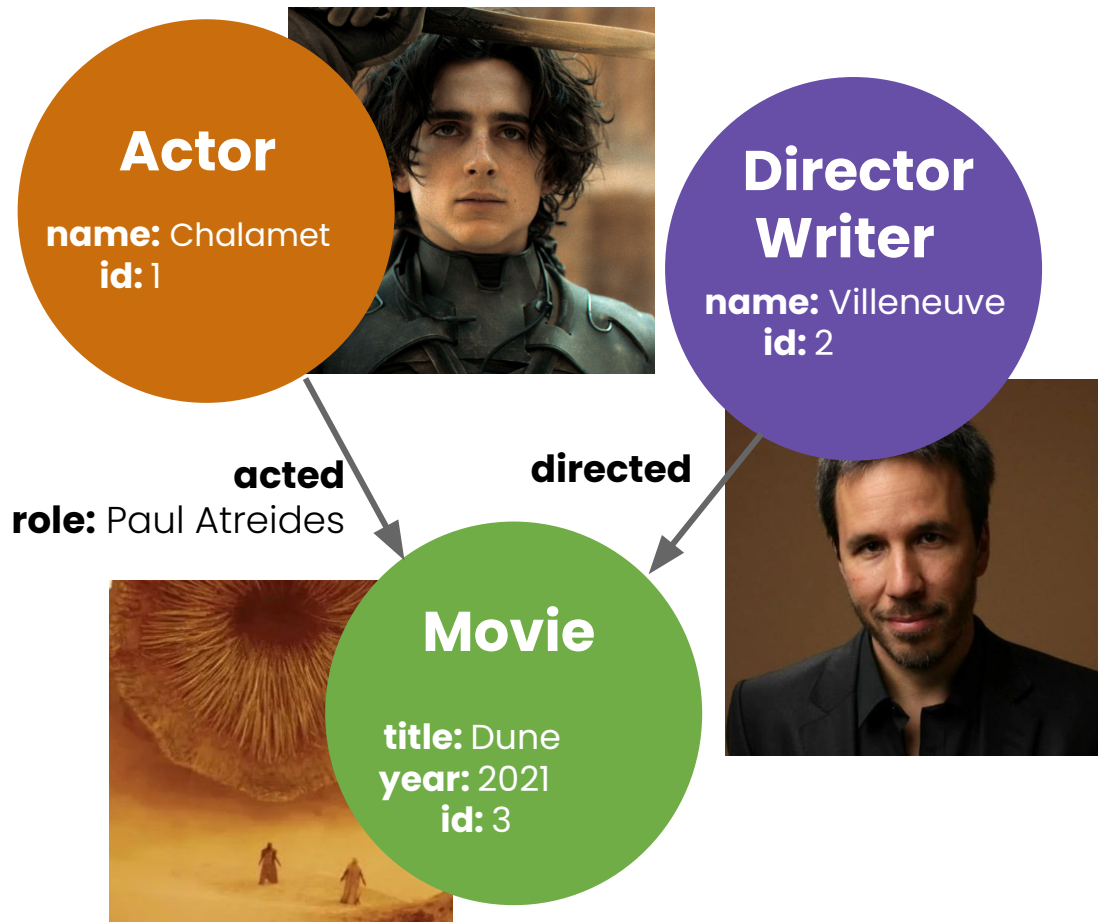
- Entity types, Ex:





Relational modeling is a natural fit  
for graphs

# Labelled Property Graphs as Relational Graphs

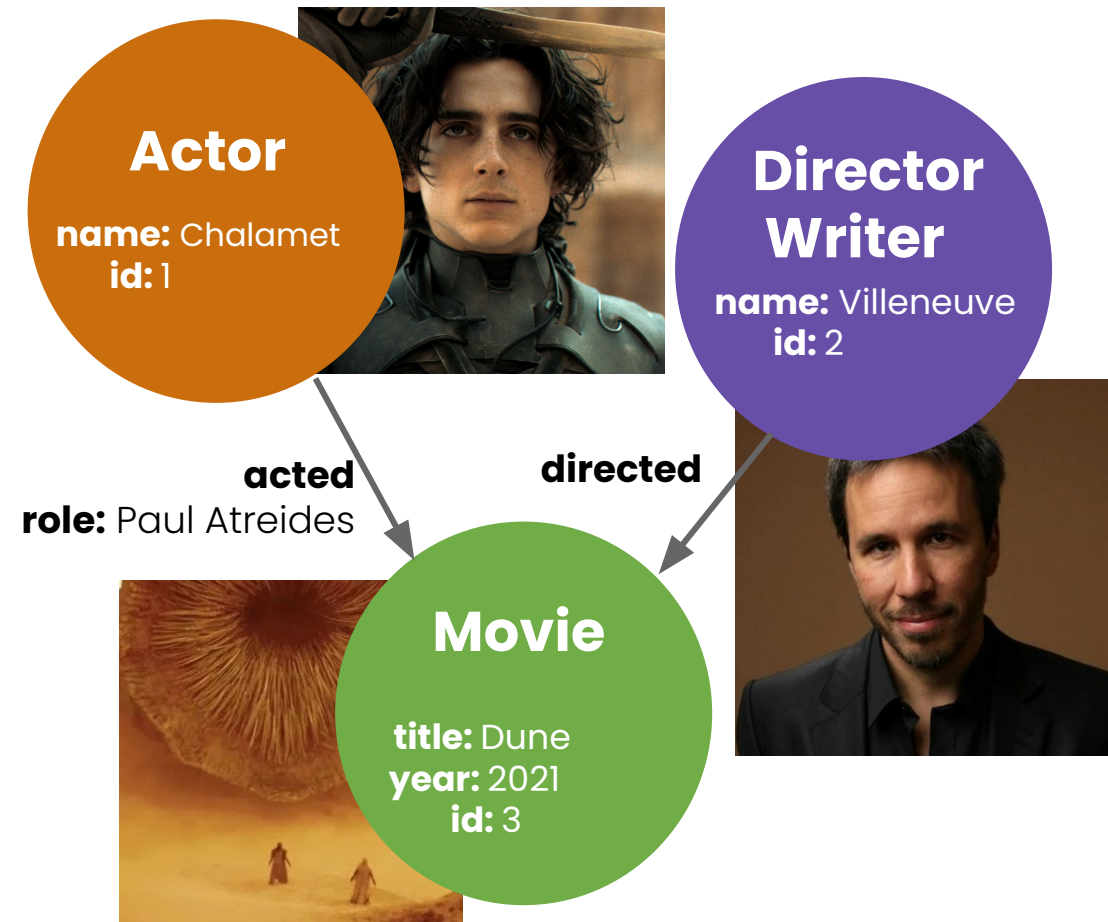


```
movie(3)
title(3, "Dune")
year(3, 2021)
```

```
director(2)
writer(2)
name(2, "Villeneuve")
```

```
actor(1)
name(1, "Chalamet")
```

# Labelled Property Graphs as Relational Graphs



```
movie(3)
title(3, "Dune")
year(3, 2021)
```

```
director(2)
writer(2)
name(2, "Villeneuve")
```

```
actor(1)
name(1, "Chalamet")
```

```
directed(2, 3)
```

```
acted(1, 3)
role(1, 3, "Paul Atreides")
```

# Json

A view composed of modularized relations.



```
{  
  "name": "John",  
  "age": 30,  
  "address": { "street": "123 Main St",  
                "city": "New York" },  
  "hobbies": [  
    { "hobby": "Reading",  
      "duration": 100 },  
    { "hobby": "Golfing",  
      "duration": 50 }  
  ]  
}
```



name	age	address	
John	30	street	city
John	30	123 Main St	New York
hobbies	hobby	duration	
	Reading	100	
hobbies	Golfing	50	
hobbies			

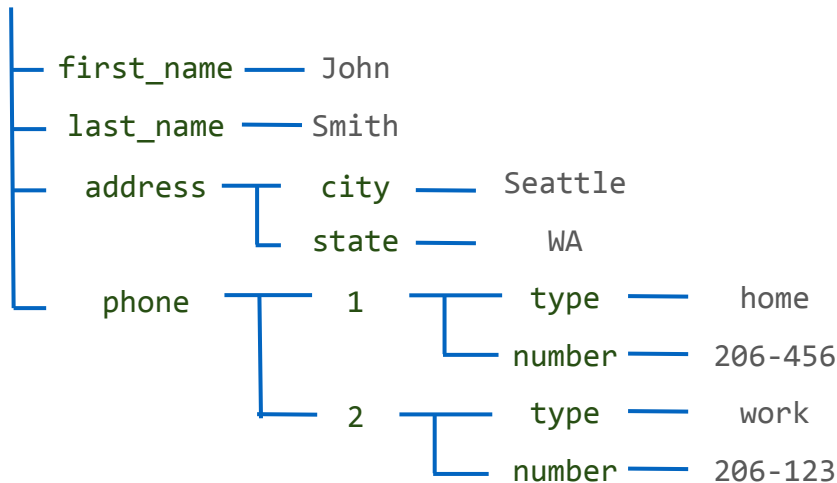
# Relational Representation of JSON

RAI represents JSON with first-order relations in graph normal form.

After parsing, JSON is typically represented as a tree (right)

Ex: `contact.json`

```
{  
  "first_name": "John",  
  "last_name": "Smith",  
  "address": { "city": "Seattle",  
               "state": "WA" },  
  "phone": [  
    { "type": "home",  
      "number": "206-456" },  
    { "type": "work",  
      "number": "206-123" }  
  ]  
}
```





# Relational Representation of JSON

First let's make the tree look more like a relation.

The **green** cells are the internal nodes of the tree.

The **grey** cells are leaves (actual data).

```
{
  "first_name": "John",
  "last_name": "Smith",
  "address": { "city": "Seattle",
               "state": "WA" },
  "phone": [
    { "type": "home",
      "number": "206-456" },
    { "type": "work",
      "number": "206-123" }
  ]
}
```

contact			
first_name	John		
last_name	Smith		
address	city	Seattle	
	state	WA	
phone	1	type	home
		number	206-456
	2	type	work
		number	206-123

# Relational Representation of JSON

The **green** prefixes are keys to the data.

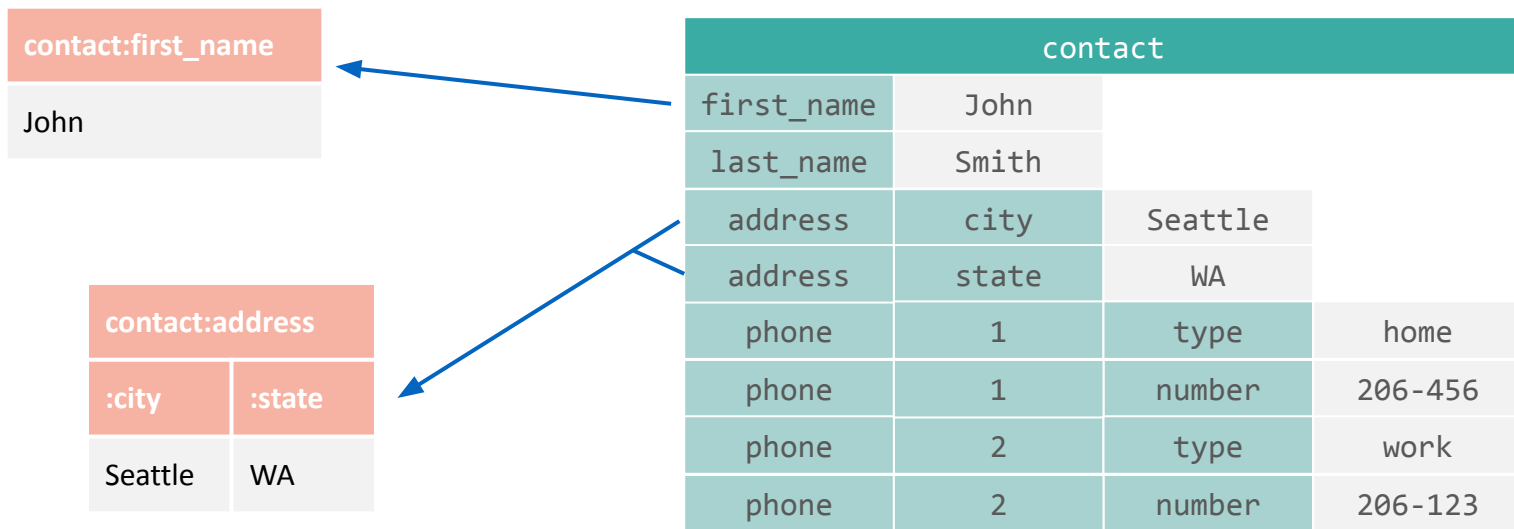
The relation (module) **contact** contains a *set of tuples*, each with a **key** and single **value**.

```
{
  "first_name": "John",
  "last_name": "Smith",
  "address": { "city": "Seattle",
               "state": "WA" },
  "phone": [
    { "type": "home",
      "number": "206-456" },
    { "type": "work",
      "number": "206-123" }
  ]
}
```

contact			
first_name	John		
last_name	Smith		
address	city	Seattle	
address	state	WA	
phone	1	type	home
phone	1	number	206-456
phone	2	type	work
phone	2	number	206-123

# Relational Representation of JSON

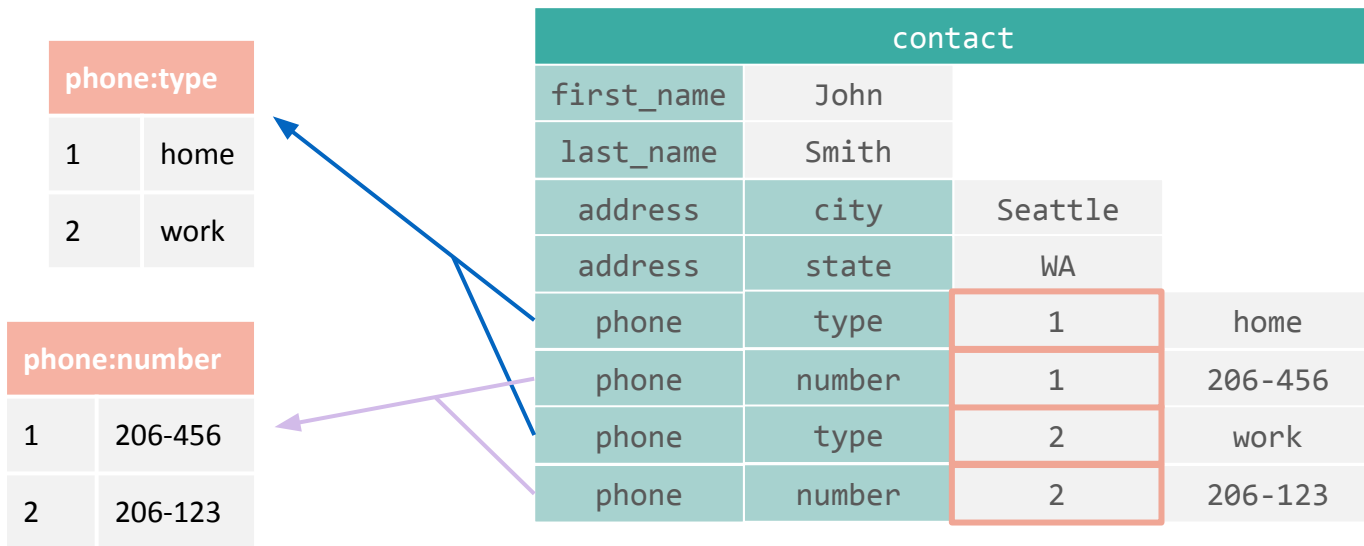
*Under the hood:* As with tables, the **green** keys are *specialized* into separate relations, distinguishing schema from data.



# Relational Representation of JSON

*Under the hood:* The data is organized by path abstraction.

The reordering here reflects the internal (*physical*) representation, but not how relations are queried (*logical* representation).



*Q: If narrow relations are so great...*

*... why aren't they  
standard already?*

*A: Performance!*

*Joins are hard :(*



# Designed for structured and semi-structured dynamic data

## Overview

- The relational model
- Views on relations
- **Performance**



## Key innovations

- ## 1. Immutability - Cloud native architecture

*Data + metadata is versioned and immutable*

- ## 2. Expressive relational language (Rel)

*Designed for complex business logic, abstraction, schema abstraction, libraries*

- ### 3. Join algorithms

*Worst-case optimal join algorithms specifically suitable for knowledge graphs*

- ## 4. Semantic optimization

*Use knowledge to apply deep and structural optimization*

- ## 5. Vectorized and just-in-time (JIT) compilation of WCOJ

Compile complex joins to native code. Vectorized engine for simpler joins

- ## 6. Incremental computation

Maintain computations incrementally  
Compile model incrementally



# Knowledge Graphs need different join algorithms

Join algorithms used in SQL-based relational databases are **binary join algorithms**.  
For knowledge graphs intermediate results are too large.

Example: *Find a child who acted in a movie directed by their parent.*

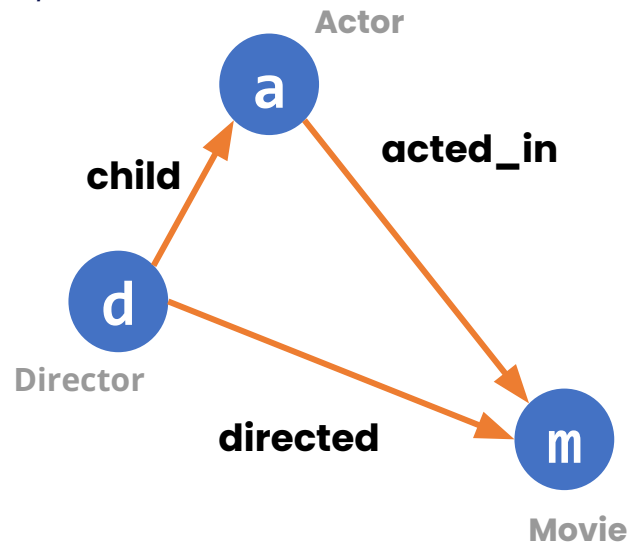
`directed(d, m) and child(d, a) and acted_in(a, m)`

Binary join options:

`directed(d, m) and child(d, a)`  
not selective: most directors have children!

`directed(d, m) and acted_in(a, m)`  
not selective: every movie has a director and actors!

`child(d, a) and acted_in(a, m)`  
not selective: every actor has parents!



**Triangle Graph Pattern**

This is one reason for the stigma 'joins are bad'



# Three ways of looking at WCOJ

We use **worst-case optimal join algorithms**. This is a new class of algorithms whose properties and trade-offs are not yet well understood.

Leapfrog Triejoin (LFTJ), GenericJoin, and Dovetail Join are WCOJ algorithms.

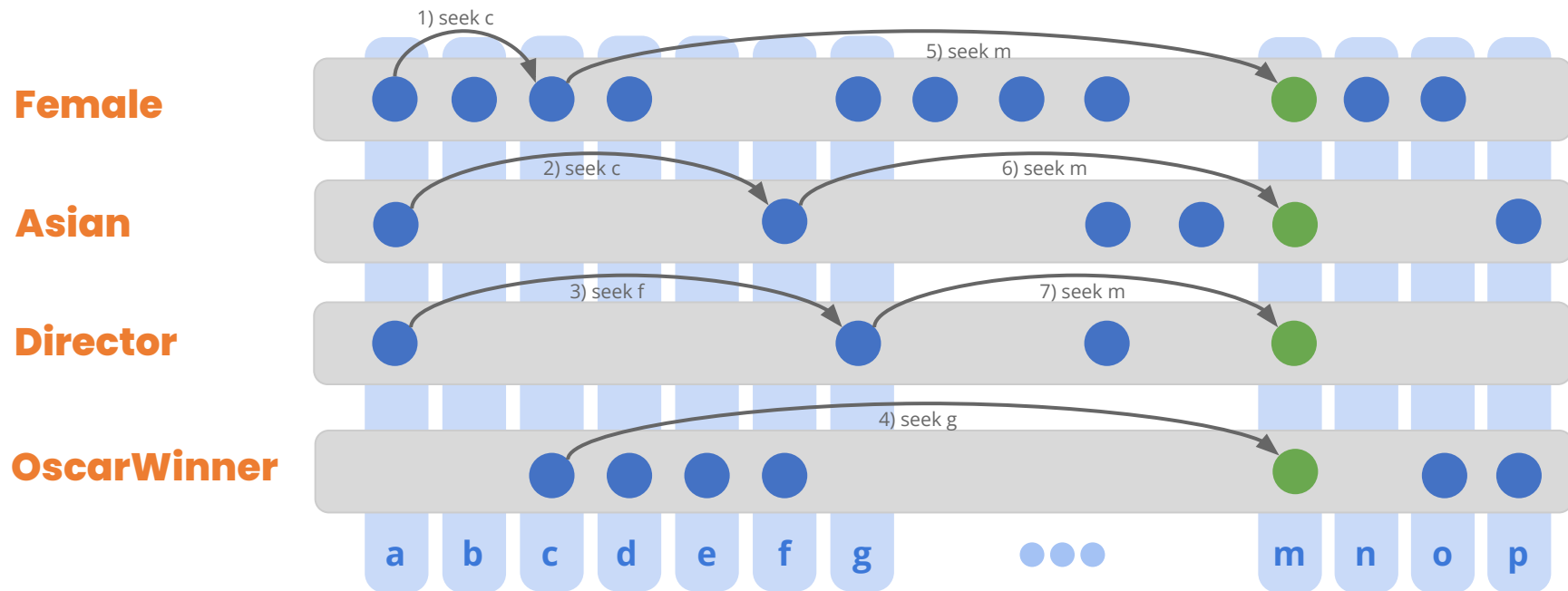
We look at the properties from three angles:

⇒ **Exploit sparsity in data**

⇒ **Recast the subquery problem and embrace correlation**

⇒ **Recast index selection problem**

# WCOJ uses sparsity of all relations to narrow down search



**Worst-case optimal join (WCOJ)** algorithms use the sparsity of all relations to narrow down the search.



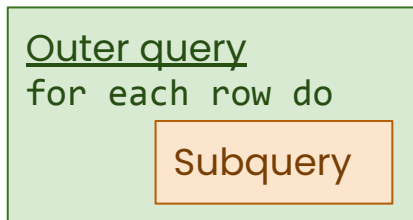
# How we recast the **subquery** problem

Two undesirable approaches

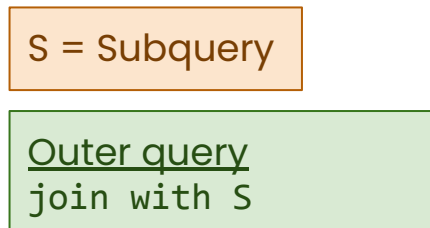
*(SQL systems attempt to rewrite and decorrelate to avoid these)*

```
select
  user.id,
  (
    select count(*)
    from post
    where post.user_id = user.id
  )
from user
where user.country = 'Mordor'
```

## Top-down: **Nested Loop**



## Bottom-up: **(over)-compute once and reuse**



We address subqueries with two powerful and general methods

1. Uncorrelated subqueries are handled by **semantic optimizer**
2. Embrace correlation: **WCOJ is also a correlated join device!**

# How we recast the **index selection** problem

Index-selection and auto-tuning is an unsolved problem.

RelationalAI users cannot be asked to manually define indexes, and even supervised tuning approaches are not acceptable.

Our solution:

- **Everything is an index in our graph-like schemas**

Compare: RDF triple stores that create indexes for all orderings

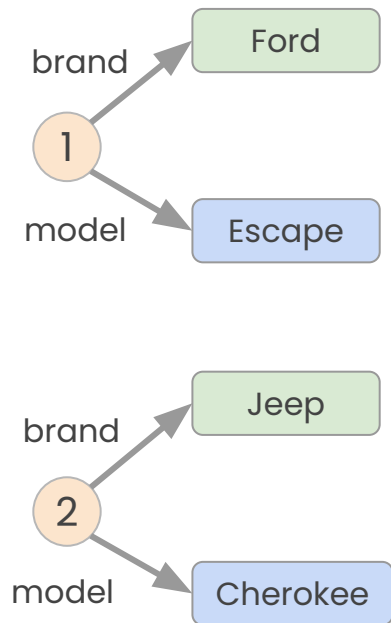
Compare: SQL table stores with an index for every functional dependency

- **WCOJ is a device to create **composite indexes on-the-fly**, cheaply**

# How we recast the **index selection** problem

WCOJ is a device to create composite indexes on-the-fly, cheaply

## Graph



## Index Building Blocks

brand

1	Ford
2	Jeep

Ford	1
Jeep	2

model

1	Escape
2	Cherokee

Escape	1
Cherokee	2

## Indexes available w/o sorting

Ford	Ford	Escape
Jeep	Jeep	Cherokee

Ford	1
Jeep	2

Ford	1	Escape
Jeep	2	Cherokee

Escape	Escape	Ford
Cherokee	Cherokee	Jeep

Escape	1
Cherokee	2

Escape	1	Ford
Cherokee	2	Jeep

1
2

1	Ford
2	Jeep

1	Ford	Escape
2	Jeep	Cherokee

1	Escape
2	Cherokee

1	Escape	Ford
2	Cherokee	Jeep

# Join Algorithms: Resources and Influences

## Worst-case optimal join algorithms

- **Worst-case Optimal Join Algorithms**  
Ngo, PODS 2012 (Best paper award)
- **Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm**  
Veldhuizen, ICDT 2015 (Best Newcomer Award)
- **A Worst-case Optimal Join Algorithm for SPARQL**  
Hogan, ISWC 2019
- **Worst-Case Optimal Graph Joins in Almost No Space**  
Arroyuelo, SIGMOD 2021

## Correlated Subqueries

- **Unnesting Arbitrary Queries**  
Neumann, BTW 2015
- **How Materialize and other databases optimize SQL subqueries**  
Brandon, Materialize Deep Dive, March 2021



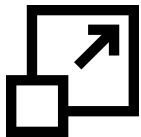
A Cloud-native Relational Database Platform  
Designed for structured and semi-structured dynamic data with

# Data and application logic together in one place

The RAI Knowledge Graph  
Management System (RKGMS)



# Core requirements



## Scalability

Separate data storage from compute



## Data independence

Separate application logic from data representation



## Semantic reasoning

Data and application logic together in database



## Live Programming

Incremental for changes to data and logic





# Data and application logic together in one place

## Overview



- Semantic Layer
- The Rel Language
- Use Case:  
FAA Flight Data
- Performance:  
Query Optimization

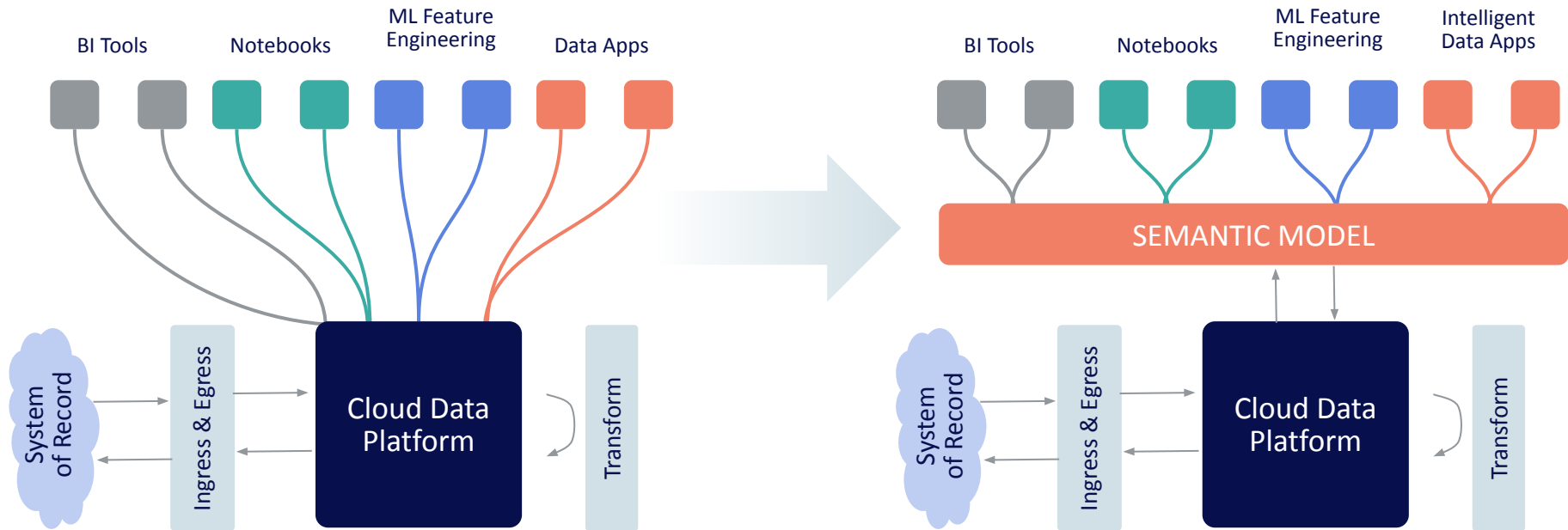
# Data and application logic together in one place

## Overview



- **Semantic Layer**
- The Rel Language
- Use Case:  
FAA Flight Data
- Performance:  
Query Optimization

# Knowledge and Semantics in the Modern Data Stack



# Approaches to Semantic Modeling

- SQL based
  - a. [DBT](#) ([announcement](#)) - SQL based approach
  - b. [Malloy](#) ([link](#)) - Google/Looker proposal to succeed LookML that maps to SQL
- Open industry based
  - a. [Legend](#) by Goldman Sachs- UML + OCL inspired approach. Runs on SQL
  - b. [Morphir](#) by Morgan Stanley ([intro video](#)) - Functional programming based approach. Runs on Spark
- Standards based:
  - a. [SBVR](#) - OMG standard for “Semantics of Business Vocabulary and Rules”
  - b. [Semantic Web](#) - W3C standard. SPARQL, RDF, [OWL](#), [SHACL](#)



# The Semantic Layer and Data Apps

Let's build a data app for a flight tracking database

## Example functionality:

- How many **transfers** do I need to travel between two cities?
- What percentage of flights were **delayed** this year by airline?
- If a percentage of flights arrive **delayed** to an airport per day, send an alert.

Knowledge/Semantics Layer

Flight database

The system **cannot answer** such questions if it does not know what **delay** and **transfer** mean to begin with!



# Flight Model and Queries

## Model

- `def cancelled(f) = Flight(f) and flight_cancelled(f, "Y")`
- `def diverted(f) = Flight(f) and flight_diverted(f, "Y")`
- `def arrived(f) = Flight(f) and not (cancelled(f) or diverted(f))`
- `def arrival_delay[f] = maximum[^Minute[0], arr_delay[f]]`

## Ask Question Over Well Defined Concepts

- `count[cancelled]`
- `mean[arrival_delay]`
- `c in Carrier : mean[arrival_delay[f] for f where operated_by(f, c)]`




# Data and application logic together in one place

## Overview



- Semantic Layer
- **The Rel Language**
- Use Case:  
FAA Flight Data
- Performance:  
Query Optimization

## Key innovations

1. **Immutability - Cloud native architecture**  
*Data + metadata is versioned and immutable*
  2. **Expressive relational language (Rel)**  
*Designed for complex business logic, abstraction, schema abstraction, libraries*
  3. **Join algorithms**  
*Worst-case optimal join algorithms specifically suitable for knowledge graphs*
  4. **Semantic optimization**  
*Use knowledge to apply deep and structural optimization*
  5. **Vectorized and just-in-time (JIT) compilation of WCOJ**  
*Compile complex joins to native code. Vectorized engine for simpler joins*
  6. **Incremental computation**  
*Maintain computations incrementally  
Compile model incrementally*
- 





# Rel in a nutshell

- ... designed to for semantic modeling and querying
- ... designed to express complex business logic
- ... able to handle structured and semi-structured data
- ... natural for graph use cases
- ... readable for business users
- ... easy for developers to pick up
- ... expressive, concise, and extensible



# Datalog - A Great Foundation

## Good

- Outstanding formal foundation
- Mutually recursive definitions

## More is needed

- Classic Datalog (globally stratified) is too limited for graph workloads:
  - Value creation, Aggregation, and Negation in recursion
- Datalog does not support abstraction (similar to SQL, Cypher, SPARQL etc)
  - Abstract over concrete relations
  - Abstract over schema

**Rel: Datalog is the IR**



# Rel language: Key Aspects

## Relational

- Data and logic is organized with relations

```
myrelation(key..., value)
```

$$\begin{bmatrix} -1.3 & 0.6 \\ 20.4 & 5.5 \\ 9.7 & -6.2 \end{bmatrix}$$


(1, 1, -1.3)  
(1, 2, 0.6)  
(2, 1, 20.4)  
(2, 2, 5.5)  
(3, 1, 9.7)  
(3, 2, -6.2)

## Declarative logic

- First-order logic based on datalog
- Principled way to express data manipulations and queries

```
def insert[:parent] = load_csv["azure://..."]  
  
def grandparent(x) =  
  exists(y, z: parent_of(x, y) and parent_of(y, z))
```



# Rel language: Key Aspects

## Abstraction

- Concise expression of knowledge
- Express path-query naturally

```
def reachable = reachable . connected
```

## Higher-order Logic

- Express complex business logic
- Enables reusable components

```
def mean[R]      = sum[R] / count[R]  
def argmax[R]    = R. (max[R])
```



# Rel language: Key Aspects

## Recursion

- Express complex business logic
- Non-stratified logic
  - support negation/aggregation

```
def damping = 0.85
def pagerank[x in node] =
  1.0, empty(pagerank[x])

def pagerank[y in node] =
  (1.0 - damping)
  + damping * sum[
    pagerank[x] / outdegree[x]
    for x where edge(x, y)
  ]
```

## Integrity Constraints

- Validate/enforce logical consistency
- Ensure data integrity (data types, domain)

```
ic check_string(s in R) {
  String(s)
}
```

```
ic is_transitive(id1, id2, id3) {
  is_older(id1, id2)
  and is_older(id2, id3)
  implies
  is_older(id1, id3)
}
```



# Rel language: Key Aspects

## Modules

- Modular organization of application logic
- Enable an ecosystem of libraries

```
def data = {2;3;5;7;11}

module stats
  def myAvg = average[data]
  def mySum = sum[data]
end

def output = stats:myAvg
```

## Schema Abstraction

- Schema and data are first-class citizen
- Handle structured and semi-structured data

```
def stats_name(name) = stats(name, _) and name != :myAvg
```



# Rel language: Key Aspects

## Schemaless Modeling

- Focus on the application logic
- Schemaless modeling (front-end)
- Type-stable stable DB (back-end)

```
def pi = {  
  "pi"; 'π';  
  3; 3.14;  
  decimal[64, 4, 3.14159];  
  rational[64, 22, 7];  
}
```

physical view

<b>String</b> pi	<b>Char</b> π	<b>Rational{Int64}</b> 22, 7
<b>Int64</b> 3	<b>Float64</b> 3.14	<b>FixedPointDecimals.FixedDecimal{Int64, 4}</b> 3.1416

logical view

#	Mixed
1	π
2	pi
3	3
4	rational[64, 22, 7]
5	3.14
6	3.1416



# Rel language: Key Aspects

## Entity / Value Types

- Abstract user-defined data types
- Enables expressive ontology
- Encoding semantic meaning

```
value type Hour(h) = {Int(h) and 0 <= h < 24}  
value type Minute(m) = {Int(m) and 0 <= h < 60}
```

```
value type Time = Hour, Minute  
value type TimeInterval = Time, Time
```

```
entity type Person = String, String  
entity type State = String
```

## Specialization

- Going between data and metadata
- Complex schema transformations
- User-controlled partitioning

```
def platform_allowed =  
  (#("1a"), "muggles");  
  (#(2), "muggles");  
  (#(9+3/4), {"wizards"; "witches"})
```





# Rel language: Key Aspects

## Native Transaction Control

- Relations instead of keywords
- Entangles application logic with transaction logic

```
def insert[:log](t, name, v...) = {  
  t = datetime_now  
  and insert(name, v...)  
  and name != :log  
}  
  
def output = log
```

## Reflective modeling

- Self-modifying logic
- Enables novel package management
- Enables code introspection
- Makes Rel extremely extensible

```
def insert:rel:catalog:model["Model1"] =  
  ""  
  def foo = 1  
  ""  
  
def output = foo
```



## Let's summarize

- Relational
- Declarative logic
- Higher-order Logic
- Recursion
- Abstraction
- Schema Abstraction
- Schemaless Modeling
- Modules
- Integrity Constraints
- Entity / Value Types
- Specialization
- Native Transaction Control
- Reflective modeling





Core Innovations for  
Relational Knowledge Graphs

**Interfaces: SQL**  **Rel**

# DuckDB-based SQL Interface

DuckDB is an embeddable SQL OLAP database management system with great performance, excellent quality, small footprint and enjoying quick adoption.

RAI uses DuckDB for SQL support. Rel is used to model SQL tables, which are used by DuckDB for SQL query evaluation. Individual 'columns' can be data vs views.

DuckDB has outstanding support for working with a dynamic catalog.

Other approaches we evaluated:

- Calcite
- DuckDB query plan
- PostgreSQL parser

```
module order
  def orderkey[o] = ...
  def customer[o] = ...
  def orderdate[o] = ...
  def totalprice[o] = sum[num: charge[o, num]]
end
```



```
SELECT orderkey, customer, orderdate
FROM order
WHERE totalprice > 100
```

# Data and application logic together in one place

## Overview

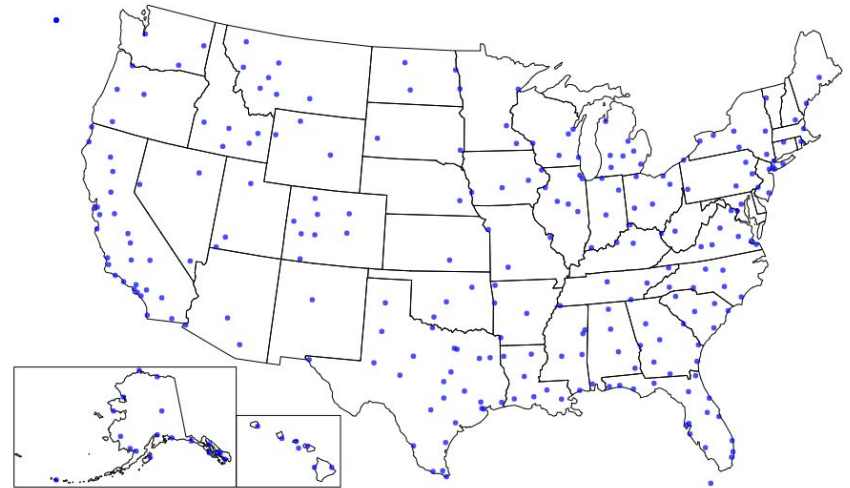


- Semantic Layer
- The Rel Language
- **Use Case:  
FAA Flight Data**
- Performance:  
Query Optimization

# Flight Data: How to build an intelligent data app

## Opportunities in such an application

- Performance of carriers, airports, aircraft models and individual aircraft
- Discover relationships between seasons, time of day, weather and delays
- Planning and optimization
- Historical trends
- If live, act on issues that are arising



# Rel language: Key Aspects

- Relational
- Declarative logic
- Higher-order Logic
- Recursion
- Abstraction
- Schema Abstraction
- Schemaless Modeling
- Modules
- Integrity Constraints
- Entity / Value Types
- Specialization
- Native Transaction Control
- Reflective modeling



Primary key?

Hours?  
Minutes?

Period?  
Minutes?  
Hours?

Miles?  
Kilometers?

Are these  
exclusive?

carrier	origin	destination	flight_num	flight_time	tail_num	dep_time	arr_time	dep_delay	arr_delay	taxi_out	taxi_in	distance	cancelled	diverted	id2
F9	MCI	DEN	818	89	N916FR	2005-09-26 08:23:00 UTC	2005-09-26 09:07:00 UTC	-7	-8	9	6	533	N	N	36606824
WN	LBB	ELP	819	41	N708SW	2000-08-18 07:30:00 UTC	2000-08-18 07:20:00 UTC	0	-5	7	2	295	N	N	4369021
NW	ATL	MEM	819	52	N607NW	2001-11-16 07:15:00 UTC	2001-11-16 07:29:00 UTC	-5	-9	19	3	332	N	N	11838308
DL	SLC	BOI	819	48	N296WA	2001-12-04 22:12:00 UTC	2001-12-04 23:53:00 UTC	7	4	49	4	291	N	N	12060416
WN	PHX	SAN	819	51	N391SW	2001-12-05 09:11:00 UTC	2001-12-05 09:17:00 UTC	11				304	N	N	12383068
WN	LAS	AUS	819	135	N519SW	2002-04-05 08:18:00 UTC	2002-04-05 12:47:00 UTC	8				1085	N	N	13763279
WN	SJC	LAS	819	63	N528SW	2002-07-14 06:30:00 UTC	2002-07-14 07:47:00 UTC	0	-3	11	3	386	N	N	15284777
WN	LAS	AUS	819	137	N502SW	2002-09-16 08:20:00 UTC	2002-09-16 12:52:00 UTC	0	-8	11	4	1085	N	N	16027516
DL	MSP	SLC	819	149	N3754A	2002-09-29 19:34:00 UTC	2002-09-29 21:35:00 UTC	9	15	25	7	991	N	N	16399211
DL	MSP	SLC	819	145	N3745B	2002-12-06 19:27:00 UTC	2002-12-06 21:16:00 UTC	-3	-9	18	6	991	N	N	17417961
WN	SJC	LAS	819	54	N730SW	2003-06-26 06:30:00 UTC	2003-06-26 07:44:00 UTC	0	-11	15	5	386	N	N	20460576
WN	SJC	LAS	819	60	N501SW	2003-09-15 06:30:00 UTC	2003-09-15 07:50:00 UTC	0	0	18	2	386	N	N	22113592
NW	ATL	MEM	819	51	N785NC	2003-11-20 07:11:00 UTC	2003-11-20 07:15:00 UTC	-9	-23	10	3	332	N	N	23383534
DL	MSP	ATL	819	120	N906DE	2004-02-10 09:59:00 UTC	2004-02-10 13:36:00 UTC	-6	0	30	7	906	N	N	25206983
US	CHS	CLT	820	38	N592US	2002-07-01 19:32:00 UTC	2002-07-01 20:54:00 UTC	37	64	9	35	168	N	N	15142411
FL	TPA	ATL	820	64	N955AT	2003-01-31 11:29:00 UTC	2003-01-31 12:55:00 UTC	-6	-5	13	9	406	N	N	17949329
WN	RDU	PHL	820	73	N382SW	2004-12-02 11:10:00 UTC	2004-12-02 12:31:00 UTC	0	-19	5	3	336	N	N	30796766
HP	PHX	BOS	820	0	N826AW	2005-10-25 00:00:00 UTC	2005-10-25 00:00:00 UTC	0	0	0	0	2300	Y	N	37174931
US	PBI	CLT	821	90	N624AU	2002-05-18 06:35:00 UTC	2002-05-18 07:07:00 UTC	-5	-8	10	6	590	N	N	14247814
US	PBI	CLT	821	87	N624AU	2002-05-18 06:50:00 UTC	2002-05-18 07:43:00 UTC	-7	-7	16	7	590	N	N	20289351
DL	GSO	CVG	821	70	N943DL	2002-05-18 08:12:00 UTC	2002-05-18 08:12:00 UTC	1	11	14	7	330	N	N	21396171

Not a delay

Risk of  
messing up  
aggregates

Include helicopters?

Possible values?



# The Semantic Layer and Data Apps

Let's build a data app for a flight tracking database

## Example functionality:

- How many **transfers** do I need to travel between two cities?
- What percentage of flights were **delayed** this year by airline?
- If a percentage of flights arrive **delayed** to an airport per day, send an alert.

Knowledge/Semantics Layer

Flight database

The system **cannot answer** such questions if it does not know what **delay** and **transfer** mean to begin with!



# Model

## Better conceptual model

```
def Heliport(x in Airport) =  
  fac_type(x, "HELIPORT")
```

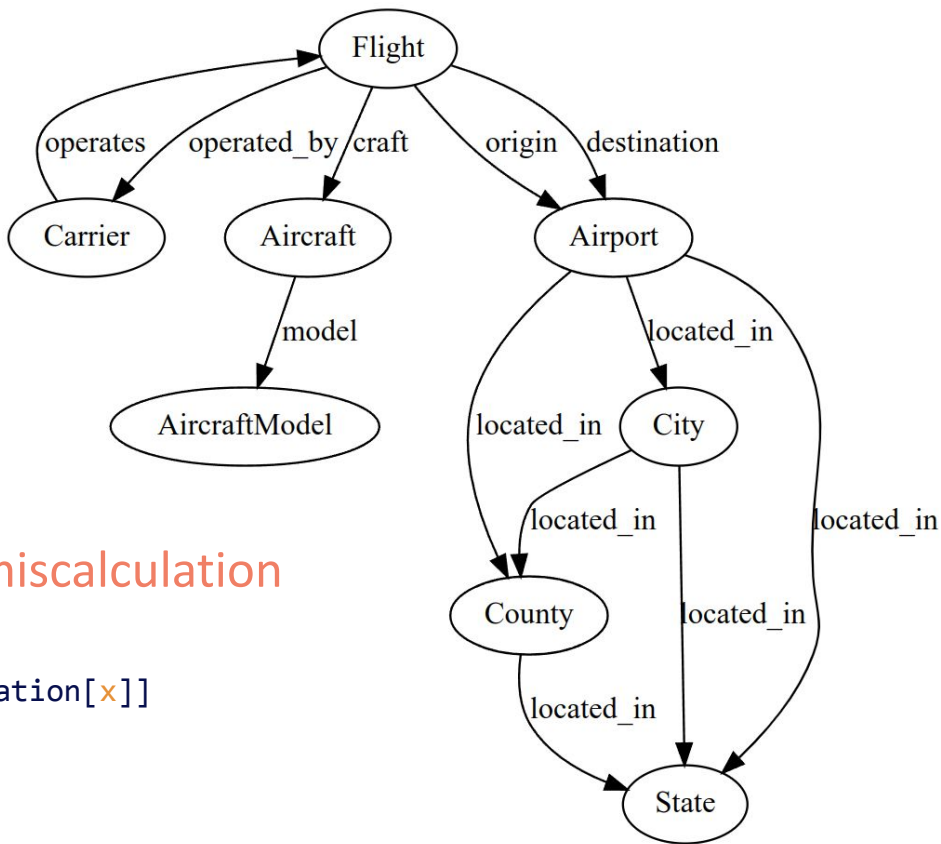
```
def cancelled(f in Flight) =  
  flight_cancelled(f, "Y")
```

```
def arrival_delay[f in Flight] =  
  ^Minute[maximum[0, arr_delay[f]]]
```

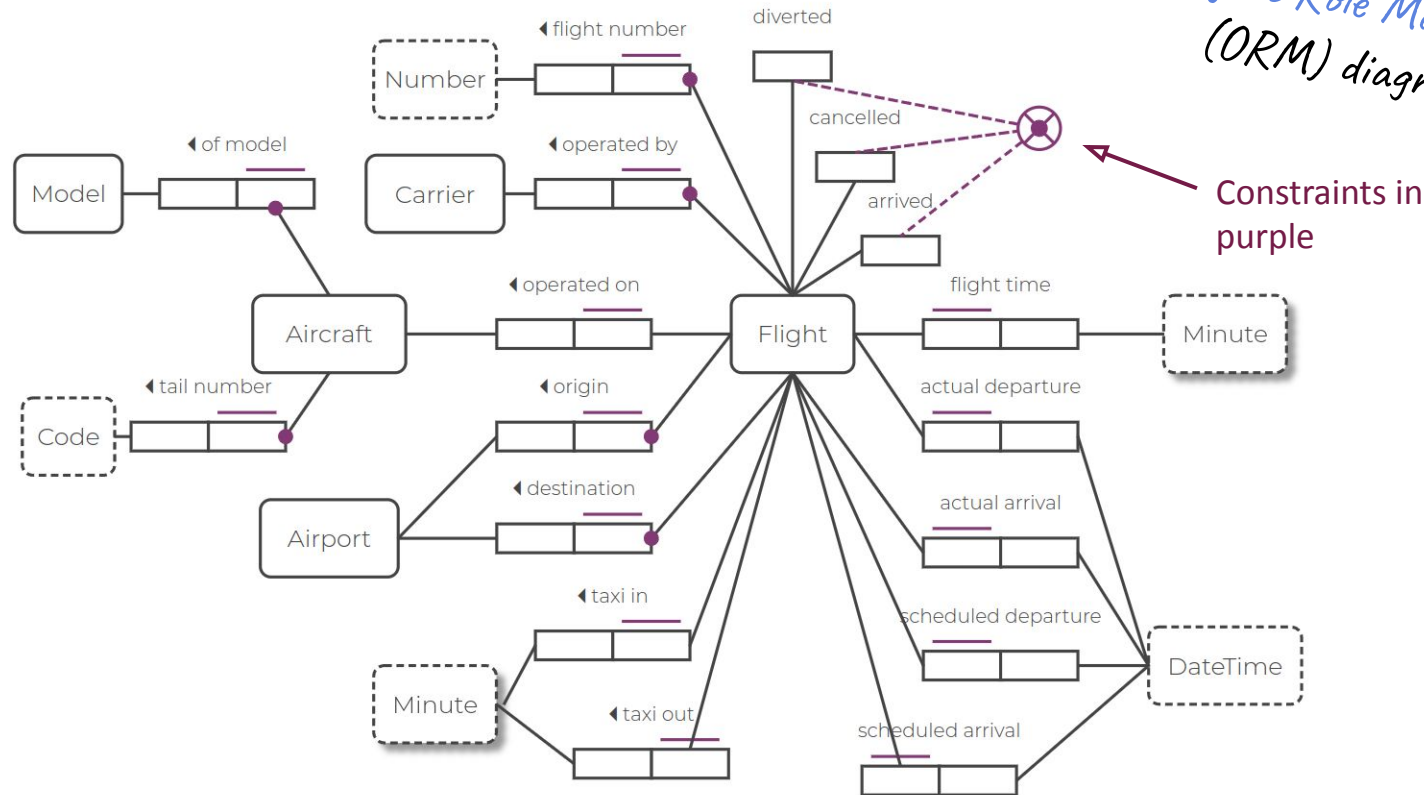
## Units of measurements to prevent miscalculation

```
def coordinate[x in Airport] =  
  ^LLA[latitude[x], longitude[x], elevation[x]]
```

```
def LengthUnit = :Feet; :Meters; :Miles  
value type Length = LengthUnit, Number  
value type Degree = Number  
value type LLA = Degree, Degree, Length
```



# A Better Flight Model



# Data Integrity

## Nodes involved in relationships

```
ic forall(f, ap: origin(f, ap) implies Flight(f) and Airport(ap))
```

## Required relationships

```
ic forall(f: Flight(f) implies exists origin[f])
```

## Functional dependency (flight can have only one origin)

```
ic function(origin)
```

## Arbitrarily complex

```
ic forall(f in cancelled: not exists flight_duration[f])  
ic forall(f in flight: cancelled(f) xor diverted(f) xor arrived(f))
```



# Aggregation

## Total number of flights

```
count[Flight]
```

37,561,525

## Carrier with most flights

```
c: count[f: operated_by(f, c)]
```

Southwest	5,775,777
Delta	4,477,929
American	4,434,727

## Carriers mean arrival delay

```
c: mean[f.arrival_delay for f where operated_by(f, c)]
```

Airtran	15 min
Atlantic Coast	13 min
United Airlines	13 min
...	
Aloha Airlines	6 min
Hawaiian Airlines	3 min

## Airport ratio of cancelled arriving flights

```
ap: ratio[cancelled, ap.arriving_flight]
```

Unalaska	19%
Worcester Regional	11%
Nantucket Memorial	9%



# Reasoning

## Rel supports general mutually recursive definitions + type inference

```
def located_in(x, y) = exists(t: located_in(x, t) and located_in(t, y))
```

## General computations

```
def airport_distance[ap1 in Airport, ap2 in Airport] =  
  distance[coordinate[ap1], coordinate[ap2]]
```

```
def distance[x in LLA, y in LLA] = haversine_function[earth_radius, x, y]
```

```
def earth_radius = ^Length[:Kilometers, 6378.1]
```

## Library abstractions and aggregation in recursive definitions

```
def transfer_count[c in Carrier, ap1 in Airport, ap2 in Airport] =  
  shortest_path_length[route[c], ap1, ap2]
```



# Schema Abstraction

Rel exposes the schema logically as data making it “feel” like a dynamic language.  
To evaluate logic over schema, partial evaluation methods and specialization are used.

## Count all nodes

38,061,144

```
count[node_label, node_id:  
  flight_graph(node_label, node_id)  
]
```

Flight	37,561,525
Aircraft	359,928
AircraftModel	60,461
City	50,944
Airport	19,793
Heliport	5,135
County	3,009
Major	270
State	58
Carrier	21

## Count all nodes, grouped by type

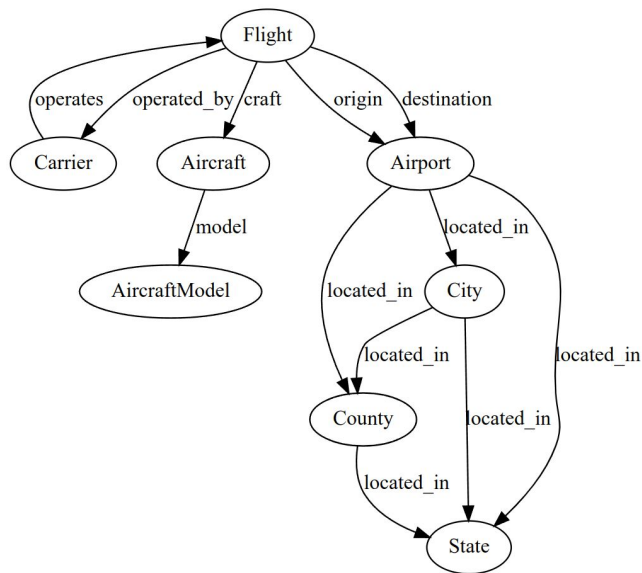
```
node_label: count[node_id:  
  flight_graph(node_label, node_id)  
]
```



# Schema Abstraction

## Schema data are first-class citizens

- Reason, query, and visualize schema data
- Library can be applied to schema and data



```
module schema_graph[G]
  def node(x) = G(x, _)
  def edge(e, tx, ty) = {
    G(e, x, y) and
    G(tx, x) and
    G(ty, y) and
    Entity(x) and
    Entity(y)
    from x, y
  }
end

def output = graphviz[schema_graph[flight_graph]]
```





# Schema Abstraction

## Schema: shortest path from Flight to State

```
shortest_path[schema_graph[flight_graph], :Flight, :State]
```

```
Flight -> destination -> Airport -> located_in -> State
```

```
Flight -> origin -> Airport -> located_in -> State
```

## Schema: all acyclic paths from Flight to State

```
acyclic_path[schema_graph[flight_graph], :Flight, :State]
```

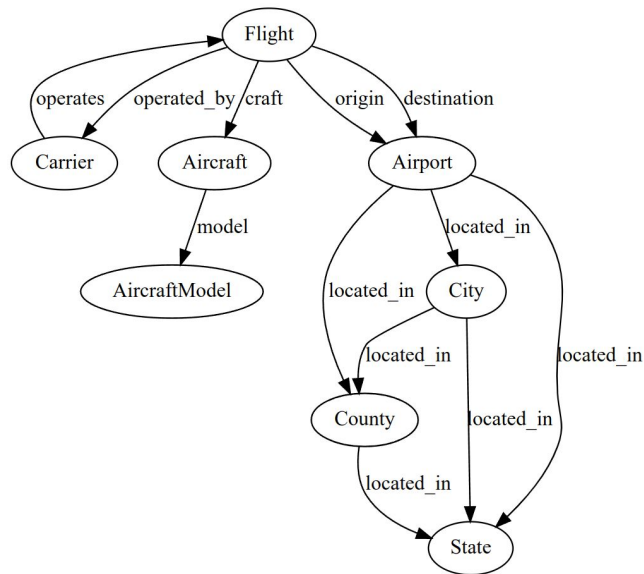
```
Flight -> destination -> Airport -> located_in -> City -> located_in -> County -> located_in -> State
```

```
Flight -> destination -> Airport -> located_in -> City -> located_in -> State
```

```
Flight -> destination -> Airport -> located_in -> County -> located_in -> State
```

```
Flight -> destination -> Airport -> located_in -> State
```

...



*Note: The path algorithms are written in Rel (not foreign functions)*



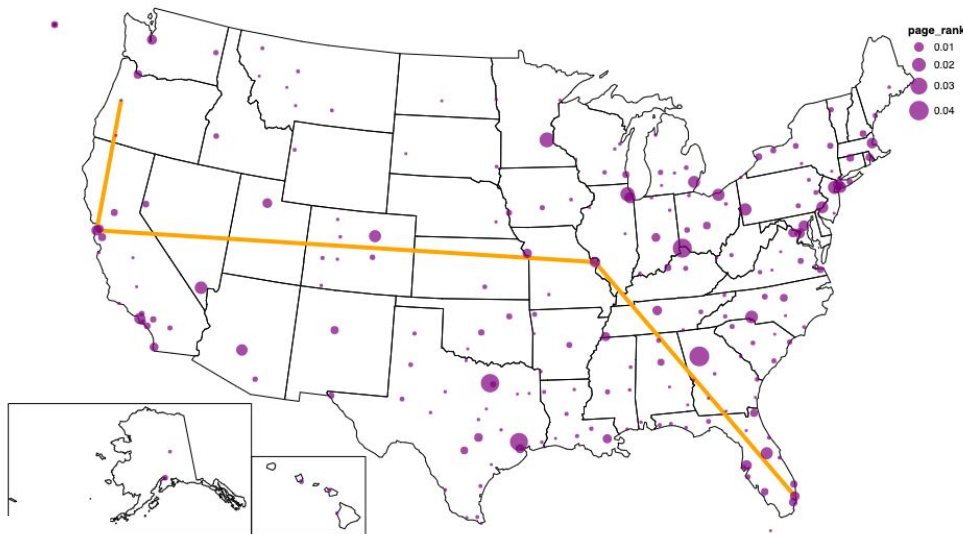
# Graph Analytics

Rel can express graph algorithms, for example **pagerank** and **shortest path**.

**Pagerank** for major airports

Highlighted is a **shortest path** between two airports.

*Rel supports **geographical data** and **JSON**.  
The maps are computed in Rel from shapes  
of the states, part of the knowledge base.  
Visualization is Vega-Lite.*



# Data and application logic together in one place

## Overview



- Semantic Layer
- The Rel Language
- Use Case:  
FAA Flight Data
- **Performance:  
Query Optimization**

# Key innovations

## 1. Immutability - Cloud native architecture

*Data + metadata is versioned and immutable*

## 2. Expressive relational language (Rel)

*Designed for complex business logic, abstraction, schema abstraction, libraries*

## 3. Join algorithms

*Worst-case optimal join algorithms specifically suitable for knowledge graphs*

## 4. Semantic optimization

*Use knowledge to apply deep and structural optimization*

## 5. Vectorized and just-in-time (JIT) compilation of WCOJ

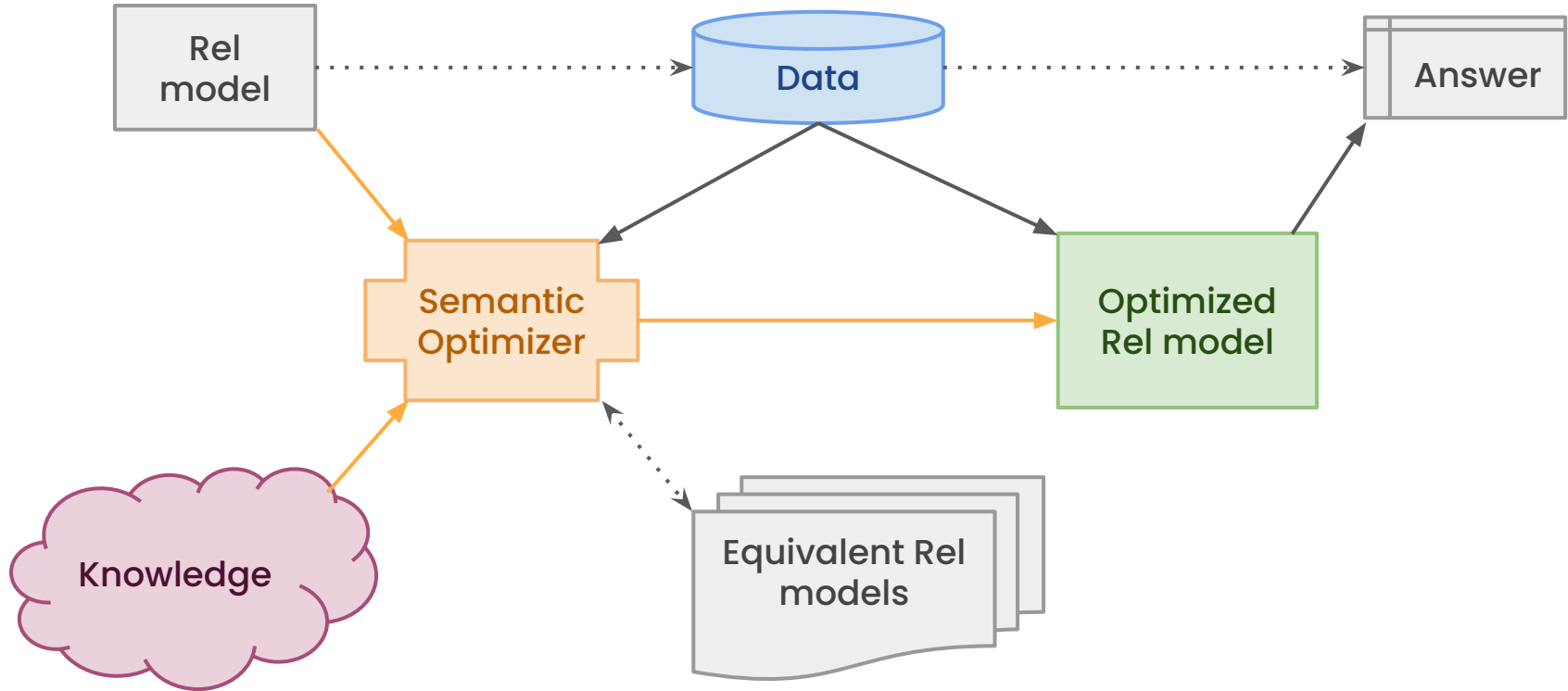
*Compile complex joins to native code. Vectorized engine for simpler joins*

## 6. Incremental computation

*Maintain computations incrementally  
Compile model incrementally*



# Semantic Optimization



# What Knowledge

## User-specified constraints

- Functional dependencies etc
- Total functions, disjoint etc

## Mathematical axioms

- Semirings, rings, fields, lattices, ...

## Learned from the data

- Data: Summary statistics, histograms
- Query: Samples cardinality estimation

$x + y = y + x$	commutativity of $+$
$x \times y = y \times x$	commutativity of $\times$
$z \times (x + y) = z \times x + z \times y$	distributivity of $\times$ over $+$
$x \times 1 = x$	identity of $\times$ is 1
$x + 0 = x$	identity of $+$ is 0
$x \times 0 = 0$	0 is an annihilator

$\min(x, y) = \min(y, x)$	commutativity of $\min$
$x + y = y + x$	commutativity of $+$
$z + \min(x, y) = \min(z + x, z + y)$	distributivity of $+$ over $\min$
$x + 0 = x$	identity of $+$ is 0
$\min(x, +\infty) = x$	identity of $\min$ is $+\infty$
$x + \infty = \infty$	$\infty$ is an annihilator



# Semantic Optimization

Using mathematical knowledge in semantic optimization

`min[i, j: f[i] + g[j]]`

optimizer

`min[f] + min[g]`

`min[i: f[i] + g[i]]`

optimizer

`min[f] + min[g]`

`count[f × g]`

optimizer

`count[f] * count[g]`



# Semantic Optimization is not Syntactic or Ad-hoc

count[x, y: R(x) and S(y) and x != y]

optimizer

$$1_{x \neq y} = 1 - 1_{x=y}$$

count[R] \* count[S] - count[x, y: R(x) and S(y) and x = y]



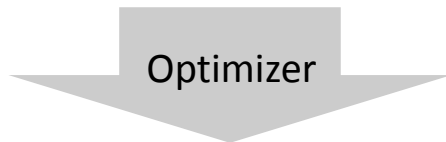


# Semantic Optimization: Push Agg into Recursion

Push min aggregation into a recursive path to derive Dijkstra's algorithm

```
def path[x, y] = edge[x, y]
def path[x, y] = path[x, t] + edge[t, y] from t

def shortest_path_length[x, y] = min[path[x, y]]
```



```
def shortest_path_length[x, y] = min[
  edge[x, y];
  shortest_path_length[x, t] + edge[t, y] from t
]
```



# Semantic Optimization: Resources and Influences

- FAQ: Questions Asked Frequently  
Khamis, Ngo, Rudra, PODS 2016 (Best Paper Award)
- What Do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another  
Khamis, Ngo, Suciu, PODS 2017
- Precise complexity analysis for efficient Datalog queries  
Tekle et al., PPDP 2010
- Functional Aggregate Queries with Additive Inequalities  
Khamis et al., PODS 2019
- Convergence of Datalog over (Pre-) Semirings  
Khamis, Ngo, Pichler, Suciu, Wang, PODS 2022 (Best paper award)
- Factorised representations of query results: size bounds and readability  
Olteanu, Zavodny, ICDT 2012 (2022 Test of time award)



## Key innovations

- ## 1. Immutability - Cloud native architecture

*Data + metadata is versioned and immutable*

- ## 2. Expressive relational language (Rel)

*Designed for complex business logic, abstraction, schema abstraction, libraries*

- ### 3. Join algorithms

*Worst-case optimal join algorithms specifically suitable for knowledge graphs*

- ## 4. Semantic optimization

*Use knowledge to apply deep and structural optimization*

- ## 5. Vectorized and just-in-time (JIT) compilation of WCOJ

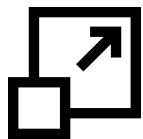
Compile complex joins to native code. Vectorized engine for simpler joins

- ## 6. Incremental computation

Maintain computations incrementally  
Compile model incrementally

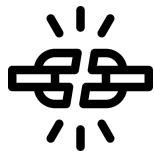


# Core requirements



## Scalability

Separate data storage from compute



## Data independence

Separate application logic from data representation



## Semantic reasoning

Data and application logic together in database



## Live Programming

Incremental for changes to data and logic



# The Incremental Maintenance Stack

RAI aims to support incremental processing of changes to **code** as well as **data**.

**Dependency tracking** to determine which computations are affected by a change.

**Demand-driven execution** to only compute what users are actively interested in.

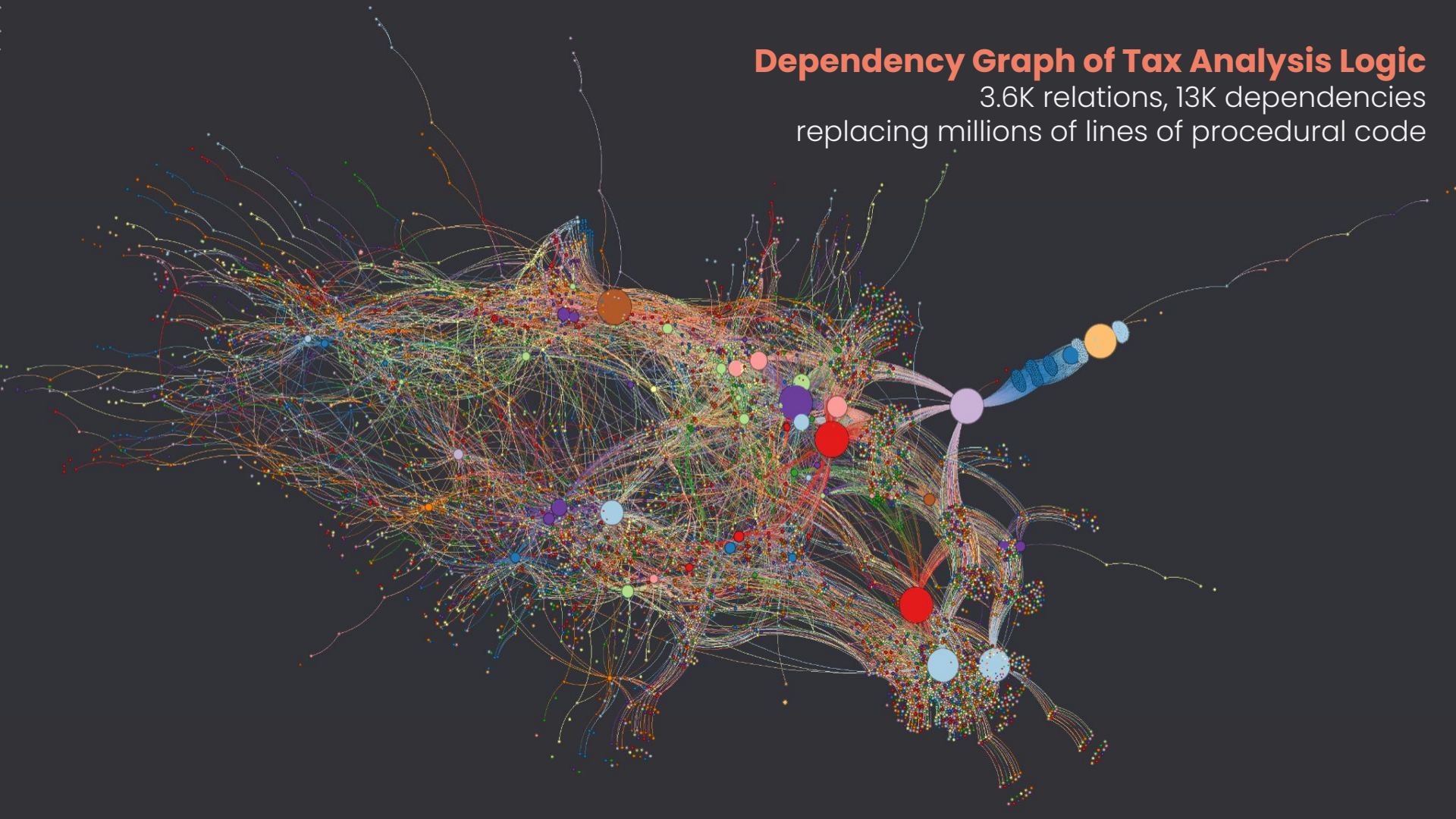
**Differential computation** to incrementally maintain even general recursion.

**Semantic information** to determine that a recursive computation is monotonic

**Semantic optimization** to recover better maintenance algorithms where possible.

## Dependency Graph of Tax Analysis Logic

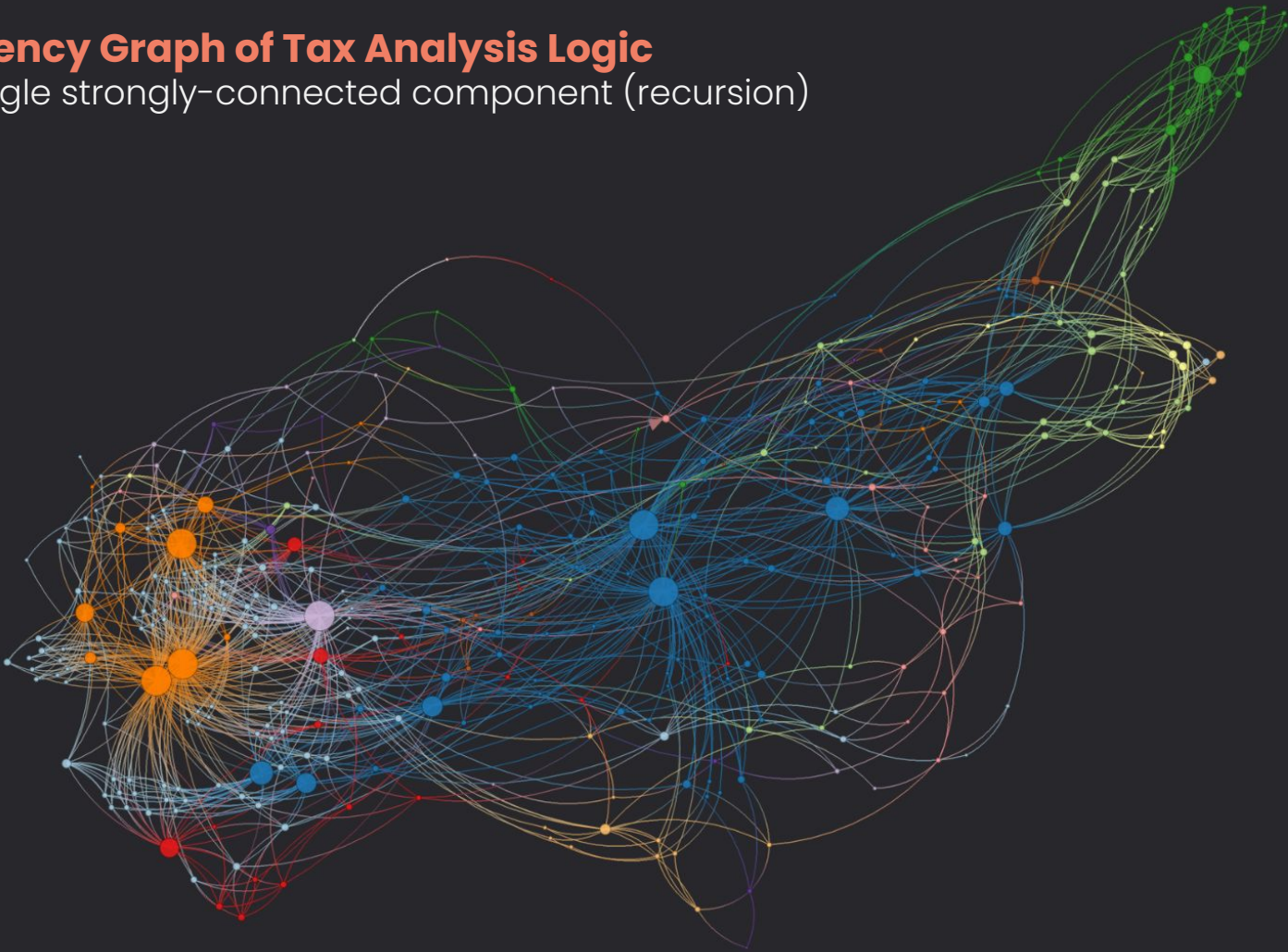
3.6K relations, 13K dependencies  
replacing millions of lines of procedural code





# Dependency Graph of Tax Analysis Logic

Focus: Single strongly-connected component (recursion)

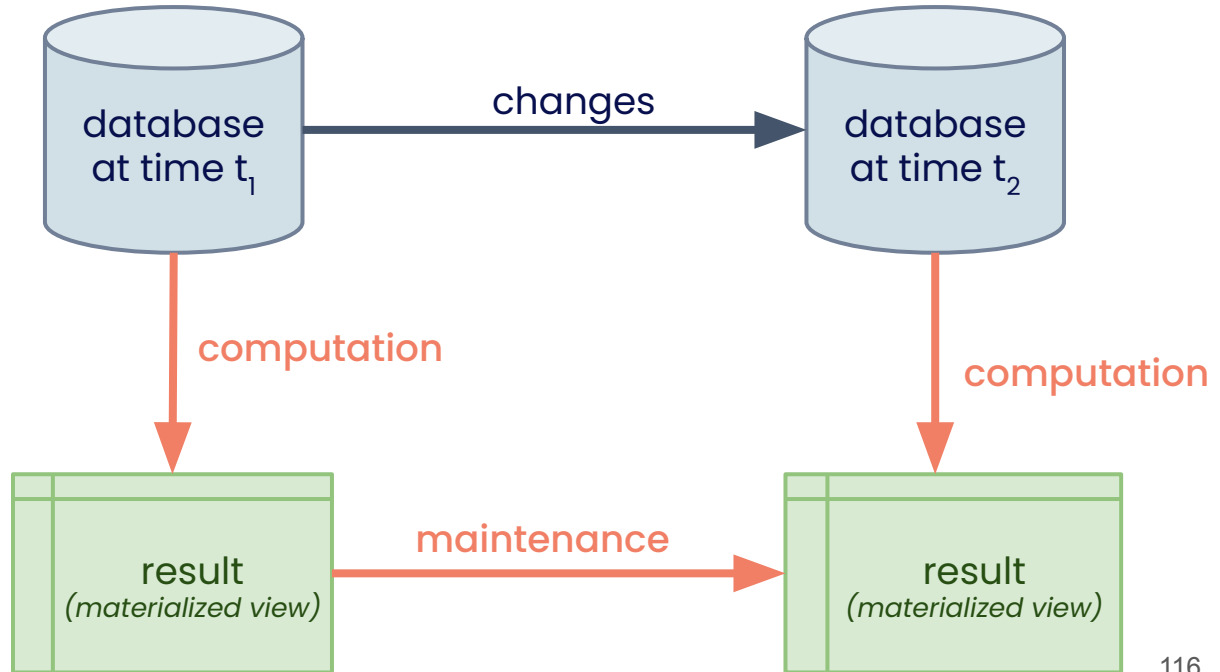


# Incremental Computation

Goal: maintain computations (views) incrementally wrt changes in the inputs.

Inputs can change along two dimensions:

I) Changes caused by changes to the state of the database



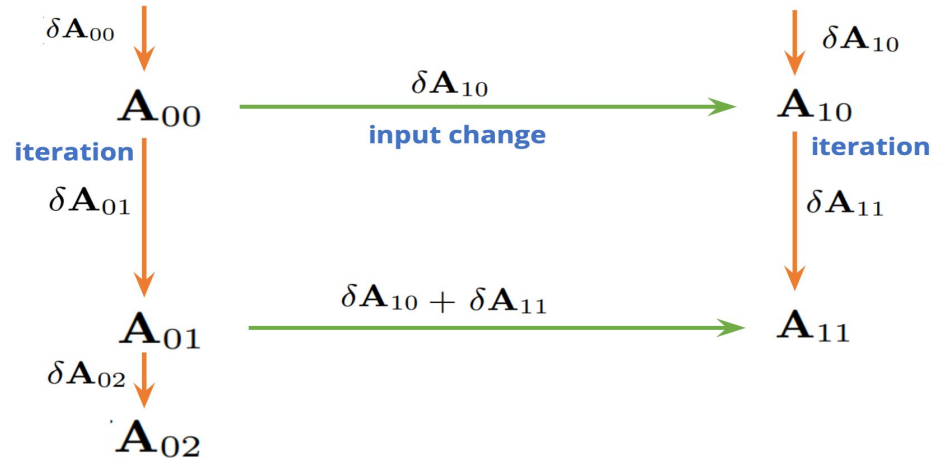


# Incremental Computation

Goal: maintain computations (views) incrementally wrt changes in the inputs.

Change along two dimensions:

- Changes caused by changes to the state of the database
- Changes caused by iterative computations



# Incrementality and Demand-driven Evaluation

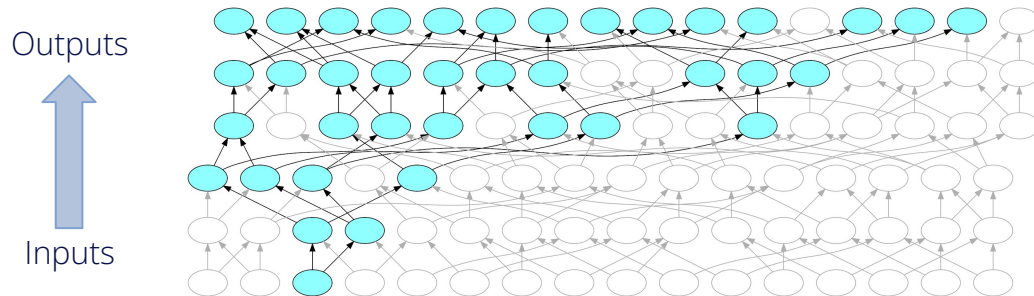
The architecture is based on PL incremental compiler research for IDEs.

Key ingredients:

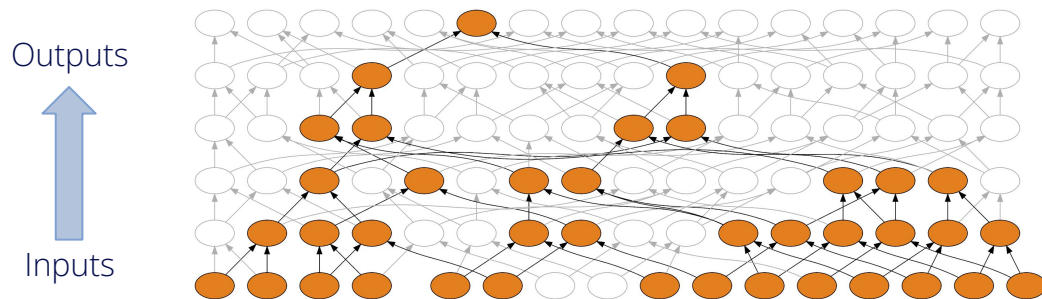
- Precise dependency tracking (treat access to the catalog as queries)
- Memoization and invalidation (on input changes)

We've open-sourced **Salsa.jl**, our framework for writing responsive compilers.

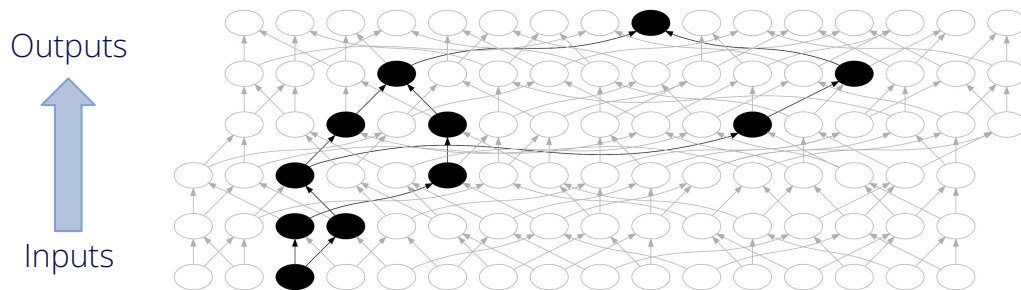
- [Responsive compilers - Nicholas Matsakis - PLISS 2019](#)
- [JuliaCon 2020 - Salsa.jl - Nathan Daly](#)



**Eager maintenance is bad**



**Lazy maintenance is bad**  
*detecting dirty computations is too expensive when an output is queried.*



**Best: Eager invalidation  
lazy evaluation**

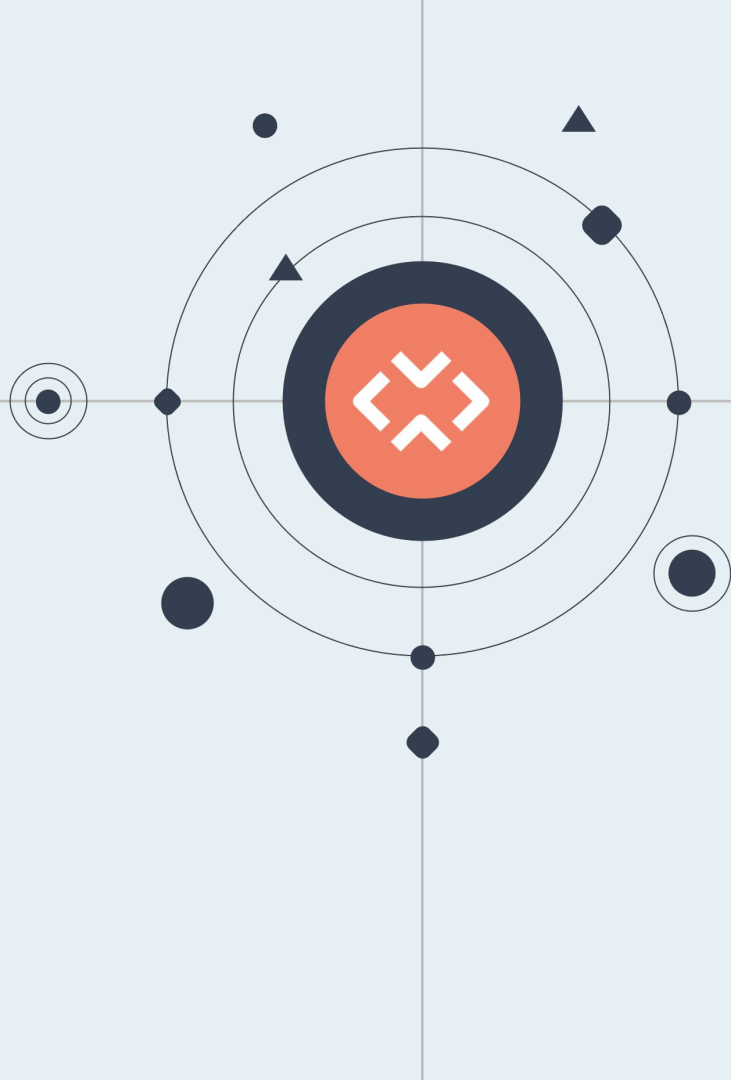
# Incremental Computation: Resources and Influences

- **Convergence of Datalog over (Pre-) Semirings**  
Abo Khamis, Ngo, Pichler, Suciu, Wang, PODS 2022 (Best paper award)
- **Differential dataflow**  
McSherry, Murray, Isaacs, Isard, CIDR 2013
- **Reconciling Differences**  
Green, Ives, Tannen, Theory of Computing Systems 2011
- **F-IVM: Incremental View Maintenance with Triple Lock Factorization Benefits**  
Nikolic and Olteanu, SIGMOD 2018



# The Future of Intelligent Databases

The RAI Knowledge Graph  
Management System (RKGMS)



# The Future of Intelligent Databases

## Overview



- Predictive Reasoning
- Use case:  
Battleship engine
- Mathematical  
Optimization
- Relational ML

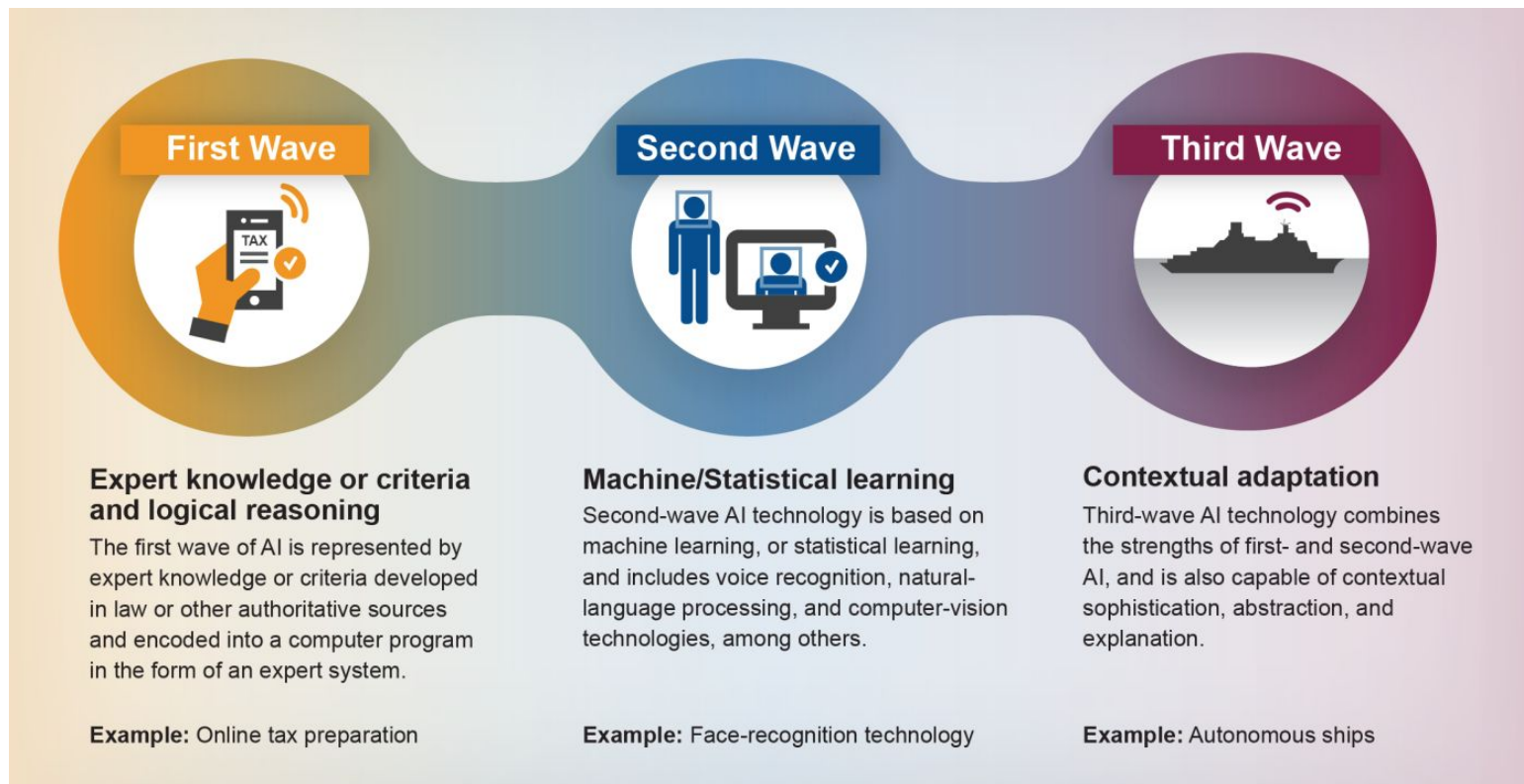
# The Future of Intelligent Databases

## Overview



- **Predictive Reasoning**
- Use case:  
Battleship engine
- Mathematical  
Optimization
- Relational ML



# The Three Waves of Artificial Intelligence





# Making a Database intelligent

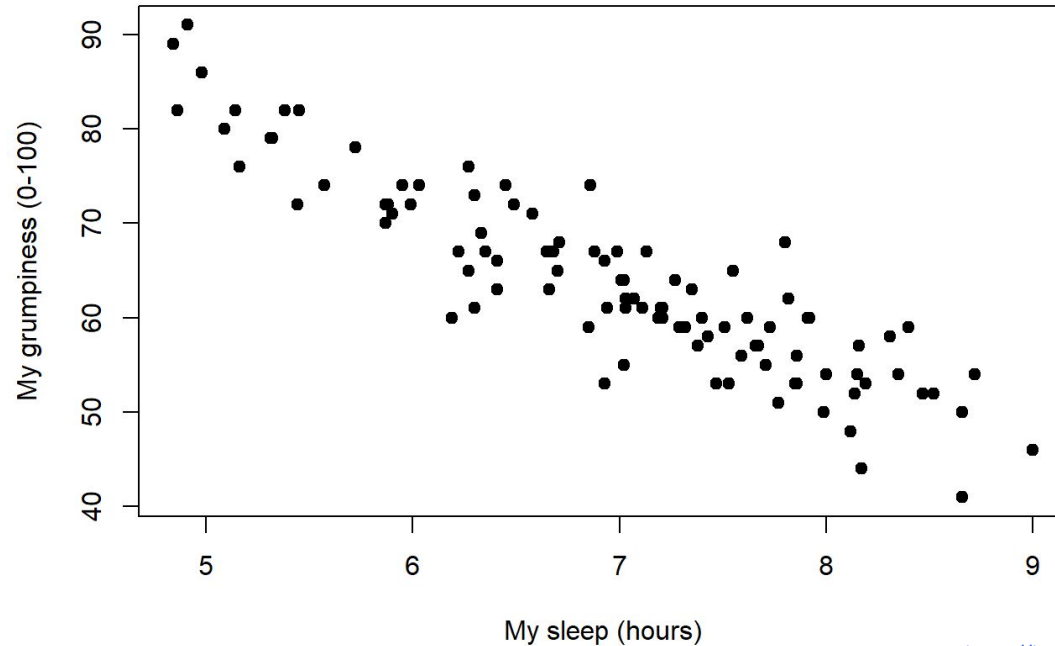
There two main approaches to make a DB intelligent (AI + DB):

Binding to external packages	Pure in-DB support
<ul style="list-style-type: none"><li>• Leverage the power of existing specialized packages</li><li>• Support a wide variety of special needs (GLM, decision trees, ...)<ul style="list-style-type: none"><li>• ML: xgboost, mlpack, ... </li><li>• Optimization: CBC, ... </li></ul></li><li>• Rel Modules are ideal to package these bindings</li></ul>	<ul style="list-style-type: none"><li>• Make full use of the relational model</li><li>• Leverage the power of our join algorithms and the query optimizer<ul style="list-style-type: none"><li>○ Can unlock large speed-ups</li></ul></li><li>• Targeted AI Methods<ul style="list-style-type: none"><li>○ Relational ML</li><li>○ Mathematical Optimization</li><li>○ Simulations &amp; probabilistic models/databases</li></ul></li></ul>

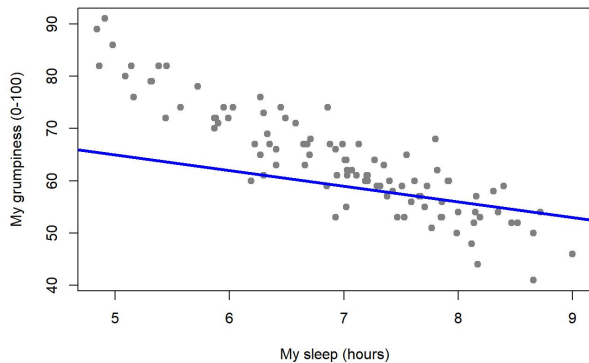
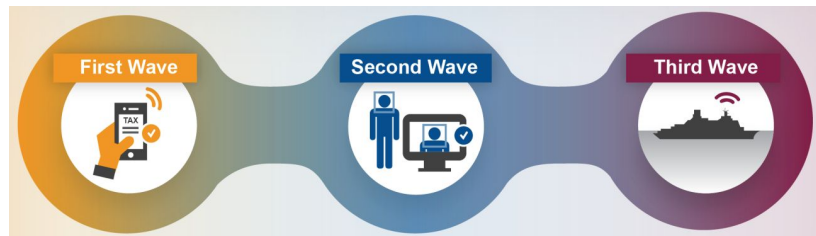


# Simulations

Simulations are key to model a world with uncertainty and noise.

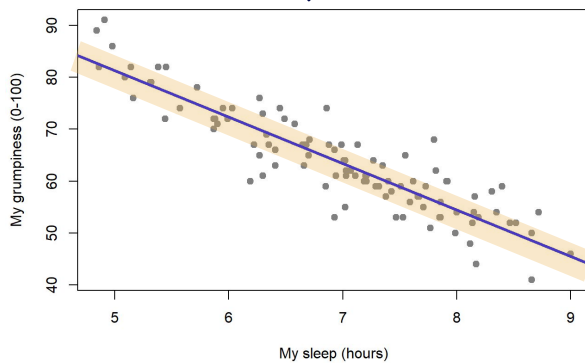


# Simulations



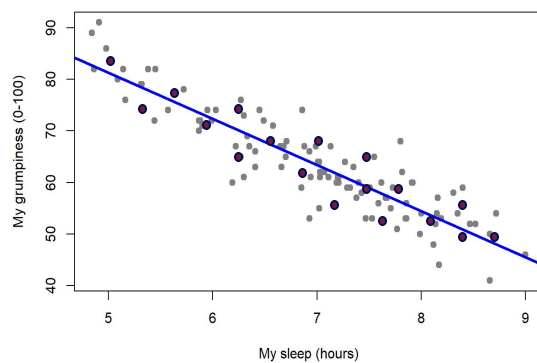
Expert Knowledge  
(Best guess)

$$y = c_{\text{expert}} x + b$$



Data-Driven Optimization  
(Best fit, MLE)

$$y = \hat{c} x + b$$



Probabilistic Modeling  
(Simulate uncertainty)

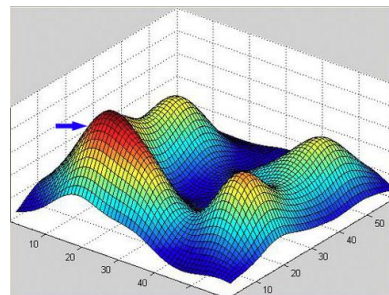
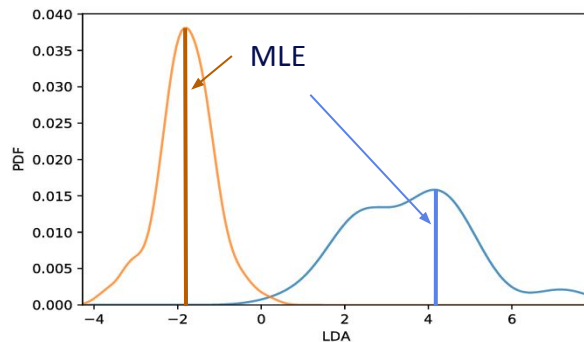
$$y = c x + b$$

$$c \sim \text{Normal}(\mu, \tau)$$

# Simulation - Variables are Distributed

Probabilistic modeling and simulations treat uncertainty as first-class citizen.

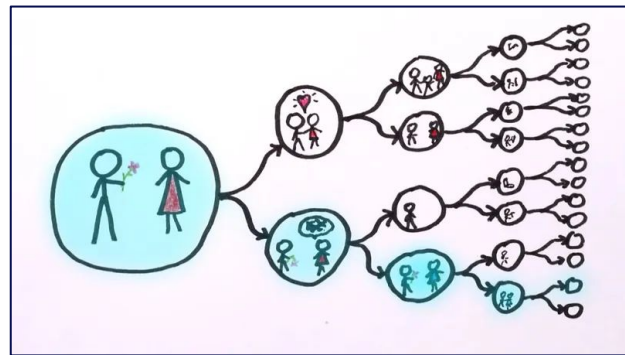
MLE (maximum likelihood estimates) become problematic for multimodal distributions.



# Simulations in a Database

Simulations compute the possible outcomes of a **probabilistic model**.

**Probabilistic models** are **logical models** with **random variables**.



**Key requirements** for **in-DB simulations**:

- Evaluation of probabilistic models needs to be deterministic
- Stable under repetitive evaluation (to support IVM)
- Stable under execution reordering (semantic query optimization)



# Randomness in the Database

How to incorporate randomness in a deterministic environment?

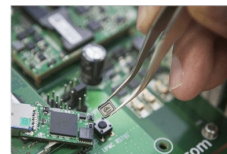
Pseudorandom number generators (PRNG)

Our RKGMS supports:

- Multiple PRNGs (Threefry, MT, ...)
- Support two common sources of non-deterministic randomness



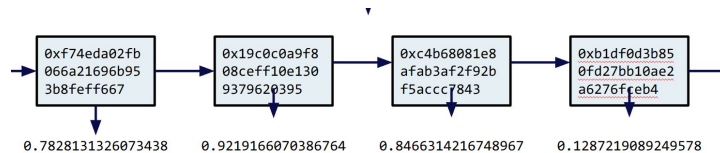
Time	Random seed
Transaction/creation times are accessible in Rel.	OS random device initialized per transaction.



# PRNGs for Declarative Programming

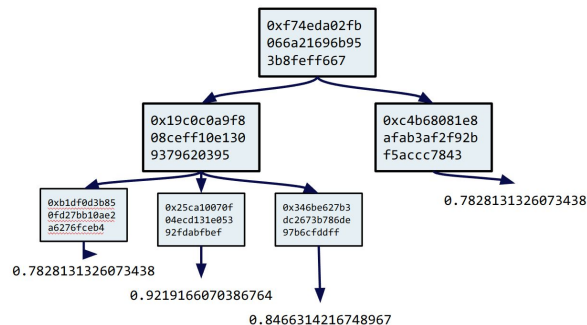
## Serial PRNGs are non ideal for declarative programming

- PRNG state evolve sequentially
- Values depend on the execution order
- Re-evaluating the same statement gives different results



## Counter-base PRNG have advantageous properties

- PRNG states evolve tree-like
- Tree-like structure can resemble logic program order
- Parallel evaluation



# Simulation: Throwing Dice in Rel

## Case 1:

Throwing a dice multiple times



Define random variable

```
def die[key] = uniform[key][1, 6]
```



Throwing dice

```
def output = die[key] for key in range[1,5,1]
```



#	Int64	Int64
1	1	2
2	2	5
3	3	4
4	4	2
5	5	5

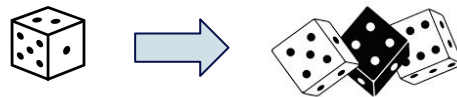




# Simulation: Throwing Dice in Rel

## Case 2:

Throwing  $n$  dice depending on the face value ( $n$ ) of the first die



Number of **random variables** isn't fixed

```
def data = multiple_dice[k]
  for k in range[1, 3, 1]

def multiple_dice[key] =
  die[threefry[key, i]]
  for i in range[1, die[key], 1]

def output = data
```



#	Int64	Int64	Int64
1	1	1	4
2	1	2	1
3	2	1	1
4	2	2	6
5	2	3	6
6	2	4	5
7	2	5	4
8	3	1	2
9	3	2	5
10	3	3	5
11	3	4	1

Express intuitively simulations using **Rel abstractions**



# Simulation: Reason Across Datasets

Experiments can be viewed as possible worlds of a probabilistic database.

Now we can reason across possible worlds.

- Variance of the mean in each dataset

```
def output = sample_stddev[mean[data[i]] for i]
```



#	Float64
1	1.096006394326828

- Mean of the sum of each dataset

```
def output = mean[sum[data[i]] for i]
```



#	Float64
1	12.202

- ...



Observations lead to rejections of incompatible worlds



# The Future of Intelligent Databases

## Overview



- Predictive Reasoning
- Use case:  
Battleship engine
- Mathematical  
Optimization
- Relational ML

# Battleship - Motivation

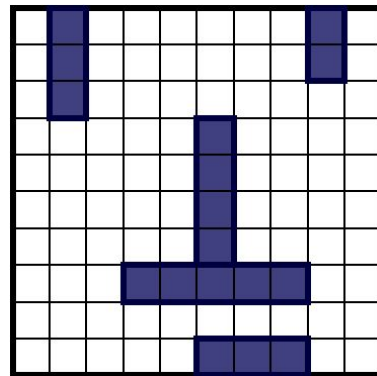
Logic of games are great test beds  
for the expressiveness and capabilities of a system



# Battleship - Setup & Rules

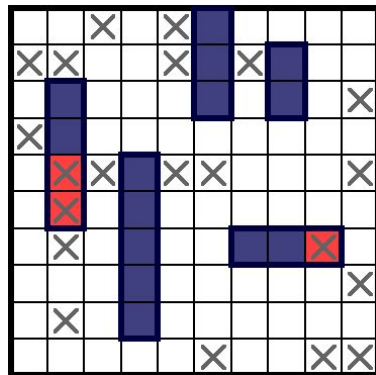
## Goal

- Be the first to find/sink all (5) hidden ships of your opponent



## Rules

- Each player takes a turn and guesses the location of a ship
- Ships can't overlap



# Battleship - A Valid Board

## Analyzing the board

- Higher-order Logic enables elegant and generic formulations
- Abstraction leads to concise code blocks

```
def ships_non_overlapping(SHIPS) = {  
  function({x, y, id: SHIPS(id, x, y)})  
    and not empty(SHIPS)  
}
```

branching the seed

```
def get_random_board[SHIPSIZE, seed] = {  
  random_ship[SHIPSIZE[i], threefry[seed, i]]  
  for i  
}
```

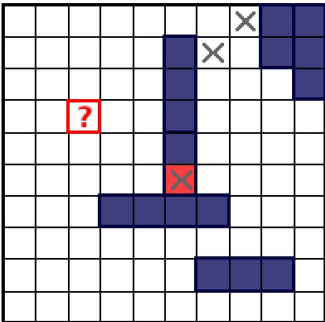
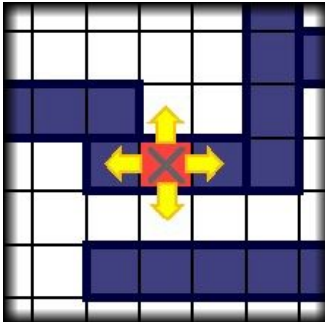
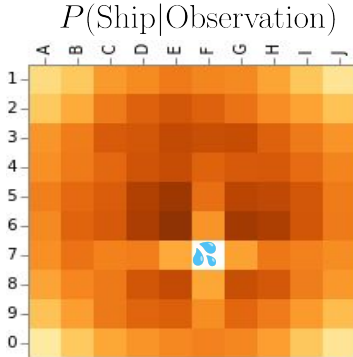
## Initializing board

- Non-monotonic recursion
  - Iterate until a valid configuration
- Module syntax helps to organize program

```
module initial_ship_configuration  
  // start  
  def key = threefry[PARAMETER:SEED, 2], empty(ships)  
  // finish  
  def key = key, ships_non_overlapping(ships)  
  // main recursion (triggered by rejection)  
  def key = threefry[key, 1], not ships_non_overlapping(ships)  
  // ship configuration for current PRNG state  
  def ships = get_random_board[PARAMETER:SHIP_SIZE, key]  
end
```



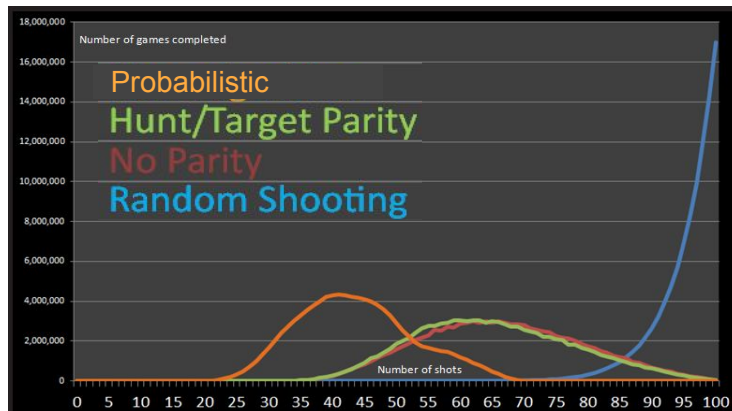
# Battleship - Strategies

Random	Hunt/Target	Probabilistic
<ul style="list-style-type: none"> <li>Randomly guess fields</li> <li>Ignore past observations</li> </ul>  <p>N = 4 Hits = 1</p>	<ul style="list-style-type: none"> <li>Random guesses until a hit occurs</li> <li>Target the discovered ship</li> </ul> 	<ul style="list-style-type: none"> <li>Analyze the probability of finding a ship on each field</li> </ul> 

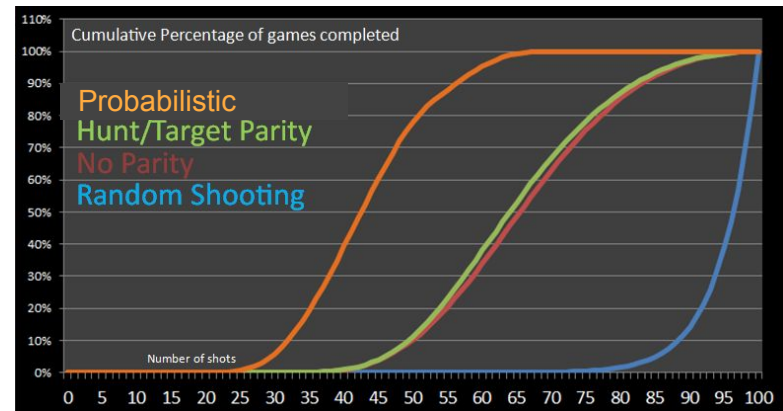
# Battleship - Compare Strategies

How successful is each strategy to discover all ships?

Probability density (PDF)



Cumulative distribution (CDF)



<https://datagenetics.com/blog/december32011/index.html>

- Random: Almost all fields (95 of 100) need to be tested
- Hunt/Target: Around 65 (of 100) guesses are needed
- Probabilistic: Around 40 (of 100) guesses are needed

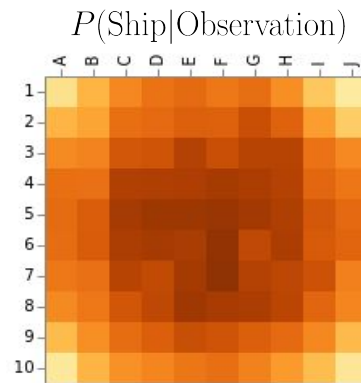




# Battleship - Probabilistic Strategy

## Goal

- Find the field with the highest probability of having a ship given our past observations



## What we need to do

- Valid ship configurations

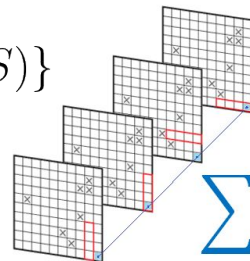
$$C_{\text{valid}} = \{S \in C \mid \text{ships\_dont\_overlap}(S) \wedge \text{consistent\_with\_observation}(S)\}$$

- Calculate the conditional probability for field location (x,y)  
 $P(x,y \mid \text{Observations})$

$$P(x, y) = \frac{\sum_{S \in C_{\text{valid}}} \sum_{s \in S} \mathbf{1}_s(x, y)}{|C_{\text{valid}}|}$$

- Highest Probability

$$\text{argmax}_{x,y} [P(x, y)]$$



# Battleship - Probabilistic Strategy in Rel

## Valid ship configurations

$$C_{\text{valid}} = \{S \in C \mid \text{ships\_dont\_overlap}(S) \wedge \text{consistent\_with\_observation}(S)\}$$

## Likelihood

$$P(x, y|O) = \frac{\sum_{S \in C_{\text{valid}}} \sum_{s \in S} \mathbf{1}_s(x, y)}{|C_{\text{valid}}|}$$

## MLE

$$\text{argmax}_{x,y}[P(x, y)]$$

```
def _valid_sample(sample) = {  
    ships_non_overlapping(_ship[sample])  
    and ships_consistent_with_observation(  
        _ship[sample],  
        analyze_guess[Game])}
```

```
// probability distribution of a ship location  
def ship_distribution[x, y] =  
    count[sample in _valid_sample:  
        | _ship(sample, _, x, y)  
    ]  
    / count[_valid_sample]
```

```
// find the best next guess(es)  
def _next_guess = argmax[x, y, c :  
    ship_distribution(x, y, c)  
    and not get_guess[Game](x, y)  
]
```

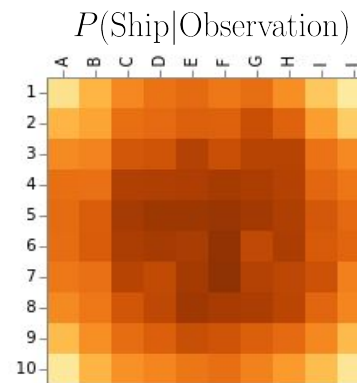


# Battleship - Let's Play

**Round 1:** ~61% of 10k samples are rejected

**Board**

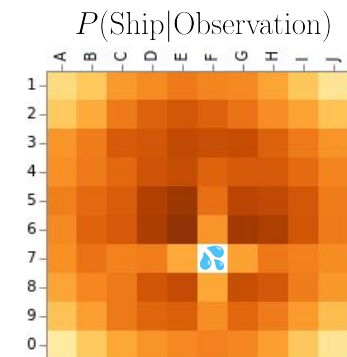
	A	B	C	D	E	F	G	H	I	J
1										
2										
3										
4										
5										
6										
7										
8										
9										
10										



**Round 2:** ~70% of 10k samples are rejected

**Board**

	A	B	C	D	E	F	G	H	I	J
1										
2										
3										
4										
5										
6										
7						✖				
8										
9										
10										

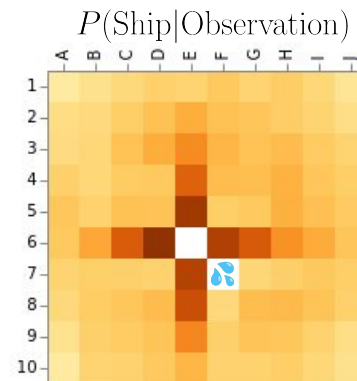


# Battleship - Let's Play

**Round 3:** ~93.3% of 10k samples are rejected

**Board**

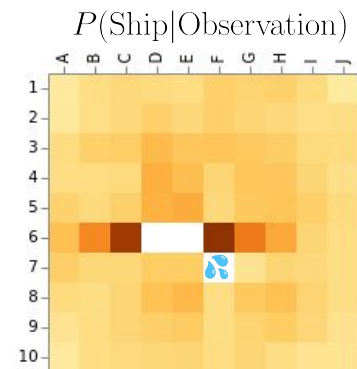
	A	B	C	D	E	F	G	H	I	J
1										
2										
3										
4										
5										
6					✳					
7						🚢				
8										
9										
10										



**Round 4:** ~97% of 10k samples are rejected

**Board**

	A	B	C	D	E	F	G	H	I	J
1										
2										
3										
4										
5										
6				✳	✳					
7						🚢				
8										
9										
10										



# Sampling Strategies

## Expressing various rejection approaches

- Naive Monte Carlo

- Sampling doesn't scale

- Sequential Monte Carlo

- Reject as soon as possible

- Gibbs Sampling, ...

Naive MC	Sequential MC
<pre>def MultipleDice(id, die, value) {   MultipleDice(id, die-1, _)   and die &lt;= n_dice   and value = random_int64[6][id, die] }  def MultipleDice_final(id, die, value) {   MultipleDice(id, die, value)   and   rejection_condition(MultipleDice[id]) }  def rejection_condition( R ) {   forall(die where R(die, _) :     R[die]%2 != 0) }</pre>	<pre>def MultipleDice(id, die, value) {   MultipleDice(id, die-1, _)   and die &lt;= n_dice   and value = random_int64[6][id, die]   and sequential_condition(value) }  def MultipleDice_final(id, die, value) {   MultipleDice(id, die, value)   and   MultipleDice(id, n_dice, _) }  def sequential_condition(x) {x%2 != 0}</pre>

Future: Develop Rel-native High-level Abstractions



# Benefits of Simulations in a Database



## Data Generation

Generate as much synthetic data as needed.

- Naturally handling out-of-core and out-of-disk data

DB ideal to manage (synthetic) data

- Materialize data as needed
- Incrementally maintained data



## Predictive Reasoning

Analyze possible what-if scenarios.

- Use synthetic data to train better ML models.

Optimize data generation and data analytics together in one place.

- Minimize data transfer
- Exploit data independence



# The Future of Intelligent Databases

## Overview



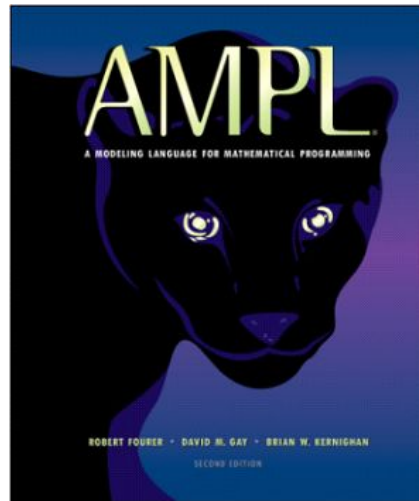
- Predictive Reasoning
- Use case:  
Battleship engine
- Mathematical  
Optimization
- Relational ML

# Optimization



GUROBI  
OPTIMIZATION

FICO® Xpress  
Optimization



## Unconstrained Optimization

- Objective: the error/loss function
- Solver: differentiable function, often gradient descent
- All solutions are acceptable

## Constrained optimization

- Objective: minimize or maximize the function
- Solver: LP, ILP, MIP etc
- Not all solutions are acceptable: constraints
- Mathematical optimization problems are specified in high-level math expressions (AMPL, JuMP). The problems are easily written in Rel

IBM  
CPLEX





# Model for Manufacturing Problem

Given:  $P$ , a set of products  
 $a_j$  = tons per hour of product  $j$ , for each  $j \in P$   
 $b$  = hours available at the mill  
 $c_j$  = profit per ton of product  $j$ , for each  $j \in P$   
 $u_j$  = maximum tons of product  $j$ , for each  $j \in P$

Define variables:  $X_j$  = tons of product  $j$  to be made, for each  $j \in P$

Maximize:  $\sum_{j \in P} c_j X_j$

Subject to:  $\sum_{j \in P} (1/a_j) X_j \leq b$   
 $0 \leq X_j \leq u_j$ , for each  $j \in P$

```
@variable(model, make[products])
@objective(model, Max, sum(prod_profit[p] * make[p] for p in products))
@constraint(model, sum(1 / prod_rate[p] * make[p] for p in products) <= 40)
@constraint(model, [p in products], 0 <= make[p] <= prod_max[p])
```



```
var Make{p in PROD}
maximize Profit: sum{p in PROD} prod_profit[p] * Make[p];
subject to Time: sum{p in PROD} (1 / prod_rate[p]) * Make[p] <= 40;
subject to Limit{p in PROD}: 0 <= Make[p] <= prod_max[p]
```



# Relational Model

Rel supports expressing the objective function and constraints.

The system grounds the constraint in the database and pass the problem to a solver (eg CPLEX, Gurobi, Xpress)

```
def total_profit =  
    sum[prod_profit[p] * make[p] for p in products]  
  
def time_avail() =  
    sum[(1 / prod_rate[p]) * make[p] for p in products] ≤ avail  
  
def demand_market() =  
    forall(p in products: make[p] ≤ prod_market[p])
```

Optimization happens in the dependency graph, so inputs to the solver can be computed  
Rel definitions or even other optimization problems.



# The Future of Intelligent Databases

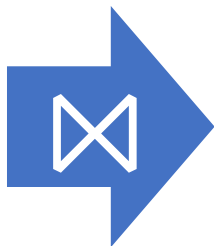
## Overview



- Predictive Reasoning
- Use case:  
Battleship engine
- Mathematical  
Optimization
- **Relational ML**

## 152

The Great Chain of Being is a complex, dense flowchart illustrating the hierarchy of the universe, from the divine to the material. The chart is organized into columns and rows, with various entities and concepts connected by lines. Key figures like Plato, Aristotle, and Ptolemy are mentioned, along with various celestial bodies and elements. The chart is a detailed representation of the medieval cosmological and philosophical system.



Step 1: **throw away all** the structure and knowledge on the data set (eg dependencies).

[illegible]

# PyTorch



sku	store	date	<b>sold</b>
1	S1	2022-03-26	5
1	S1	2022-03-27	7
1	S1	2022-03-28	3

sku	color	price
1	Red	\$5.14

date	temp
2022-03-26	53

store	city	size
S1	Seattle	4000 sqft

city	state
Seattle	WA



sku	store	date	<b>sold</b>	color	price	city	size	state	temp
1	S1	2022-03-26	5	Red	\$5.14	Seattle	4000 sqft	WA	53
1	S1	2022-03-27	7	Red	\$5.14	Seattle	4000 sqft	WA	53
1	S1	2022-03-28	3	Red	\$5.14	Seattle	4000 sqft	WA	53



sku	store	date	<b>sold</b>	<b>Red</b>	<b>Green</b>	price	<b>Seattle</b>	<b>San Diego</b>	size	WA	CA	temp
1	S1	2022-03-26	5	1	0	\$5.14	1	0	4000 sqft	1	0	53
1	S1	2022-03-27	7	1	0	\$5.14	1	0	4000 sqft	1	0	53

# Relational Modelling for Machine Learning

With our research network we have developed training methods that do not require creating a design matrix of features and **operate directly on the relational structure**.

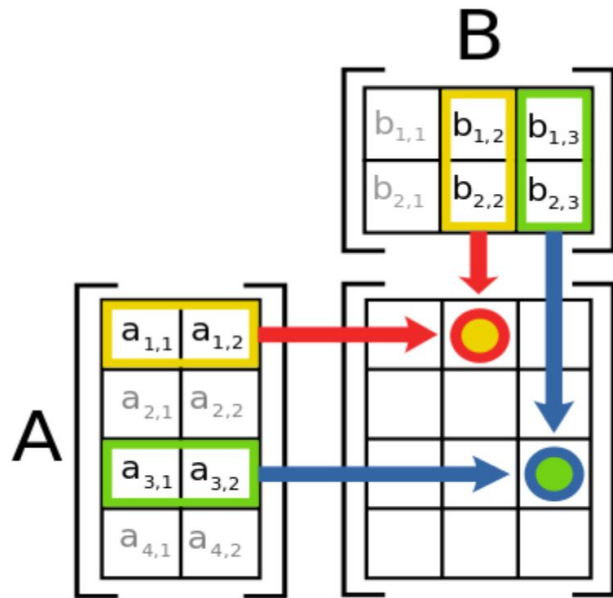
## Key innovations

- **Rel language** - concisely expressing generic machine learning models
- **Automatic differentiation** - relational cost function
- **Semantic optimizer** - exploit relational structure and independence
- **Optimization method** - executed iteratively in RAI system



# Tensors as Relations: Matrix Multiplication

[Deep Learning with Relations at NeurIPS](#)



## Math

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

**Rel** *Our new relational language*

```
def C[i, j] = sum[k: A[i, k] * B[k, j]]
```

## SQL

```
SELECT A.row, B.col, SUM(A.val * B.val)
FROM A INNER JOIN B ON A.col = B.row
GROUP BY A.row, B.col
```



# Rel - Math for Linear Regression

## Generic models

*This is in a reusable library. Note this uses Rel schema abstraction (features is schema)*

```
def predict_linear[X, M][k...] =  
  sum[f: M[f] * X[f, k...]] + sum[f: M[f, X[f, k...]]] + M[:bias]  
  
def linear_regression[X, Y, M] =  
  minimize[rmse[predict_linear[X, M], Y]]
```

## Application-specific instantiation

```
def features[:gdp_per_capita] = ...  
def response = life_satisfaction  
def model = linear_regression[features, response, initial_point]
```





# Relational Machine Learning: Resources and Influences

- **A Layered Aggregate Engine for Analytics Workloads**  
Schleich, Olteanu, Khamis, Ngo, Nguyen, SIGMOD 2019
- **Learning Models over Relational Data Using Sparse Tensors and Functional Dependencies**  
Khamis, Ngo, Nguyen, Olteanu, Schleich, PODS 2018, TODS 2020
- **The Relational Data Borg is Learning**  
Olteanu, VLDB 2020 Keynote (youtube recording: [/watch?v=0ic0jMiOpM0](#), [/watch?v=kWm-0BnbEoU](#))
- **Structure-Aware Machine Learning over Multi-Relational Databases**  
Schleich, PhD thesis, Honorable mention for the 2021 SIGMOD Jim Gray Doctoral Dissertation Award
- **Relational Knowledge Graphs as the Foundation for Artificial Intelligence**  
Aref (youtube recording: [/watch?v=VpyGbjUzG7Y](#))
- **Rk-means: Fast Clustering for Relational Data**  
Curtin, Moseley, Ngo, Nguyen, Olteanu, Schleich, AISTATS 2020



# That Was a Lot – Let's Summary

1. A cloud-native relational database platform
2. designed for structured and semi-structured dynamic data
3. with data and application logic together in one place
4. Example: Intelligent data application with Rel
5. The future of intelligent databases:
  - Probabilistic modeling, optimization, relational ML





# Thank You!