

TypeDB Fundamentals Lecture Series

Type Theory as the Unifying Foundation for Modern Databases



Dr. Christoph Dorn

Head of Research, Vaticle

Previously: Theoretical Computer Scientist in Category Theory
@ Oxford University

The status quo

Many paradigms in use: relational, graph, document, triplestore and RDF, etc.

And many commonly known problems:

- Mismatch of conceptual and logical model
 - Object-relational mismatch, reification, multi-valued attributes, etc.
 - Lack of support for polymorphic and highly connected data

```
DO $$
DECLARE
    inserted_user_id INTEGER;
    inserted_resource_id INTEGER;
    inserted_action_id INTEGER;
    inserted_requestee_id INTEGER;
    inserted_permission_id INTEGER;
BEGIN
    INSERT INTO users (id, full_name)
    VALUES (DEFAULT, 'John Doe')
    RETURNING id INTO inserted_user_id;

    INSERT INTO user_emails (user_id, email, is_primary)
    VALUES
        (inserted_user_id, 'john.doe@vaticle.com', TRUE),
        (inserted_user_id, 'j.doe@vaticle.com', FALSE),
        (inserted_user_id, 'john@vaticle.com', FALSE);

    INSERT INTO employees (user_id, employee_id)
    VALUES (inserted_user_id, 183);

    INSERT INTO full_time_employees (user_id)
    VALUES (inserted_user_id);

    INSERT INTO resources (id)
    VALUES (DEFAULT)
    RETURNING id INTO inserted_resource_id;

    INSERT INTO files (inserted_resource_id, path)
    VALUES (inserted_resource_id, '/home/johndoe/repos/typedb/readme.md');

    INSERT INTO actions (id, name)
    VALUES (DEFAULT, 'edit file')
    RETURNING id INTO inserted_action_id;

    INSERT INTO users (id, full_name)
    VALUES (DEFAULT, NULL)
    RETURNING id INTO inserted_requestee_id;

    INSERT INTO user_emails (user_id, email, is_primary)
    VALUES (inserted_requestee_id, 'kevin@vaticle.com', FALSE);

    INSERT INTO permissions (id, subject, object, action)
    VALUES (DEFAULT, inserted_user_id, inserted_resource_id, inserted_action_id)
    RETURNING id INTO inserted_permission_id;

    INSERT INTO change_requests (id, target, requestee, requested_change)
    VALUES (DEFAULT, inserted_permission_id, inserted_requestee_id, 'revoke');

    COMMIT;
END $$;
```

Distributing fields of
a single object across
multiple tables

The status quo

Many paradigms in use: relational, graph, document, triplestore and RDF, etc.

And many commonly known problems:

- Mismatch of conceptual and logical model
 - Object-relational mismatch, reification, multi-valued attributes, etc.
 - Lack of support for polymorphic and highly connected data
- No easy system extensibility and maintainability
 - Imperative, long, complex, and brittle queries
 - No facility for composable, generic queries that are highly reusable

```
db.groups.aggregate( [
  { "$addFields": { "ownership_type": "group_ownership" } },
  { "$unionWith": {
    "coll": "resources",
    "pipeline": [ { "$addFields": { "ownership_type": "resource_ownership" } } ]
  } },
  { "$lookup": {
    "from": "users",
    "localField": "owner",
    "foreignField": "_id",
    "as": "user_owners"
  } },
  { "$lookup": {
    "from": "groups",
    "localField": "owner",
    "foreignField": "_id",
    "as": "user_group_owners"
  } },
  { "$addFields": { "owners": { "$concatArrays": [ "user_owners", "user_group_owners" ] } } },
  { "$unwind": "$owners" },
  { "$addFields": {
    "owned_type": { "$switch": { "branches": [
      { "case": { "$eq": [ "$group_type", "user_group" ] }, "then": "user_group" },
      { "case": { "$eq": [ "$resource_type", "file" ] }, "then": "file" }
    ] } }
  } },
  { "$addFields": {
    "owned_id": { "$switch": { "branches": [
      { "case": { "$eq": [ "$group_type", "user_group" ] }, "then": "$name" },
      { "case": { "$eq": [ "$resource_type", "file" ] }, "then": "$path" }
    ] } }
  } },
  { "$addFields": {
    "owner_type": { "$switch": { "branches": [
      { "case": { "$eq": [ "$owners.user_type", "user" ] }, "then": "user" },
      { "case": { "$eq": [ "$owners.user_type", "admin" ] }, "then": "admin" },
      { "case": { "$eq": [ "$owners.group_type", "user_group" ] }, "then": "user_group" }
    ] } }
  } },
  { "$project": {
    "_id": false,
    "ownership_type": true,
    "owned_type": true,
    "owned_id": true,
    "owner_type": true,
    "owner_id": { "$switch": { "branches": [
      { "case": { "$eq": [ "$owners.user_type", "user" ] }, "then": "$owners.email" },
      { "case": { "$eq": [ "$owners.user_type", "admin" ] }, "then": "$owners.email" },
      { "case": { "$eq": [ "$owners.group_type", "user_group" ] }, "then": "$owners.name" }
    ] } }
  } }
] )
```

Hard-coded values
in switch cases

The status quo

Many paradigms in use: relational, graph, document, triplestore and RDF, etc.

And many commonly known problems:

- Mismatch of conceptual and logical model
 - Object-relational mismatch, reification, multi-valued attributes, etc.
 - Lack of support for polymorphic and highly connected data
- No easy system extensibility and maintainability
 - Imperative, long, complex, and brittle queries
 - No facility for composable, generic queries that are highly reusable
- Semantic data integrity is easily violated
 - No meaningful data validation due to no sufficiently expressive schemas
 - Data redundancies need to be carefully synced

```
MATCH
  (john:User),
  (readme:File {path: "/home/johndoe/repos/typedb/readme.md"}),
  (edit:Action {name: "edit file"})
WHERE (
  john.primary_email = "john@vaticle.com"
  OR "john@vaticle.com" IN john.alias_emails
) AND NOT EXISTS {
  MATCH (john)-[:SUBJECT]-(:perm:Permission)-[:OBJECT]->(readme)
  WHERE EXISTS ( (perm)-[:ACTION]->(edit) )
}
WITH john, readme, edit
CREATE (john)-[:SUBJECT]-(:perm:Permission)-[:OBJECT]->(readme)
WITH edit, perm
CREATE (perm)-[:ACTION]->(edit);
MATCH
  (perm:Permission),
  (perm)-[:SUBJECT]->(john:User),
  (perm)-[:OBJECT]->(readme:File {id: "/home/vaticle/repos/typedb/readme.md"}),
  (perm)-[:ACTION]->(edit:Action {name: "edit file"}),
  (kevin:User)
WHERE (
  john.primary_email = "john@vaticle.com"
  OR "john@vaticle.com" IN john.alias_emails
) AND (
  kevin.primary_email = "kevin@vaticle.com"
  OR "kevin@vaticle.com" IN kevin.alias_emails
)
CREATE
  (rqst:ChangeRequest {requested_change: "revoke"}),
  (rqst)-[:TARGET]-(:perm),
  (rsqt)-[:REQUESTEE]-(:kevin);
```

Missing checks on the semantic validity of created relations

Watch our **next lecture!**

For details on these common database pain points:

A lecture poster with a dark space background. At the top, there's a colorful nebula. In the bottom right, the Vaticle TypeDB logo is visible. The text on the poster includes the event title, date and time, speaker name, and a small image of the Earth's horizon.

LECTURE | NOVEMBER 30TH | 5PM GMT | 12PM ET

Why We Need a Polymorphic Database

James Whiteside, Research Engineer at Vaticle

...today we embark on a more theory-driven journey!

Re-thinking foundations

Existing database paradigms

- OK for **domain-specific** applications
- Cannot express **polymorphism**
- Not built for **composable** and **extensible** modern systems
- No native **interoperability**

Inspiration

*how would we re-think the **foundations of modern databases** from first principles?*

Inspiration

"Domain-specific"
mathematics

Modernization

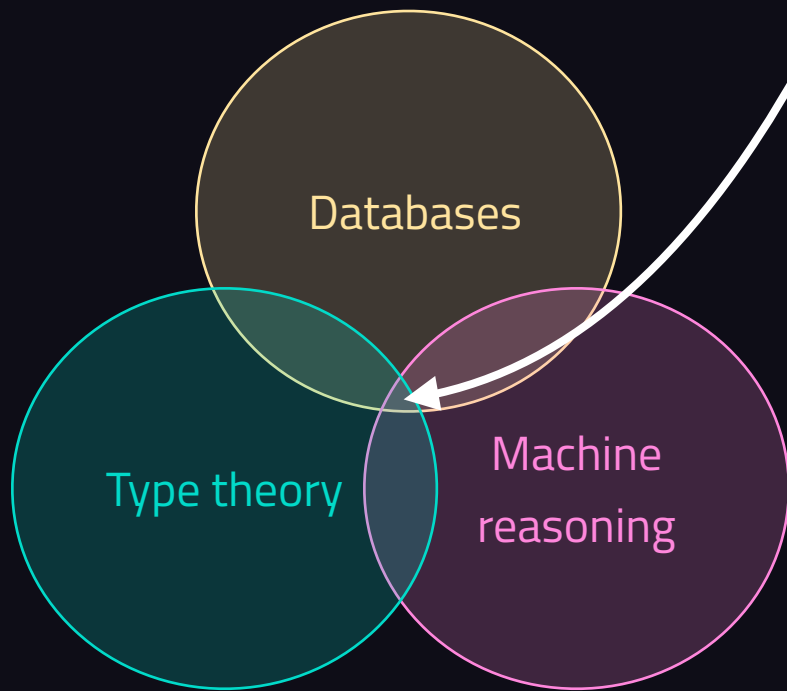
type-theoretic
mathematics

Theory of relations, graphs, trees, etc. ...

General theory of composable structures

The TypeDB approach

The TypeDB ecosystem



- Type-theoretic query language
- Conceptual data model enhanced with polymorphism
- Type inference engine and extensible type schemas
- Machine reasoning engine

not relational
not graph
not document
...but a *unification* thereof!

In this lecture we learn how **type theory** underlies all of the the above!

Part I:
The Modernization of Mathematics

or

What are foundations of general mathematical structures?

The original inspiration for relational algebra

Classical foundations based on *two ingredients*

Sets

$x \in X$

Relations

$\phi(x_1, x_2, \dots, x_n)$

a.k.a. predicates + predicate logic

The **original inspiration** for relational algebra

Classical foundations based on *two ingredients*

Sets

$p \in \text{Person}$

Relations

$\text{Marriage}(p_1, p_2)$

a.k.a. predicates + predicate logic

- Provided inspiration for **relational data model**, Prolog, etc.
- Modeling everything in **sets** and **relations** is **non-practical**

... you need not be a mathematician to know modeling with relations can be restrictive!

Mathematics **moved on** to composable systems

Modern foundations based on *one ingredient*

Types

$p : \text{Person}$

also Types

$m : \text{Marriage}(p_1, p_2)$

a.k.a. dependent type

- **Unifies** all structures as *types*, with powerful repercussions:
 - “Facts” become themselves *data* in types, that can be explicitly *referenced*
 - Dependencies on data can be *composed*
 - De-facto the *practical foundation* of modern mathematics (see [Taylor, '99])

$d : \text{Date-of}(m)$



... Can we build *database foundations* on this, too? Well, types are just “*declarative data descriptions*”!

Part II:
The *type-theoretic* query language

Crash course in **type theory**

So what precisely is a type?

*A **type** is a **description of a domain** that a **variable** can range over.*

Math/PL examples:

- **x** : integer (x in $\{ \dots, 1, 2, 3, \dots \}$)
- **y** : string (y in $\{ "a", "b", "aa", \dots \}$)
- **z** : factor of **x** \rightarrow *dependent type*
(when $x = 26$, z in $\{ 1, 2, 13, 26 \}$)

NL examples:

- **p** : Person
- **n** : Name of person **p**
- **m** : Marriage between persons **p1** and **p2**

Crash course in **type theory**

So what precisely is a type?

A **type** is a **description of a domain** that a **variable** can range over.

Math/PL Examples

Dependent type

Pair type operator

Sum type operator

...

NL Examples

Description with variable

Combined descriptions

Alternative descriptions

...

'X is a **person**, and, Y is a **city**'

"and"

'X is a **person**, and, Y is a **name of X**'

"or"

'X is a **person**, or, X is a **city**'

"dependent pair type"

Type theory makes math composable

dependencies compose

- $p_1, p_2 : \text{Person}$
 - $m : \text{Marriage of } p_1 \text{ and } p_2$
 - $d : \text{Date of } m$
- $d : \text{Date of Marriage } m \text{ of Persons } p_1 \text{ and } p_2$

Dependently Typed Programming

```
 $\Sigma [ p_1 \in \text{Person} ] ($   
 $\Sigma [ p_2 \in \text{Person} ] ($   
 $\Sigma [ m \in \text{Marriage } p_1 \ p_2 ] ($   
 $\Sigma [ d \in \text{Date } m ] \top ))))$ 
```

VS

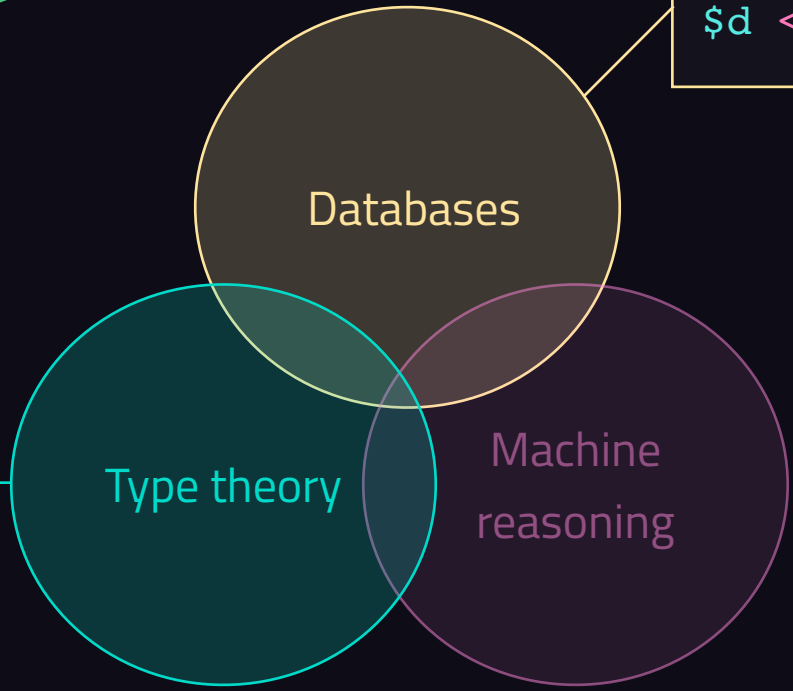
TypeQL pattern

```
 $\$p_1 \text{ isa person};$   
 $\$p_2 \text{ isa person};$   
 $\$m (\$p_1, \$p_2) \text{ isa marriage};$   
 $\$m \text{ has date } \$d;$ 
```

Type-theoretic querying

Composite dependent types are **queries**

```
Σ[ c1 ∈ City ]  
Σ[ c2 ∈ City ]  
Σ[ f ∈ Flight c1 c2 ]  
Σ[ d ∈ Duration f ]  
Σ[ proof ∈ d ≤ 120 ] ⊤
```



```
$c1 isa city;  
$c2 isa city;  
$f (from: $c1, to: $c2) isa flight;  
$f has duration $d;  
$d <= 600; # 10 hours
```

...but TypeQL can do much more than plain type theoretic languages!



Part III:
The type-theoretic *conceptual* data model
*enhanced with *polymorphism**

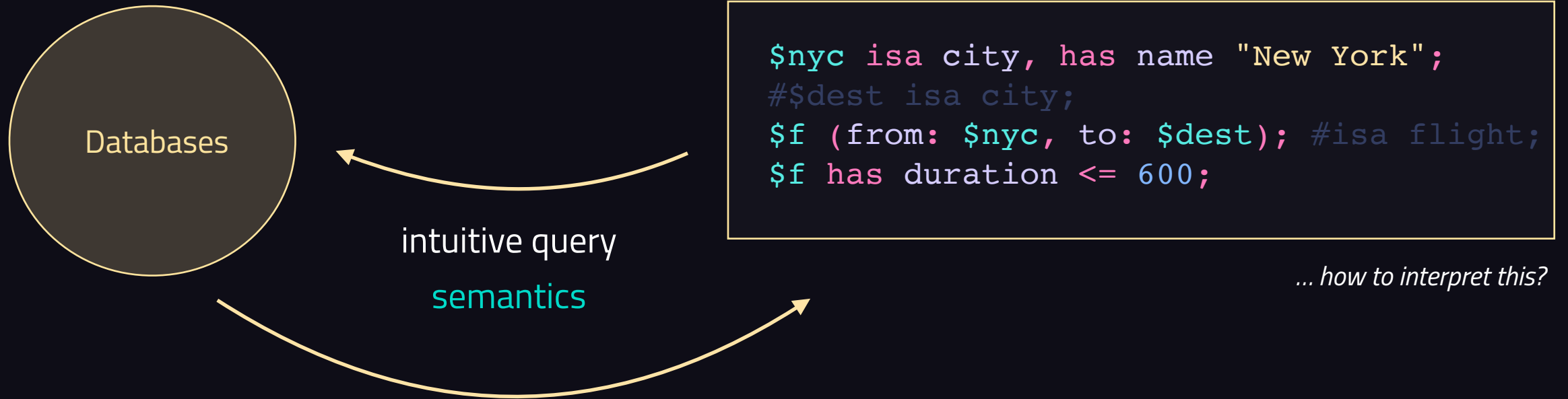
Type-theoretic querying



intuitive query
semantics

```
$nyc isa city, has name "New York";  
$dest isa city;  
$f (from: $nyc, to: $dest) isa flight;  
$f has duration <= 600;
```

Type-theoretic querying



Need: database **schema** giving context for type **inference** and type **validation**

Idea: work with an intuitive conceptual data model!

From **conceptual** schemas to type theory

The intuitive categorization of natural language:

Natural Language	Conceptual Modeling	Type Theory
Nouns <i>a <u>person</u></i>	Entities <i>person</i>	types <i>without</i> dependencies containing objects
Verbs connecting nouns <i>a person <u>marries</u> a person</i>	Relations <i>marriage of p1 and p2</i>	types <i>with</i> dependencies containing objects
Adjective or Adverbs <i>a person marries a person <u>today</u></i>	Attributes <i>date of a marriage m</i>	types <i>with</i> dependencies containing values

TypeDB is based on a **type-theoretic conceptual data model**

Reverse engineering our ERA schema

Type-theoretic query language

```
$nyc isa city, has name "New York";  
$f ( from: $nyc, to: $dest );  
$f has duration <= 600;
```

named *abstractions* of
type dependencies



Type-theoretic conceptual schema

```
city sub entity;  
name sub attribute,  
    value string;  
city owns name;  
transport sub relation,  
    relates from,  
    relates to;  
city plays transport:from,  
    plays transport:to;  
duration sub attribute,  
    value long;  
transport owns duration;
```

Enhancing our model with **type polymorphism**

1. **Inheritance polymorphism** lets types **inherit** the full specification of a **parent type**, enabling the hierarchical organization of types.

```
name sub attribute,  
    value string;  
city sub entity, owns name;  
transport sub relation,  
    relates from,  
    relates to;  
city plays transport:from,  
    plays transport:to;  
  
flight sub transport,  
    owns flight_no;  
flight_no sub attribute,  
    value string;  
#flight inherits specification!
```

The three kinds of **type polymorphism**


1. **Inheritance polymorphism** lets types **inherit** the full specification of a **parent type**, enabling the hierarchical organization of types.
2. **Interface polymorphism abstracts** input types in **dependencies**: input types need specific *capabilities* instead of full type specifications.

```
name sub attribute,  
      value string;  
city sub entity, owns name;  
person sub entity, owns name;  
# .. having a name is  
# a capability!  
  
city plays transport:from,  
      plays transport:to;  
airport sub entity;  
airport plays transport:from,  
          plays transport:to;
```

The three kinds of **type polymorphism**

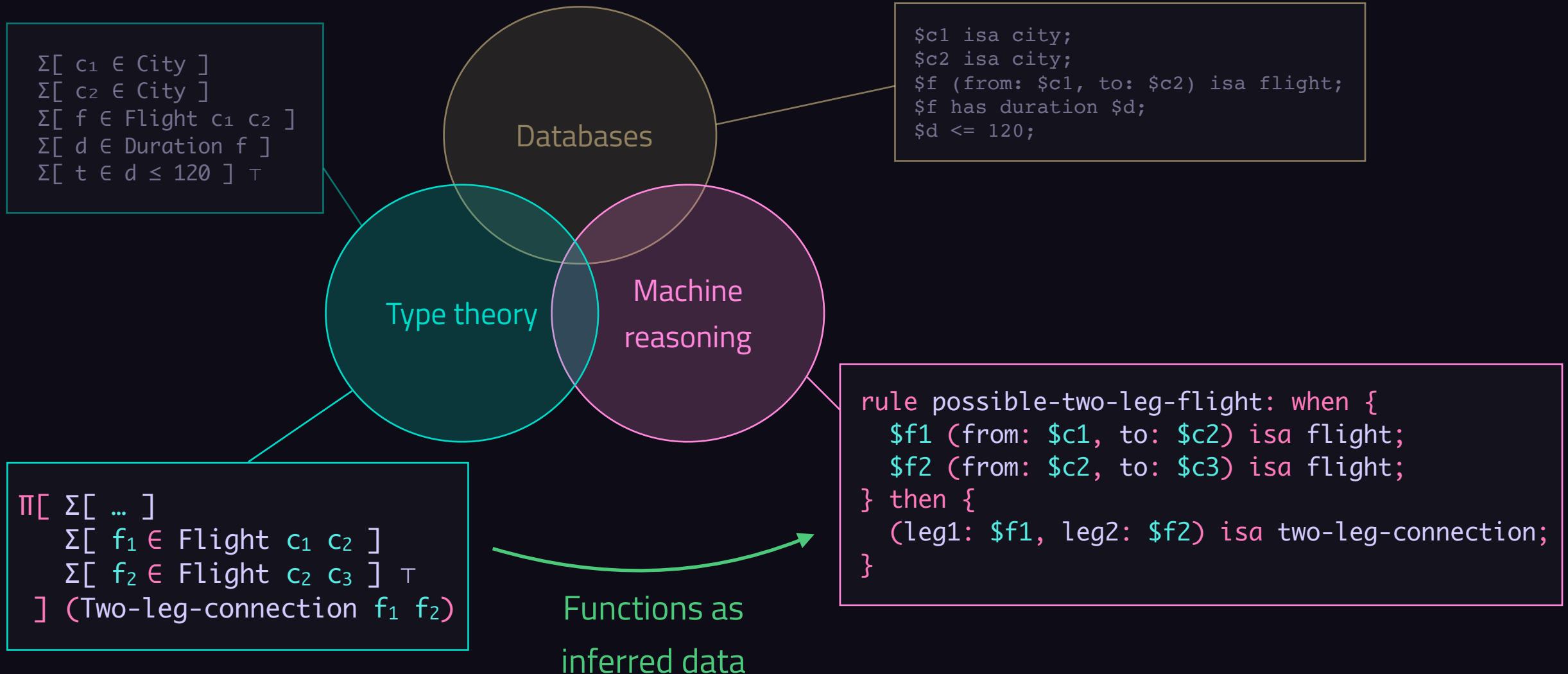
1. **Inheritance polymorphism** lets types **inherit** the full specification of a **parent type**, enabling the hierarchical organization of types.
2. **Interface polymorphism abstracts** input types in **dependencies**: input types need specific *capabilities* instead of full type specifications.
3. **Parametric polymorphism** defines generic functionality for (variabilized) types, enabling semantically generic queries

```
match
  $something owns name;
  $object isa $something;
  $object has name $name;
delete
  $object isa $something;
# 'semantically generic'
# w.r.t to name ownerships
# defined in the schema
```

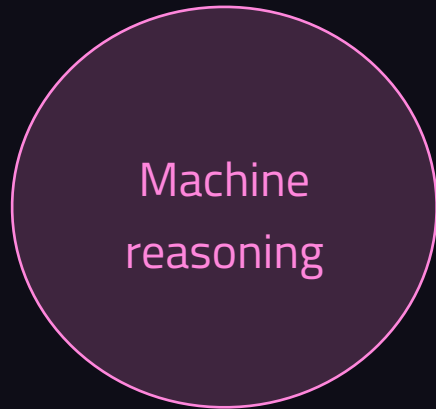



Part IV:
One reasoning engine to rule them all

Type-theoretic reasoning



Type-theoretic reasoning



intuitive rule semantics

```
rule possible-two-leg-flight: when {
  $flight1 (from: $city1, to: $city2) isa flight,
  has arrival $time1;
  $flight2 (from: $city2, to: $city3) isa flight,
  has departure $time2;
  $time2 > $time1 + 30; # allow for 30 min
} then {
  (leg1: $flight1, leg2: $flight2)
  isa two-leg-connection;
}
```



Need: Reasoning engine that evaluates rules at query time
Gain: Fine-grained control of source data and application logic

Source	Logic
Tables	Views
Nodes + Edges	Paths
Collections	Aggregates
...	...



Part V.

The result: a unifying foundation for modern databases

The **unification** of existing paradigms

- Migrating from **relational**
 - Tables \rightsquigarrow Entities
(*associative* Entities \rightsquigarrow Relations)
 - Foreign keys \rightsquigarrow Relations
 - Columns \rightsquigarrow Attributes

Relational flattens
dependencies!

```
# Table 'student'
student sub entity,
  owns subject-of-study,
  owns start-date,
  plays supervision:supervisee;

# Table 'professor'
professor sub entity,
  owns name,
  owns number-of-students,
  owns taking-new-PhDs,
  plays supervision:supervisor;

# Foreign key for 'student' in 'prof.'
supervision sub relation,
  relates supervisee,
  relates supervisor;
```

Type theory generalizes existing paradigms

■ Migrating from relational

- Tables \rightsquigarrow Entities
(*associative* Entities \rightsquigarrow Relations)
- Foreign keys \rightsquigarrow Relations
- Columns \rightsquigarrow Attributes

Relational flattens dependencies!

■ Migrating from property graph

- Node Types \rightsquigarrow Entities
- Edge Types \rightsquigarrow Relations
- Properties \rightsquigarrow Attributes
- Path composition \rightsquigarrow Logic

Graph has little control of logic!

■ Document, RDF (with reification), and more ...

```
# 'City' node with properties
city sub entity,
  owns name;

# 'Flight' edges with properties
flight sub relation,
  relates departure-city,
  relates arrival-city,
  owns departure-time,
  owns arrival-time;

# Path data types
two-leg-flight sub relation,
  relates first-leg;
  relates second-leg;
flight plays two-leg-flight:first-leg,
  plays two-leg-flight:second-leg;
```

The new status

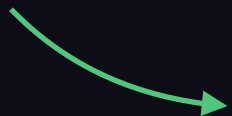
- **Mismatch of conceptual and logical model**
 - Object-relational mismatch, reification, multi-valued attributes, etc.
 - Lack of support for polymorphic and highly connected data
- **No easy system extensibility and maintainability**
 - Imperative, long, complex, and brittle queries
 - No facility for composable, generic queries that are highly reusable
- **Semantic data integrity is easily violated**
 - No meaningful data validation due to no sufficiently expressive schemas
 - Data redundancies need to be carefully synced



Solved by conceptual data model with type polymorphism



Solved by type-theoretic declarative query language



Solved by type inference and machine reasoning engine

In summary

- **TypeDB** implements the unifying **type-theoretic, polymorphic paradigm**
- This makes TypeDB an **extensible, adaptable, safe** and **robust** DBMS
- It's a **fast evolving** software ecosystem!

What's next

There is much more to talk about, see [upcoming lectures](#):

Thursday, Nov 30th



Vaticle
TypeDB

WEBINAR | NOVEMBER 30TH | 5PM GMT | 12PM ET

Why We Need a Polymorphic Database

James Whiteside, Research Engineer at Vaticle

Thursday, Dec 7th



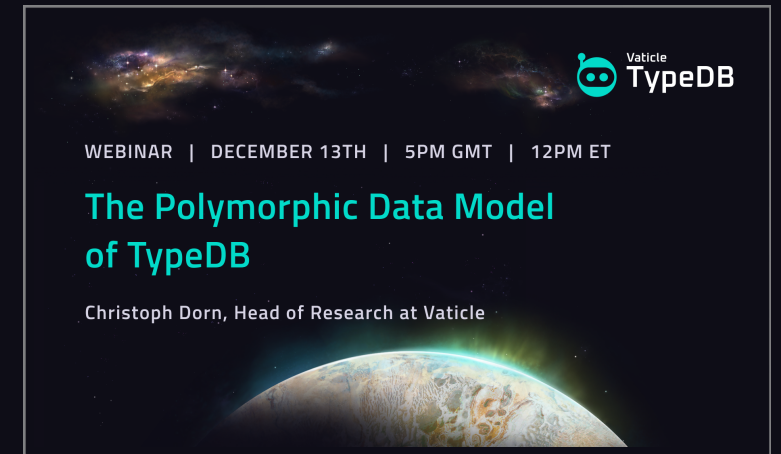
Vaticle
TypeDB

WEBINAR | DECEMBER 7TH | 5PM GMT | 12PM ET

TypeDB: the polymorphic database

James Whiteside, Research Engineer at Vaticle

Wednesday, Dec 13th



Vaticle
TypeDB

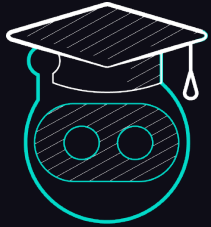
WEBINAR | DECEMBER 13TH | 5PM GMT | 12PM ET

The Polymorphic Data Model of TypeDB

Christoph Dorn, Head of Research at Vaticle

Register at TypeDB.com/lectures

More TypeDB Resources



TypeDB Learning Center - typedb.com/learn



Download TypeDB - typedb.com/deploy



TypeDB Cloud Waitlist - cloud.typedb.com —> Join Waitlist



Vaticle

TypeDB

Thank you!

Join us at typedb.com/discord