TypeDB Fundamentals Lecture Series

Why We Need a Polymorphic Database

Dr. James Whiteside

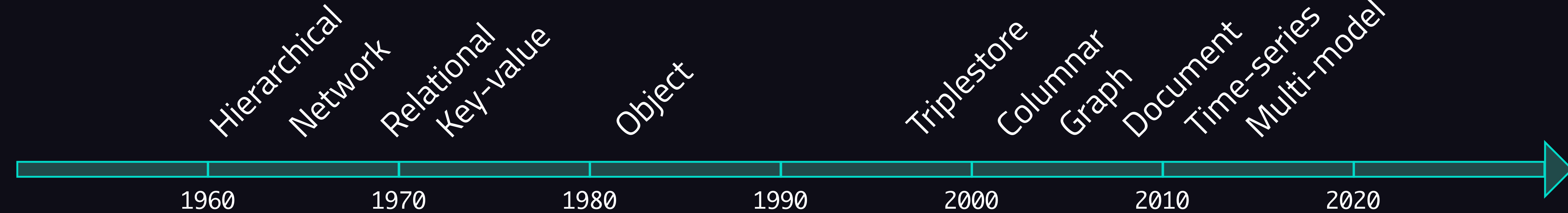
Research Engineer, Vaticle

Previously: Computational Solid-State Physicist
@ University of Surrey

Key issues with contemporary databases

- Object model mismatch
- Semantic integrity
- Polymorphic querying

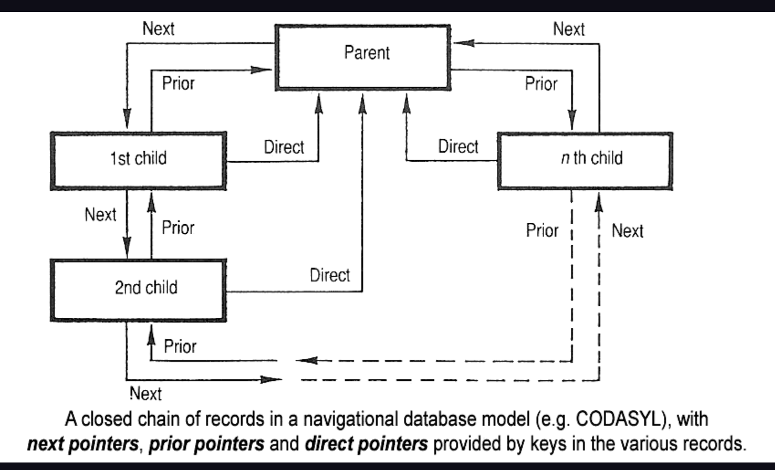
A brief overview of database paradigms



A brief overview of database paradigms

Navigational databases

- Queries are imperative in nature.
- Records are accessed by sequential traversal.
- Records have types determining properties.
- Later versions had schemas.



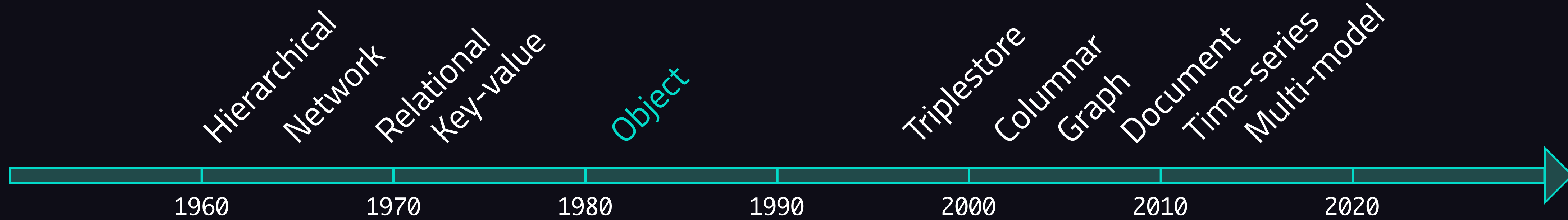
A closed chain of records in a navigational database model (e.g. CODASYL), with ***next pointers***, ***prior pointers*** and ***direct pointers*** provided by keys in the various records.

A brief overview of database paradigms

Relational databases

- Data is retrieved declaratively.
- Backed by set-theoretic algebra (Codd, 1969).
- Schema defines tables and references.
- Not natively compatible with OOP data models.

A brief overview of database paradigms



Object databases

- Store data according to application models.
- Designed for OOP data persistence.
- Schema determined by application language.
- Data is strongly tied to application.

A brief overview of database paradigms

NoSQL databases

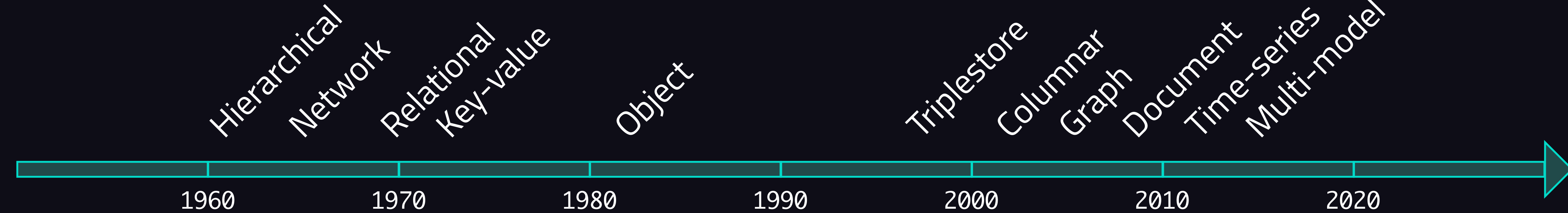
- Popularised by cloud computing.
- Wide variety of logical models.
- Typically schemaless.
- Designed for semi- and un-structured data.

A brief overview of database paradigms

Multi-model databases

- Can store data in disparate logical models.
- Data can be translated and queried across models.
- Allows leveraging of different model strengths.
- Do not introduce any new ideas.

A brief overview of database paradigms



A brief overview of database paradigms

Widespread adoption in commercial software engineering

A brief overview of database paradigms

Widespread adoption in commercial software engineering

Used for high-level application development

A brief overview of database paradigms

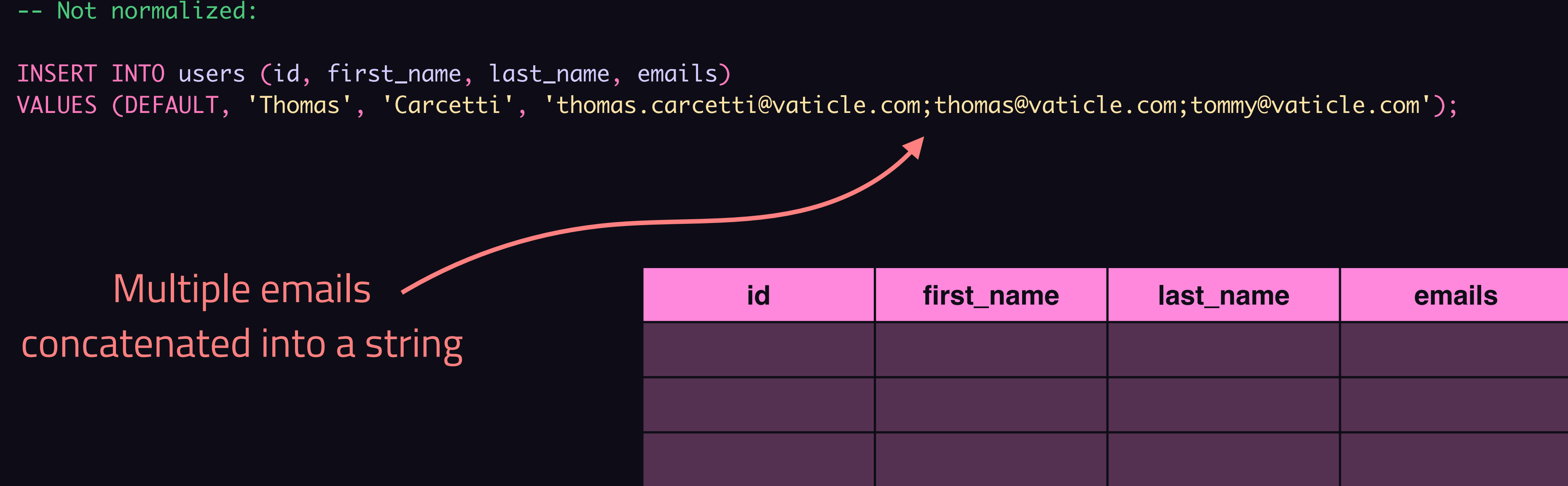
Widespread adoption in commercial software engineering

Used for high-level application development

Object model mismatch

Mismatch in relational, document, and graph databases

Mismatch in the relational paradigm



Mismatch in the relational paradigm

```sql
-- Not normalized:

INSERT INTO users (id, first_name, last_name, emails)
VALUES (DEFAULT, 'Thomas', 'Carcetti', 'thomas.carcetti@vaticle.com;thomas@vaticle.com;tommy@vaticle.com');
```

```sql
-- Normalized:

DO $$
DECLARE
  user_id INT;
BEGIN
  INSERT INTO users (id, first_name, last_name)
  VALUES (DEFAULT, 'Thomas', 'Carcetti')
  RETURNING id INTO inserted_user_id;

  INSERT INTO user_emails (user_id, email)
  VALUES
    (inserted_user_id, 'thomas.carcetti@vaticle.com'),
    (inserted_user_id, 'thomas@vaticle.com'),
    (inserted_user_id, 'tommy@vaticle.com');
END $$;
```

id	first_name	last_name

FK

user_id	email

Multiple rows required to represent atomic object

Mismatch in the relational paradigm

OBJECT-RELATIONAL MISMATCH

```
-- Normalized:

DO $$
DECLARE
  user_id INT;
BEGIN
  INSERT INTO users (id, first_name, last_name)
  VALUES (DEFAULT, 'Thomas', 'Carcetti')
  RETURNING id INTO inserted_user_id;

  INSERT INTO user_emails (user_id, email)
  VALUES
    (inserted_user_id, 'thomas.carcetti@vaticle.com'),
    (inserted_user_id, 'thomas@vaticle.com'),
    (inserted_user_id, 'tommy@vaticle.com');
END $$;
```

id	first_name	last_name

FK

user_id	email

Multiple rows required to represent atomic object

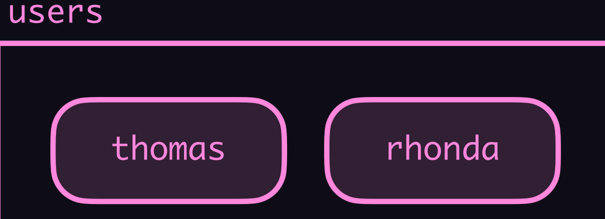
Mismatch in the relational paradigm

OBJECT-RELATIONAL MISMATCH

multi-valued attributes
one-to-many relations
many-to-many relations
inheritance hierarchies
interface structures

Mismatch in the document paradigm

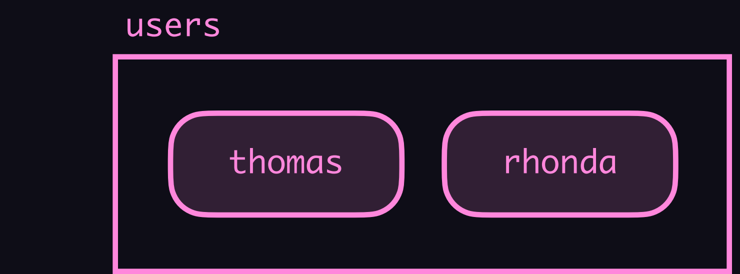
```
db.users.insert( [
  {
    "first_name": "Thomas",
    "last_name": "Carcetti",
    "emails": [
      "thomas.carcetti@vaticle.com",
      "thomas@vaticle.com",
      "tommy@vaticle.com"
    ]
  },
  {
    "first_name": "Rhonda",
    "last_name": "Pearlman",
    "emails": [
      "rhonda.pearlman@vaticle.com",
      "rhonda@vaticle.com"
    ]
  }
] )
```



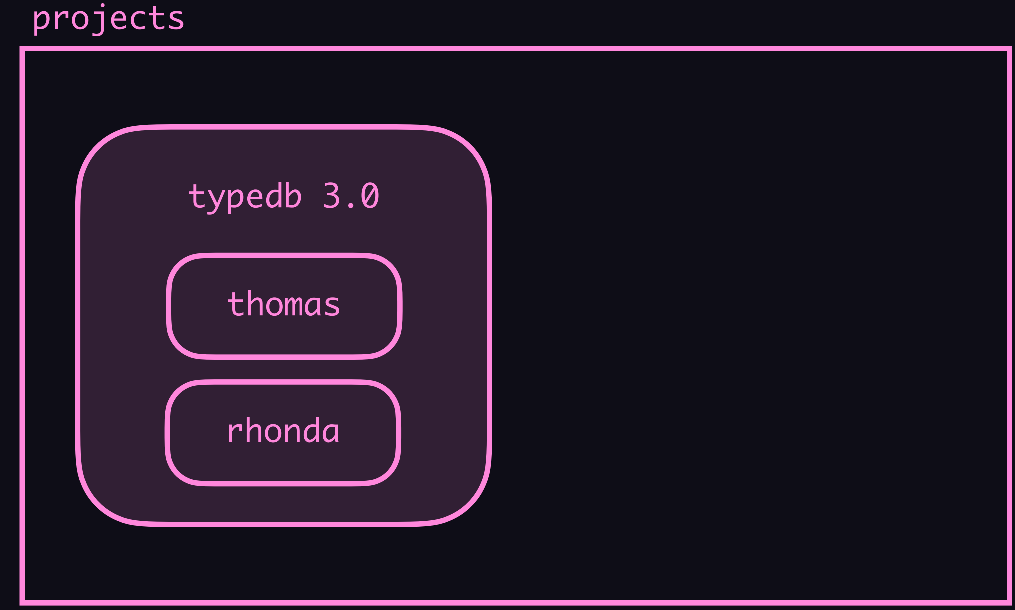
Mismatch in the document paradigm

```
db.projects.insert( [
  {
    "name": "TypeDB 3.0",
    "members": [
      {
        "first_name": "Thomas",
        "last_name": "Carcetti",
        "emails": [
          "thomas.carcetti@vaticle.com",
          "thomas@vaticle.com",
          "tommy@vaticle.com"
        ]
      },
      {
        "first_name": "Rhonda",
        "last_name": "Pearlman",
        "emails": [
          "rhonda.pearlman@vaticle.com",
          "rhonda@vaticle.com"
        ]
      }
    ]
  }
] )
```

users



projects

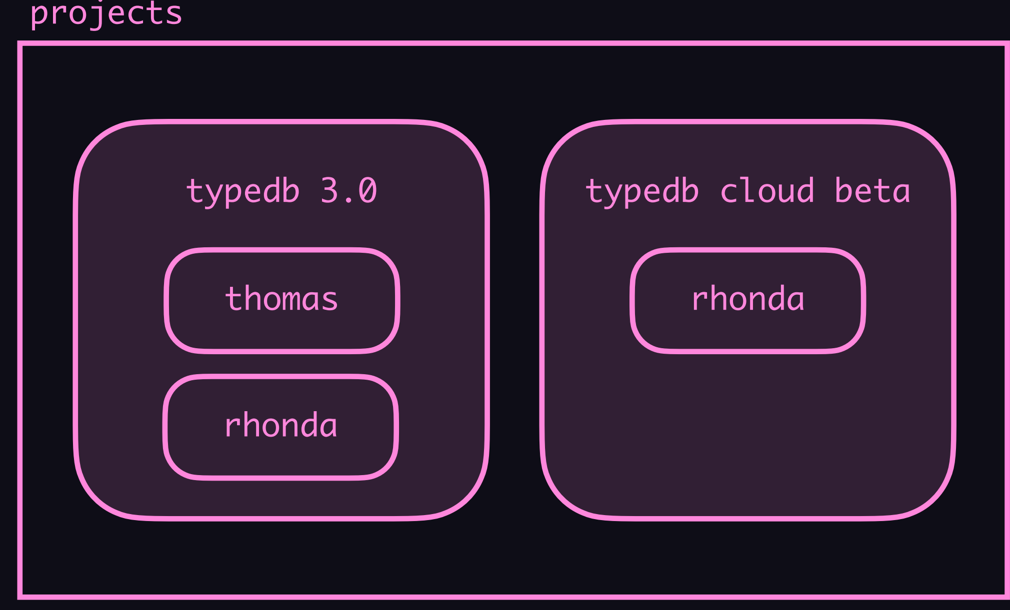


Mismatch in the document paradigm

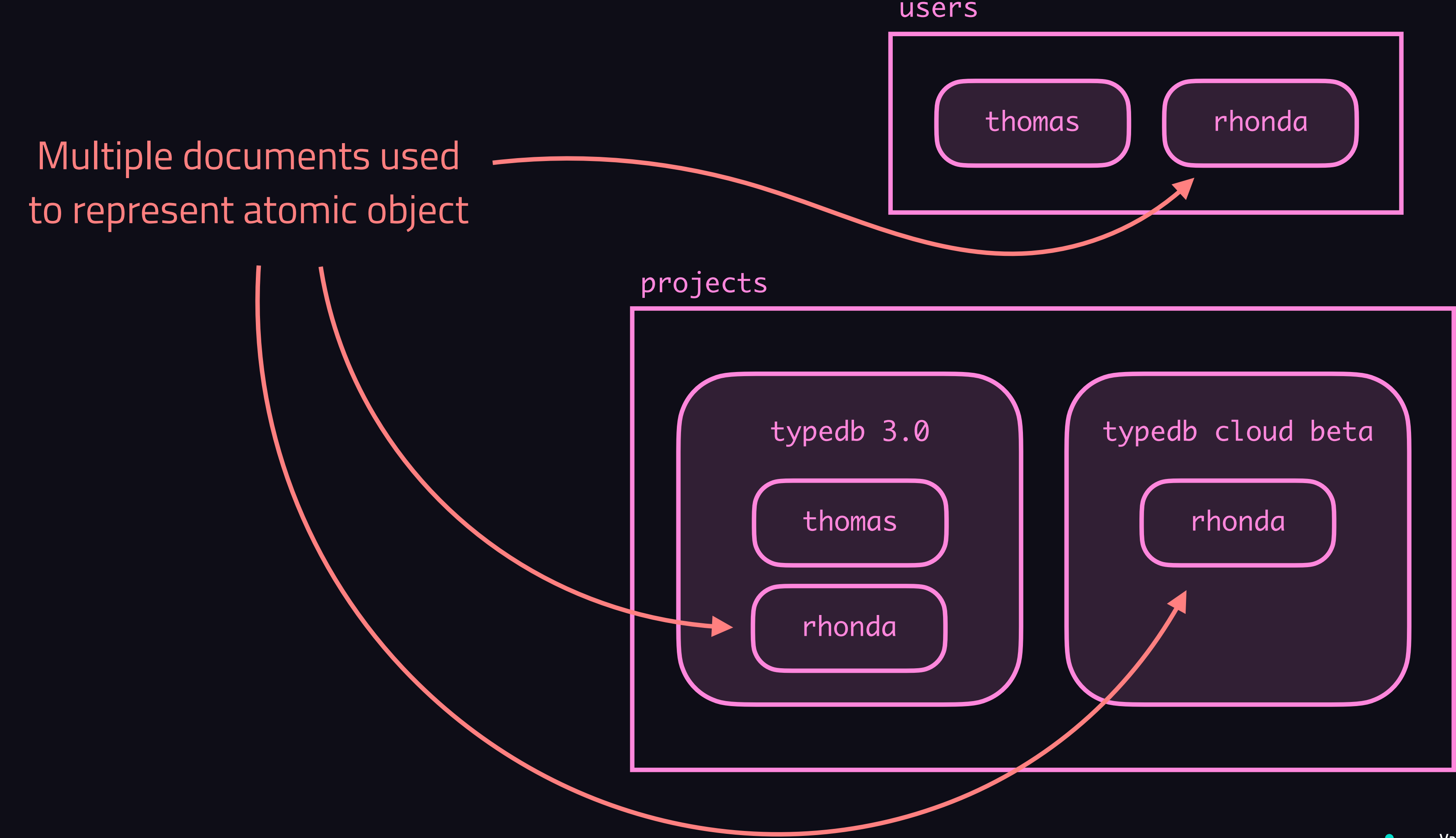
```
db.projects.insert( [
  {
    "name": "TypeDB Cloud beta",
    "members": [
      {
        "first_name": "Rhonda",
        "last_name": "Pearlman",
        "emails": [
          "rhonda.pearlman@vaticle.com",
          "rhonda@vaticle.com"
        ]
      }
    ]
  }
] )
```

users

projects



Mismatch in the document paradigm



Vaticle TypeDB

Mismatch in the document paradigm

```
thomas_id = db.users.find(
  { "emails": "thomas.carcetti@vaticle.com" }
).next()["_id"]

rhonda_id = db.users.find(
  { "emails": "rhonda.pearlman@vaticle.com" }
).next()["_id"]


db.projects.insert( [
  {
    "name": "TypeDB 3.0",
    "members": [ thomas_id, rhonda_id ]
  },
  {
    "name": "TypeDB Cloud beta",
    "members": [ rhonda_id ]
  }
] )
```

users

thomas

rhonda

projects

typedb 3.0

_id of thomas

_id of rhonda

typedb cloud beta

_id of rhonda

Document databases are not optimised to query over references

Mismatch in the document paradigm

users

thomas | rhonda

Multiple documents used to represent atomic object

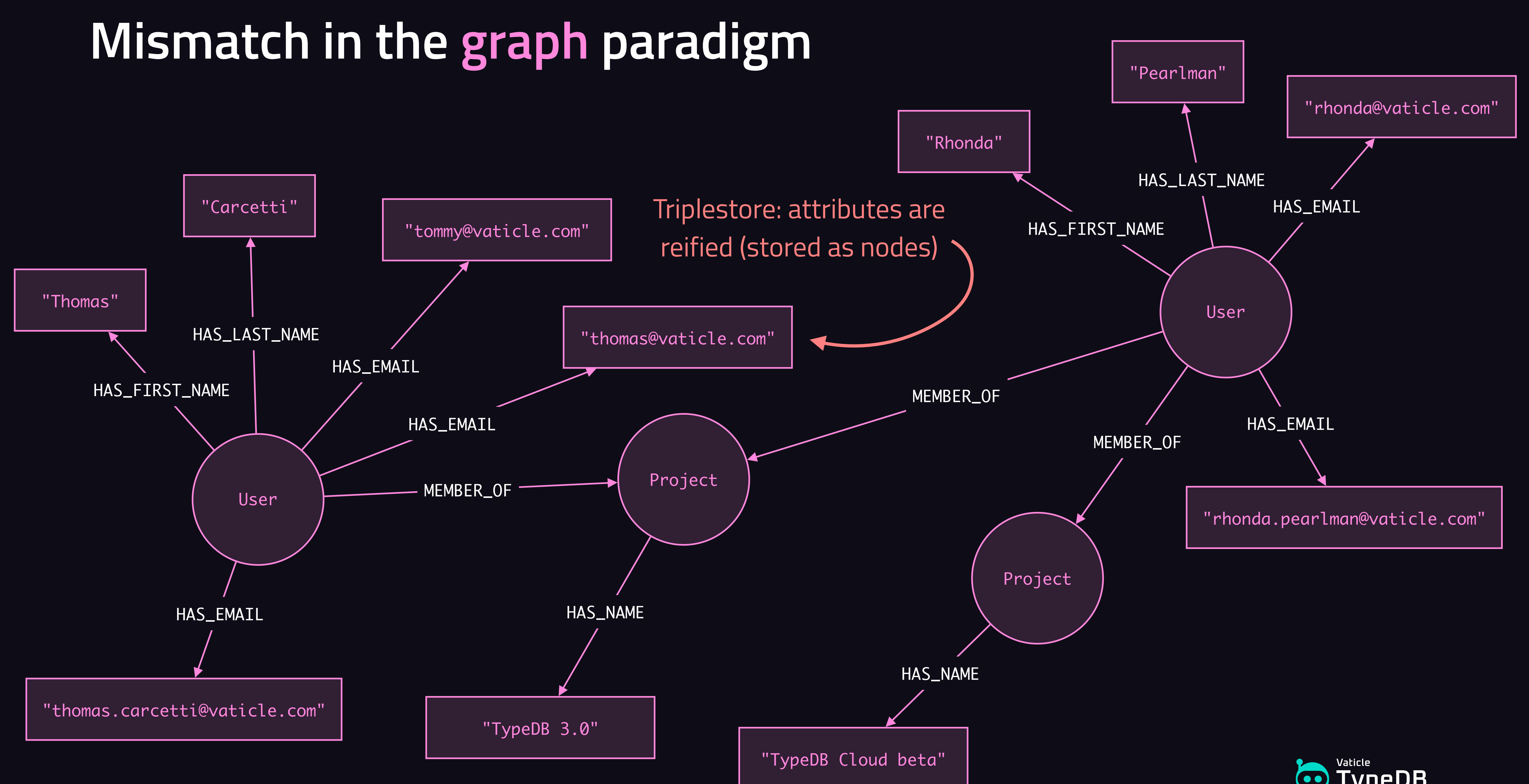
projects

typedb 3.0: thomas, rhonda

typedb cloud beta: rhonda

OBJECT-DOCUMENT MISMATCH

Mismatch in the graph paradigm



Mismatch in the graph paradigm

Graph: attributes are stored as internal structures

```
{
  "first_name": "Thomas",
  "last_name": "Carcetti",
  "emails": [
    "thomas.carcetti@vaticle.com",
    "thomas@vaticle.com",
    "tommy@vaticle.com"
  ]
}
```

User — MEMBER_OF → Project

```
{
  "name": "TypeDB 3.0"
}
```

```
{
  "first_name": "Rhonda",
  "last_name": "Pearlman",
  "emails": [
    "rhonda.pearlman@vaticle.com",
    "rhonda@vaticle.com"
  ]
}
```

User — MEMBER_OF → Project (TypeDB 3.0)

User — MEMBER_OF → Project

```
{
  "name": "TypeDB Cloud beta"
}
```

Mismatch in the graph paradigm

Relationships can only be between two nodes

Admin

ASSIGNED

User

MEMBER_OF

MEMBER_OF

User

MEMBER_OF

Project

Project

Mismatch in the graph paradigm

Relationship must be reified (stored as a node)

Admin
ASSIGNED
Membership
User
HAS
MEMBER_OF
OF
Project
User
MEMBER_OF
Project

Mismatch in the graph paradigm

Multiple nodes and edges used to represent atomic relationship

OBJECT-GRAPH MISMATCH

Mismatch in the graph paradigm

OBJECT-GRAPH MISMATCH

n-ary relations (non-binary)

nested relations (relation endpoints)

undirected relations (two-way)

Summary

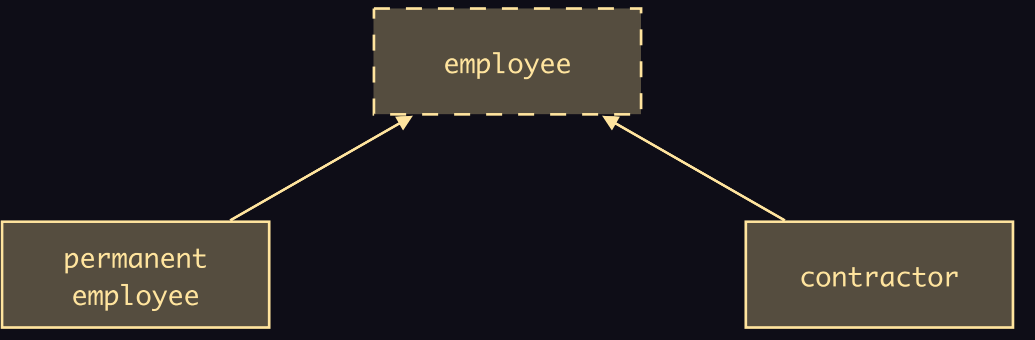
- Relational normalization → Object-relational mismatch
- Document nesting → Object-document mismatch
- Graph reification → Object-graph mismatch

Semantic integrity

Maintaining integrity of polymorphic data

Modeling inheritance polymorphism

- All employees are either permanent employees or contractors.

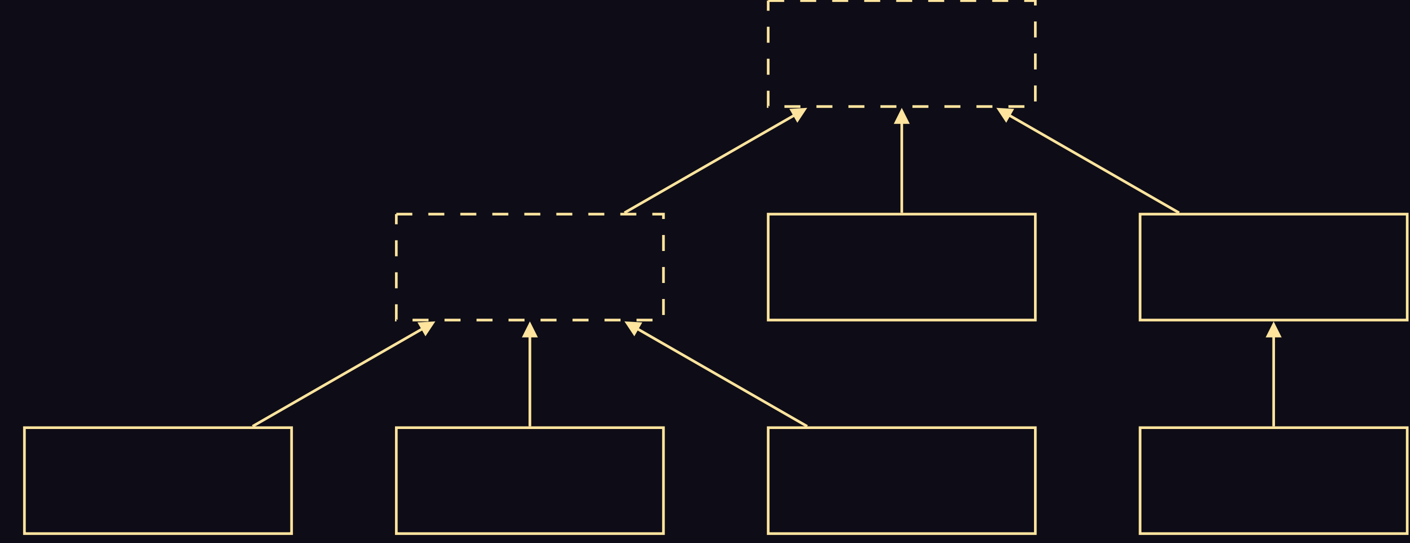


- All employees have:
  - Employee ID.
  - First name
  - Last name.
- Permanent ones have:
  - Start date.
  - End date.
  - Salary.
- Contractors have:
  - Agency ID.
  - Contract number.
  - Hourly rate.

Integrity in the relational paradigm

Design patterns for modeling inheritance (Fowler, 2002):

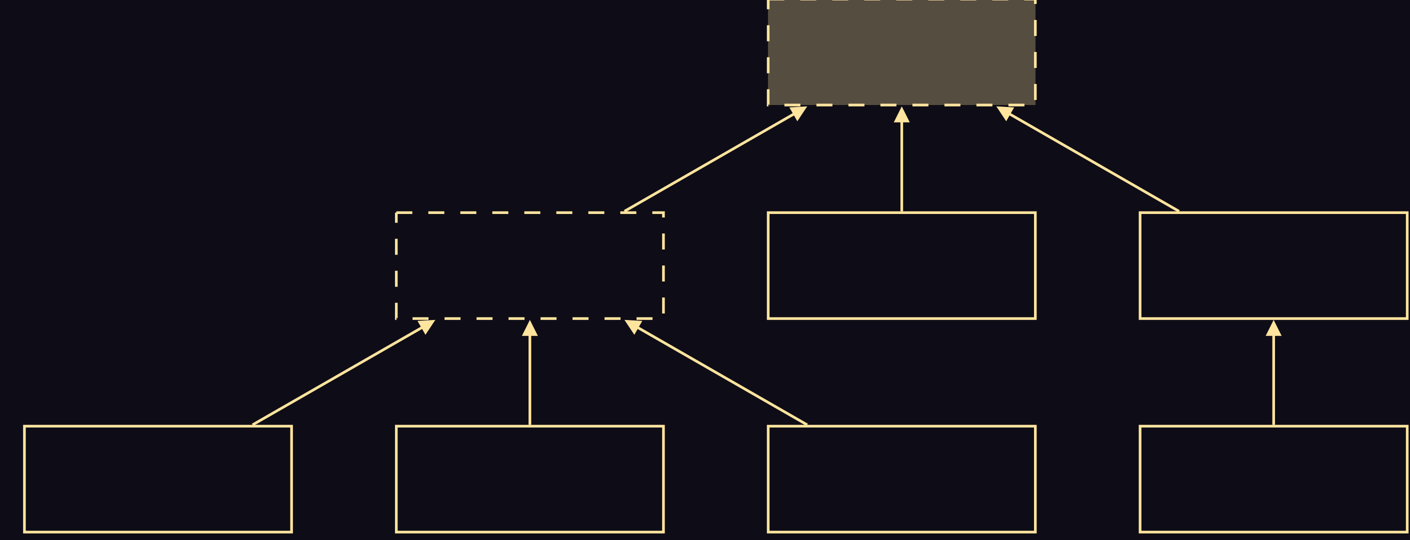
- Single-table pattern.
- Concrete-table pattern.
- Class-table pattern.



Integrity in the relational paradigm

Design patterns for modeling inheritance (Fowler, 2002):

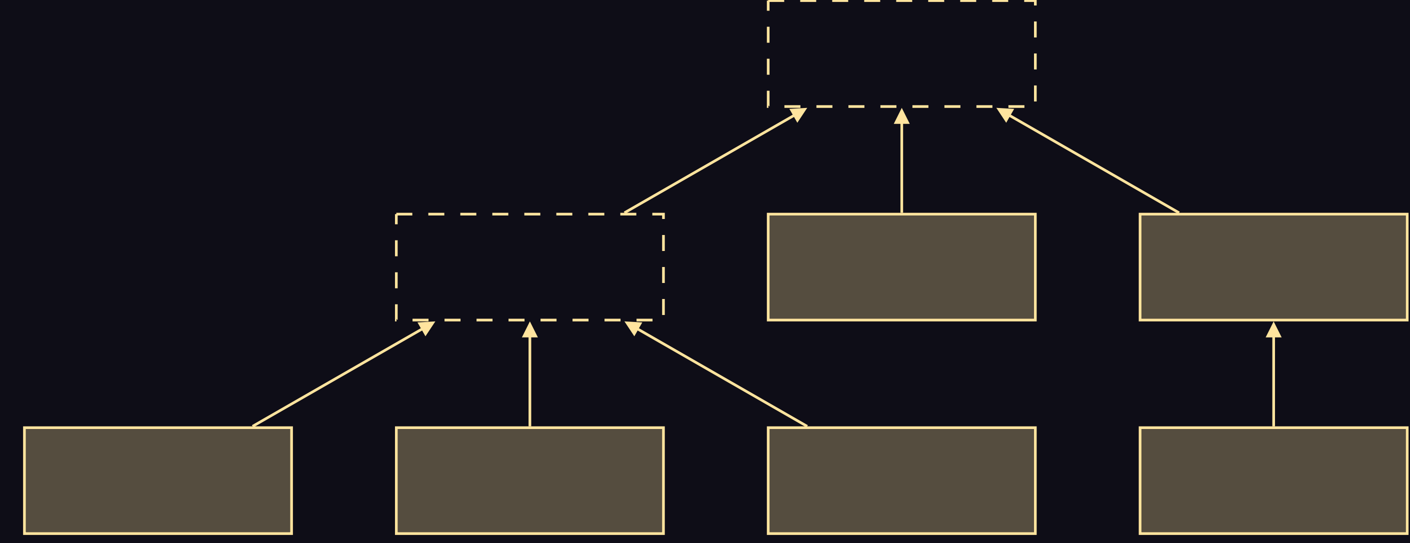
- Single-table pattern.
- Concrete-table pattern.
- Class-table pattern.



Integrity in the relational paradigm

Design patterns for modeling inheritance (Fowler, 2002):

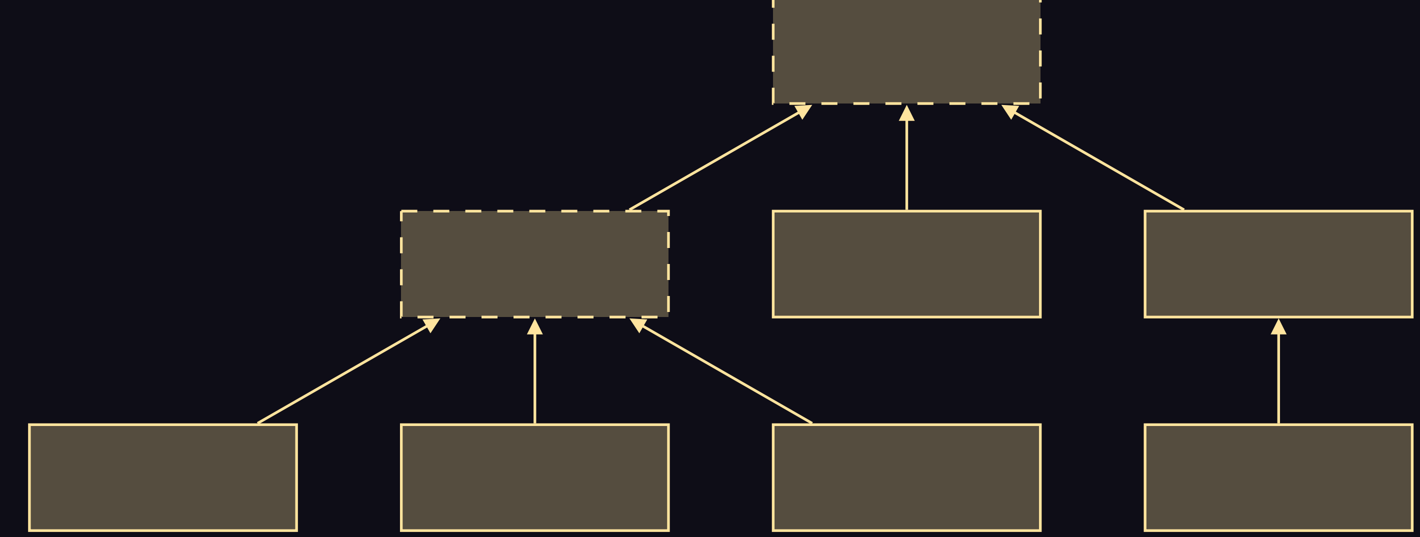
- Single-table pattern.
- Concrete-table pattern.
- Class-table pattern.



Integrity in the relational paradigm

Design patterns for modeling inheritance (Fowler, 2002):

- Single-table pattern.
- Concrete-table pattern.
- Class-table pattern.



Integrity in the relational paradigm

Single-table pattern:

employees

id	first_name	last_name	employee_type	start_date	end_date	agenc

Employee fields

Permanent employee fields

Contractor fields

Integrity in the relational paradigm

Single-table pattern:

Enum for employee types

```
CREATE TYPE EmployeeType
AS ENUM ('permanent', 'contractor');

CREATE TABLE employees (
  id SERIAL PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  employee_type EmployeeType NOT NULL,
  start_date DATE DEFAULT NULL,
  end_date DATE DEFAULT NULL,
  agency_id INT DEFAULT NULL REFERENCES agencies(id),
  contract_number INT DEFAULT NULL,
  salary MONEY DEFAULT NULL,
  hourly_rate MONEY DEFAULT NULL
);
```

```
ALTER TABLE employees
ADD CONSTRAINT check_nulls
CHECK ( (
  employee_type = 'permanent'
  AND num_nulls(start_date, salary) = 0
  AND num_nonnulls(agency_id, contract_number, hourly_rate) = 0
) OR (
  employee_type = 'contractor'
  AND num_nulls(agency_id, contract_number, hourly_rate) = 0
  AND num_nonnulls(start_date, end_date, salary) = 0
) );
```

Attribute semantics are controlled
with custom constraints

Integrity in the relational paradigm

Single-table pattern:

employees

id	first_name	last_name	employee_type	start_date	end_date	agenc

FK

```
CREATE TABLE performance_reviews (
  id SERIAL PRIMARY KEY,
  employee_id INT NOT NULL REFERENCES employees(id),
  review_date DATE NOT NULL,
  review_score INT NOT NULL,
  comments TEXT DEFAULT NULL
);
```

id	employee_id	review_date	review_score	comments

Integrity in the relational paradigm

Single-table pattern:

id	first_name	last_name	employee_type	start_date	end_date	agenc
1			permanent			
2			permanent			
3			contractor			

FK

id	employee_id	review_date	review_score	comments
	1			
	1			
	3			

There is no way to prevent performance reviews being associated with contractors

Integrity in the relational paradigm

Concrete-table pattern:

permanent_employees

id	first_name	last_name	start_date	end_date	salary

contractors

id	first_name	last_name	agency_id	contract_number	hourly_rate

Integrity in the relational paradigm

Concrete-table pattern:

```
CREATE TABLE permanent_employees (
  id INT PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  start_date DATE NOT NULL,
  end_date DATE DEFAULT NULL,
  salary MONEY NOT NULL
);

CREATE TABLE contractors (
  id INT PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  agency_id INT NOT NULL REFERENCES agencies(id),
  contract_number INT NOT NULL,
  hourly_rate MONEY NOT NULL
);
```

Integrity in the relational paradigm

Concrete-table pattern:

```
CREATE TABLE permanent_employees (
  id INT PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  start_date DATE NOT NULL,
  end_date DATE DEFAULT NULL,
  salary MONEY NOT NULL
);

CREATE TABLE contractors (
  id INT PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  agency_id INT NOT NULL REFERENCES agencies(id),
  contract_number INT NOT NULL,
  hourly_rate MONEY NOT NULL
);
```

There is no way to ensure global uniqueness of employee IDs

Integrity in the relational paradigm

Concrete-table pattern:

id	first_name	last_name	start_date	end_date	salary
1					
2					
3					

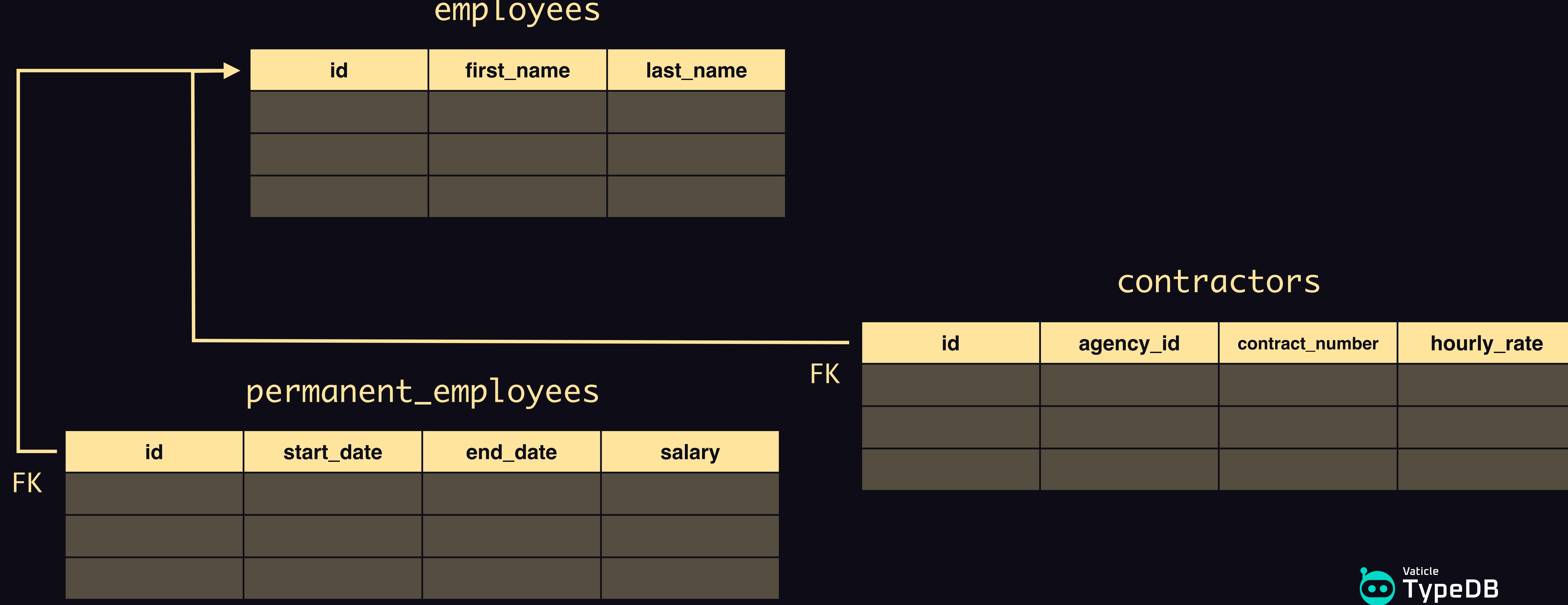
contractors

id	first_name	last_name	agency_id	contract_number	hourly_rate
2					
4					
5					

There is no way to ensure global uniqueness of employee IDs

Integrity in the relational paradigm

Class-table pattern:



Vaticle TypeDB

Integrity in the relational paradigm

Class-table pattern:

```
CREATE TABLE employees (
  id SERIAL PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL
);

CREATE TABLE permanent_employees (
  id INT PRIMARY KEY REFERENCES employee(id),
  start_date DATE NOT NULL,
  end_date DATE DEFAULT NULL,
  salary MONEY NOT NULL
);

CREATE TABLE contractors (
  id INT PRIMARY KEY REFERENCES employee(id),
  agency_id INT NOT NULL REFERENCES agencies(id),
  contract_number INT NOT NULL,
  hourly_rate MONEY NOT NULL
);
```

```
DO $$
DECLARE
  employee_id INT;
BEGIN
  INSERT INTO employees (id, first_name, last_name)
  VALUES (DEFAULT, 'Kima', 'Greggs')
  RETURNING id INTO employee_id;

  INSERT INTO permanent_employees (id, start_date, salary)
  VALUES (employee_id, '2020-09-25', 65000.00);
END $$;
```

Creating an employee involves inserting multiple rows

Integrity in the relational paradigm

Class-table pattern:

```
DO $$
DECLARE
  employee_id INT;
BEGIN
  INSERT INTO employees (id, first_name, last_name)
  VALUES (DEFAULT, 'Kima', 'Greggs')
  RETURNING id INTO employee_id;

  INSERT INTO permanent_employees (id, start_date, salary)
  VALUES (employee_id, '2020-09-25', 65000.00);
END $$;
```

employees

id	first_name	last_name
1	'Kima'	'Greggs'

permanent_employees (FK)

id	start_date	end_date	salary
1	2020-09-25	NULL	65000.00

contractors (FK)

id	agency_id	contract_number	hourly_rate

Integrity in the relational paradigm

Class-table pattern:

Employees can be neither permanent employees nor contractors

```
DO $$
DECLARE
  employee_id INT;
BEGIN
  INSERT INTO employees (id, first_name, last_name)
  VALUES (DEFAULT, 'Kima', 'Greggs')
  RETURNING id INTO employee_id;
END $$;
```

employees

id	first_name	last_name
1	'Kima'	'Greggs'

contractors

id	agency_id	contract_number	hourly_rate

FK

permanent_employees

id	start_date	end_date	salary

FK

Integrity in the relational paradigm

Class-table pattern:

Employees can be both permanent employees and contractors

```
DO $$
DECLARE
  employee_id INT;
BEGIN
  INSERT INTO employees (id, first_name, last_name)
  VALUES (DEFAULT, 'Kima', 'Greggs')
  RETURNING id INTO employee_id;

  INSERT INTO permanent_employees (id, start_date, salary)
  VALUES (employee_id, '2020-09-25', 65000.00);

  INSERT INTO contractors (id, agency_id, contract_number, hourly_rate)
  VALUES (employee_id, 8, 301, 50.00);
END $$;
```

employees

id	first_name	last_name
1	'Kima'	'Greggs'

contractors

FK

id	agency_id	contract_number	hourly_rate
1	8	301	50.00

permanent_employees

FK

id	start_date	end_date	salary
1	2020-09-25	NULL	65000.00

Integrity in the relational paradigm

- Single-table pattern → Cannot enforce subtype properties
- Concrete-table pattern → Cannot enforce supertype properties
- Class-table pattern → Cannot enforce object identity

CANNOT ENFORCE SEMANTIC INTEGRITY

Integrity in the document paradigm

- Similar documents should be stored in the same collection for optimization.
- This is called the “polymorphic pattern“ (Coupal et al., 2023).
- Splitting collections more granularly is an anti-pattern.

Integrity in the document paradigm

```
db.employees.insert( [
  {
    "id": 1,
    "first_name": "Kima",
    "last_name": "Greggs",
    "employee_type": "permanent_employee",
    "start_date": new Date("2020-09-25"),
    "salary": 65000.00
  },
  {
    "id": 2,
    "first_name": "Reginald",
    "last_name": "Cousins",
    "employee_type": "contractor",
    "agency": { "id": 8, "name": "Baltimore Agency" },
    "contract_number": 301,
    "hourly_rate": 50.00
  }
] )
```

employees

kima

reginald

baltimore

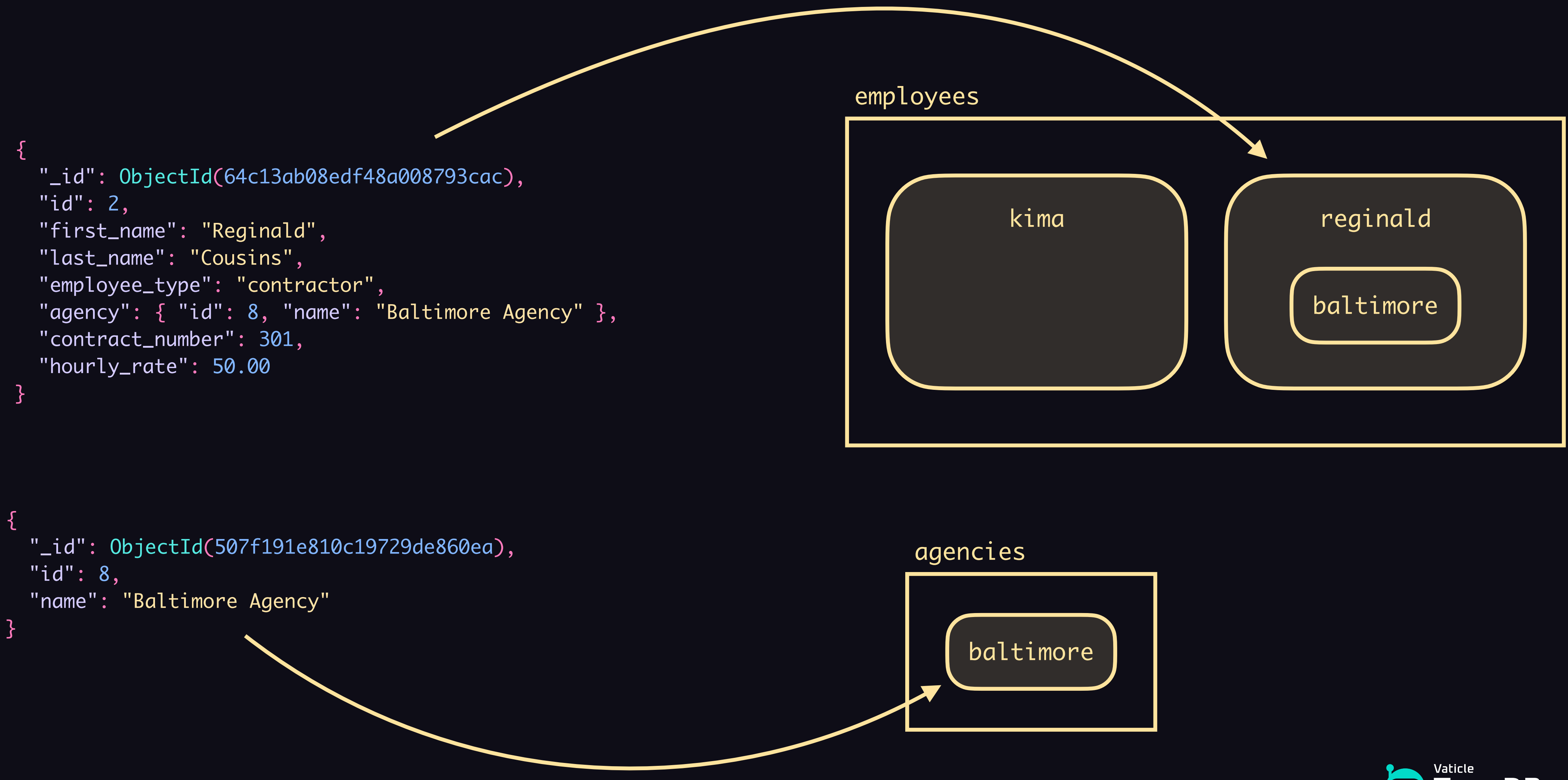
agencies

baltimore

Option 1: Agency is nested for contractors

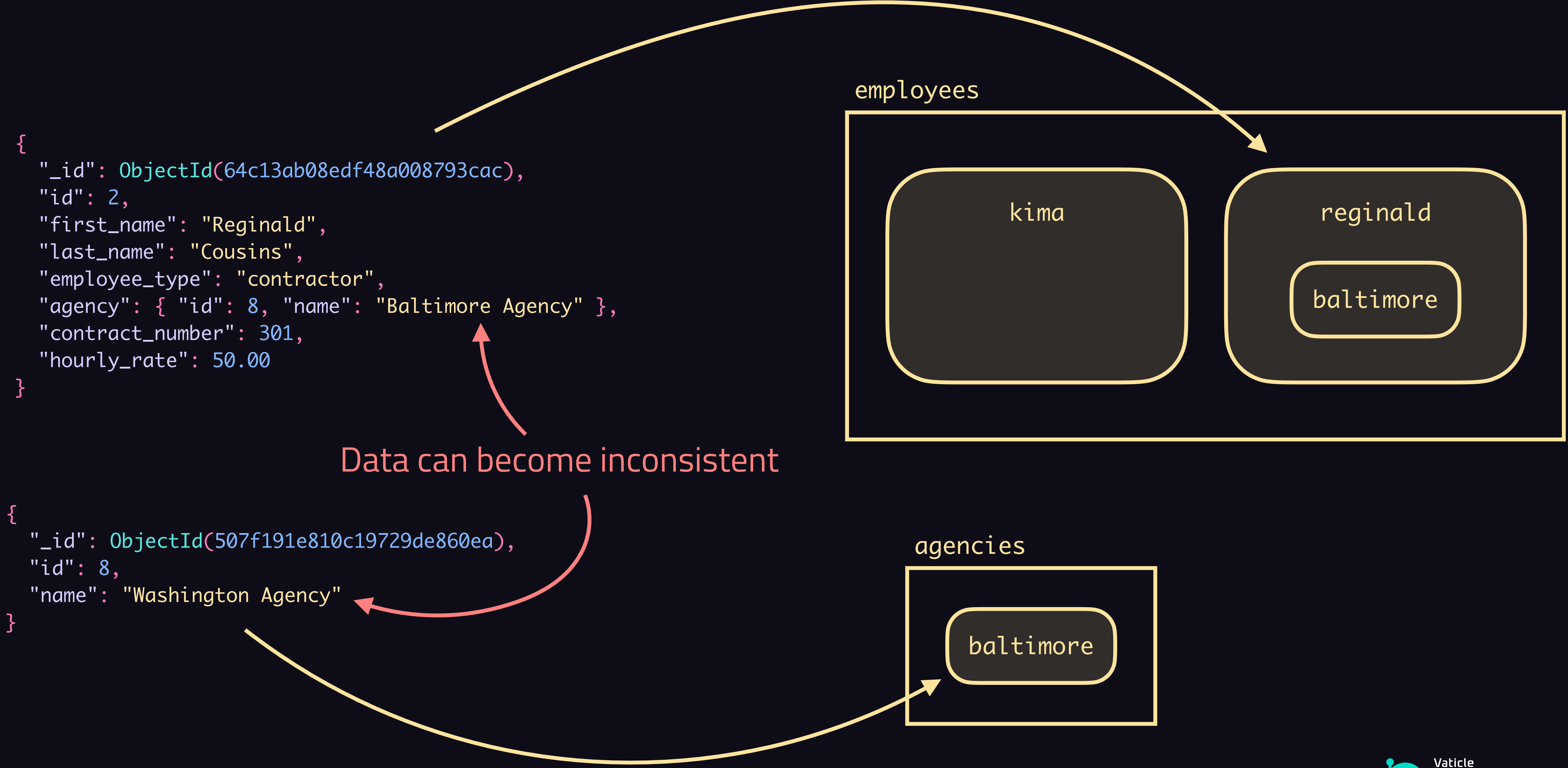
Vaticle TypeDB

Integrity in the document paradigm



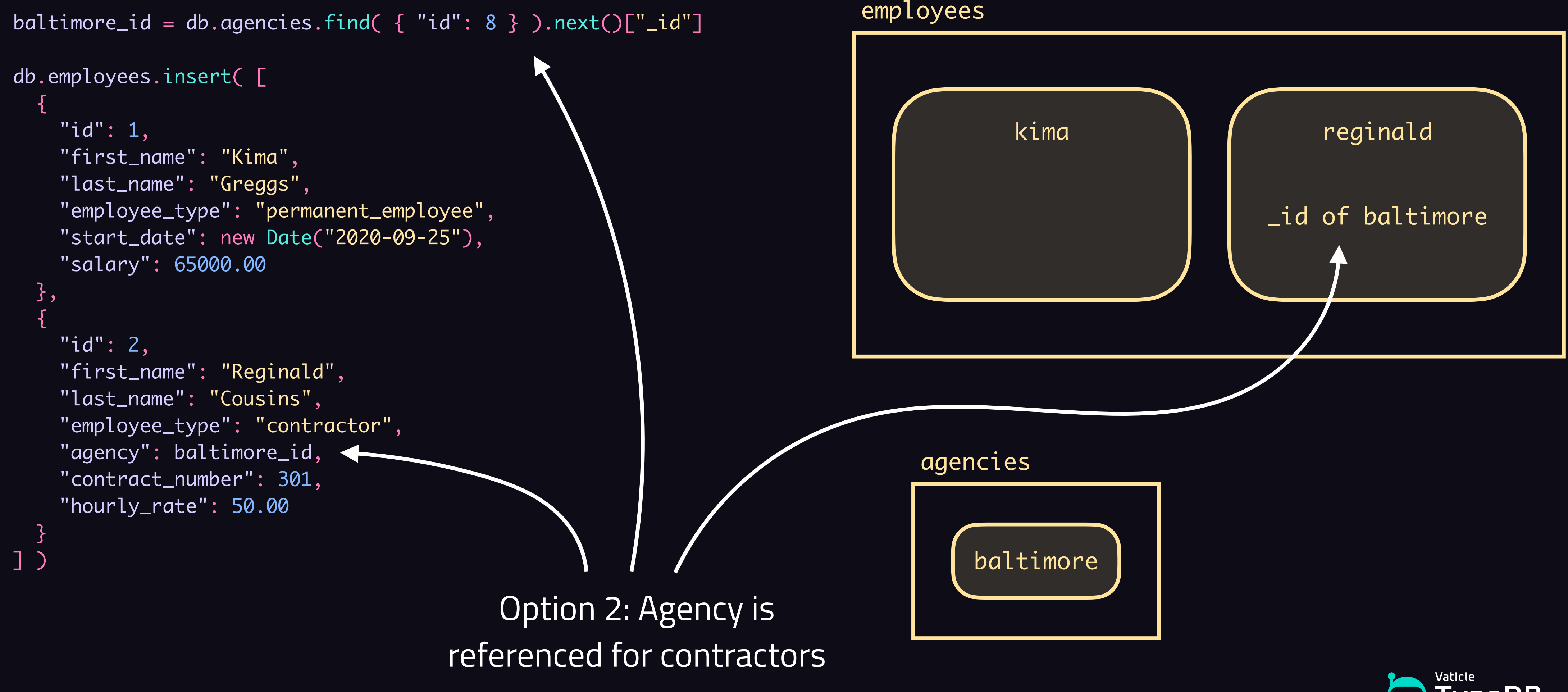
Vaticle TypeDB

Integrity in the document paradigm



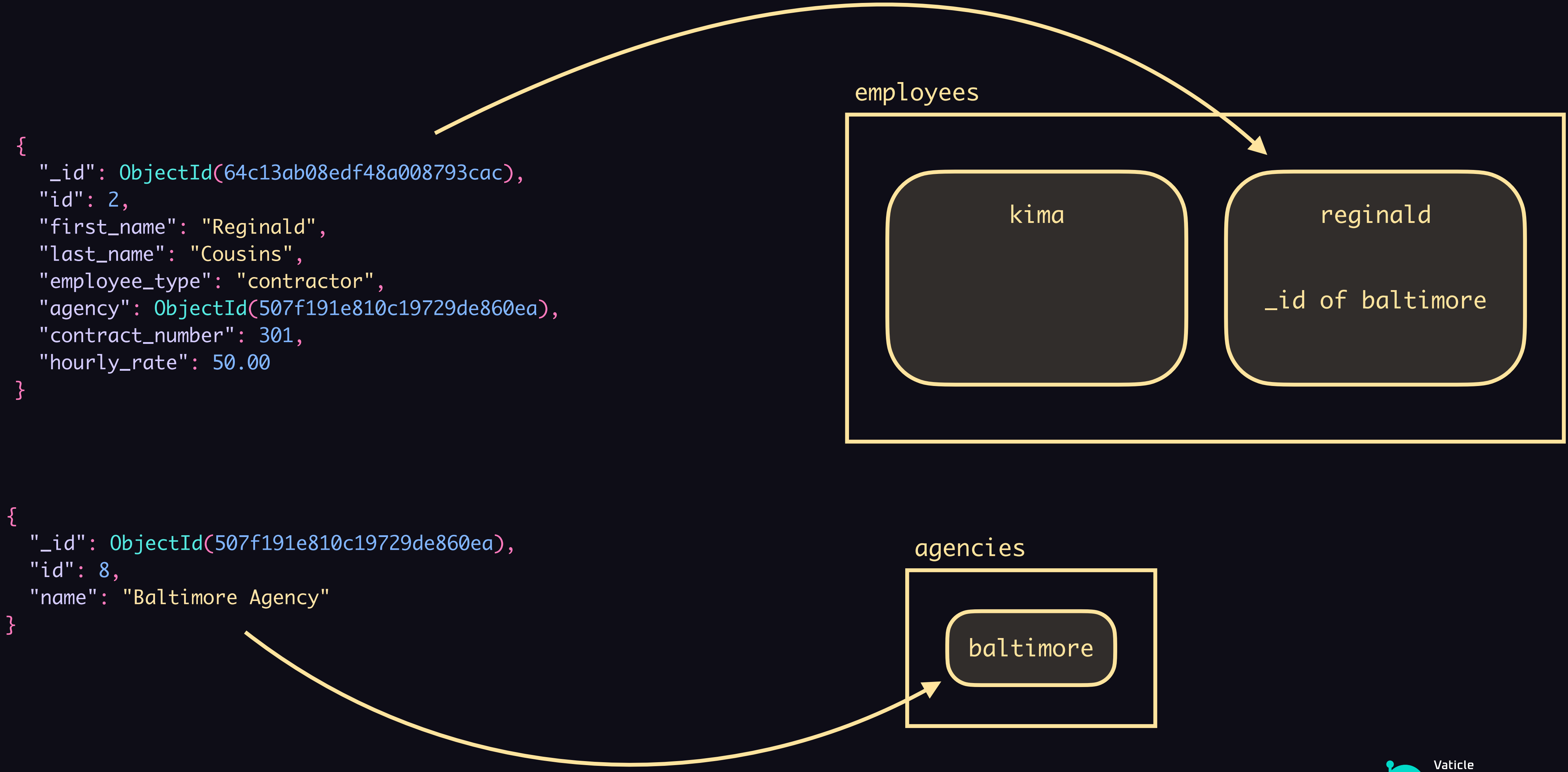
Vaticle TypeDB

Integrity in the document paradigm



Vaticle TypeDB

Integrity in the document paradigm



Vaticle TypeDB

Integrity in the document paradigm

```
{
  "_id": ObjectId(64c13ab08edf48a008793cac),
  "id": 2,
  "first_name": "Reginald",
  "last_name": "Cousins",
  "employee_type": "contractor",
  "agency": ObjectId(507f191e810c19729de860ea),
  "contract_number": 301,
  "hourly_rate": 50.00
}
```

employees

kima

reginald

_id of baltimore

Null references can be produced

agencies

Integrity in the document paradigm

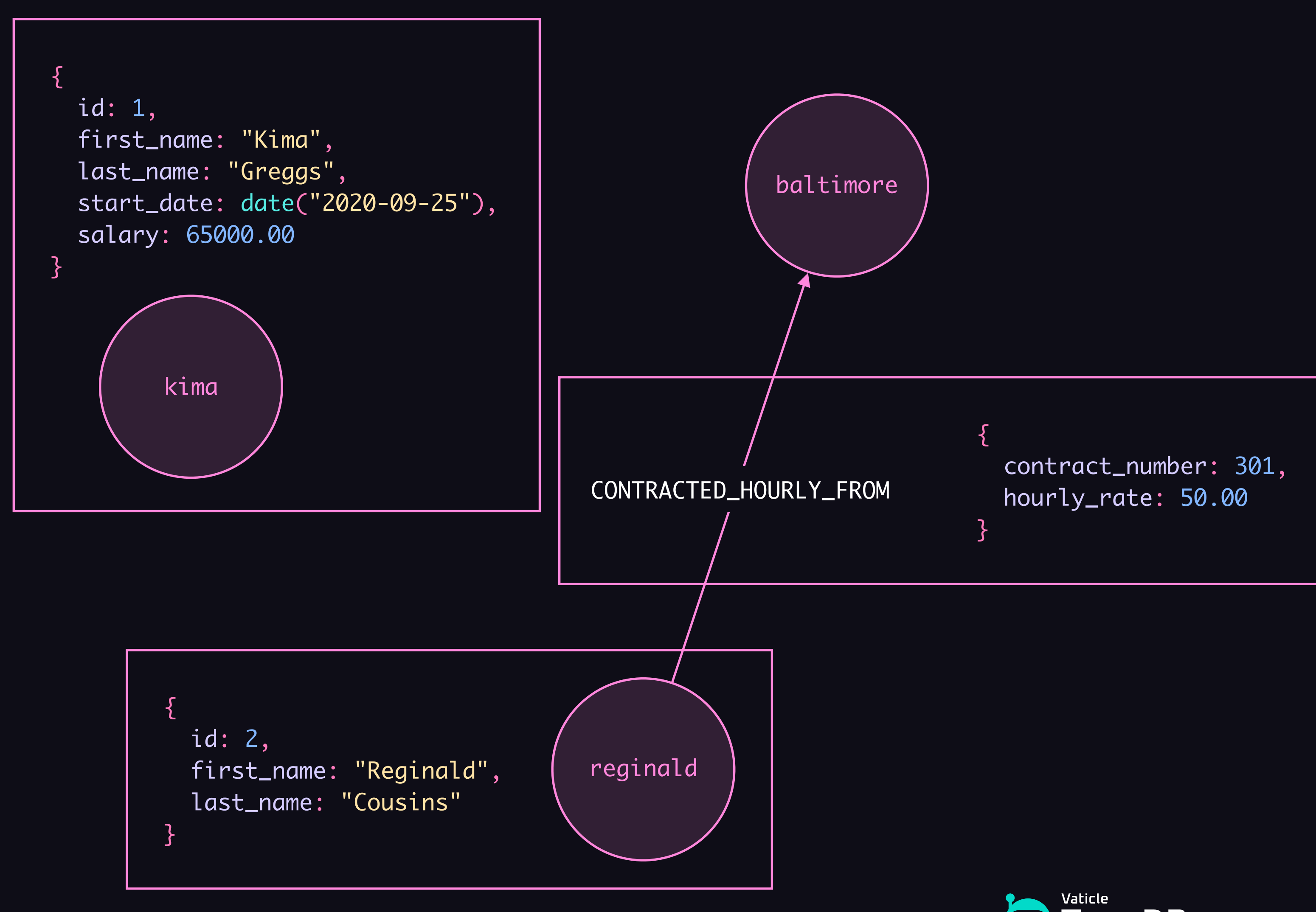
- Nesting → Cannot enforce data consistency
- Referencing → Cannot enforce referential integrity

CANNOT ENFORCE SEMANTIC INTEGRITY

Integrity in the graph paradigm

```
MATCH (baltimore:Agency {id: 8} )
CREATE
  (kima:Employee:PermanentEmployee {
    id: 1,
    first_name: "Kima",
    last_name: "Greggs",
    start_date: date("2020-09-25"),
    salary: 65000.00
  } ),
  (reginald:Employee:Contractor {
    id: 2,
    first_name: "Reginald",
    last_name: "Cousins"
  } )-[:CONTRACTED_HOURLY_FROM {
    contract_number: 301,
    hourly_rate: 50.00
  } ]->(baltimore);
```

Multiple labels can be used to emulate an inheritance hierarchy



Vaticle TypeDB

Integrity in the graph paradigm

Constraints can be defined on a per-label basis

```
MATCH (baltimore:Agency {id: 8} )
CREATE
  (kima:Employee:PermanentEmployee {
    id: 1,
    first_name: "Kima",
    last_name: "Greggs",
    start_date: date("2020-09-25"),
    salary: 65000.00
  } ),
  (reginald:Employee:Contractor {
    id: 2,
    first_name: "Reginald",
    last_name: "Cousins"
  } )-[:CONTRACTED_HOURLY_FROM {
    contract_number: 301,
    hourly_rate: 50.00
  } ]->(baltimore);
```

```
CREATE CONSTRAINT FOR (n:Employee)
REQUIRE n.id IS KEY;

CREATE CONSTRAINT FOR (n:Employee)
REQUIRE n.id IS TYPED INTEGER;

CREATE CONSTRAINT FOR (n:Employee)
REQUIRE n.first_name IS NOT NULL;

CREATE CONSTRAINT FOR (n:Employee)
REQUIRE n.first_name IS TYPED STRING;

CREATE CONSTRAINT FOR (n:Employee)
REQUIRE n.last_name IS NOT NULL;

CREATE CONSTRAINT FOR (n:Employee)
REQUIRE n.last_name IS TYPED STRING;
```

Integrity in the graph paradigm

Constraints can be defined on a per-label basis

```
MATCH (baltimore:Agency {id: 8} )
CREATE
  (kima:Employee:PermanentEmployee {
    id: 1,
    first_name: "Kima",
    last_name: "Greggs",
    start_date: date("2020-09-25"),
    salary: 65000.00
  } ),
  (reginald:Employee:Contractor {
    id: 2,
    first_name: "Reginald",
    last_name: "Cousins"
  } )-[:CONTRACTED_HOURLY_FROM {
    contract_number: 301,
    hourly_rate: 50.00
  } ]->(baltimore);
```

```
CREATE CONSTRAINT FOR (n:PermanentEmployee)
REQUIRE n.start_date IS NOT NULL;

CREATE CONSTRAINT FOR (n:PermanentEmployee)
REQUIRE n.start_date IS TYPED DATE;

CREATE CONSTRAINT FOR (n:PermanentEmployee)
REQUIRE n.end_date IS TYPED DATE;

CREATE CONSTRAINT FOR (n:PermanentEmployee)
REQUIRE n.salary IS NOT NULL;

CREATE CONSTRAINT FOR (n:PermanentEmployee)
REQUIRE n.salary IS TYPED FLOAT;
```

Integrity in the graph paradigm

```
MATCH (baltimore:Agency {id: 8} )
CREATE
  (kima:Employee:PermanentEmployee {
    id: 1,
    first_name: "Kima",
    last_name: "Greggs",
    start_date: date("2020-09-25"),
    salary: 65000.00
  } ),
  (reginald:Employee:Contractor {
    id: 2,
    first_name: "Reginald",
    last_name: "Cousins"
  } )-[:CONTRACTED_HOURLY_FROM {
    contract_number: 301,
    hourly_rate: 50.00
  } ]->(baltimore);
```

Constraints can be defined on a per-label basis

```
CREATE CONSTRAINT FOR ()-[r:CONTRACTED_HOURLY_FROM]-()
REQUIRE r.contract_number IS KEY;

CREATE CONSTRAINT FOR ()-[r:CONTRACTED_HOURLY_FROM]-()
REQUIRE r.contract_number IS TYPED INTEGER;

CREATE CONSTRAINT FOR ()-[r:CONTRACTED_HOURLY_FROM]-()
REQUIRE r.hourly_rate IS NOT NULL;

CREATE CONSTRAINT FOR ()-[r:CONTRACTED_HOURLY_FROM]-()
REQUIRE r.hourly_rate IS TYPED FLOAT;
```

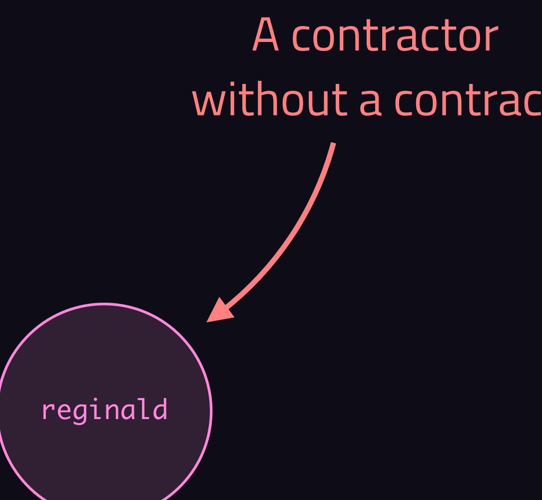
Integrity in the graph paradigm

```
MATCH (baltimore:Agency {id: 8} )
CREATE
  (kima:Employee:PermanentEmployee {
    id: 1,
    first_name: "Kima",
    last_name: "Greggs",
    start_date: date("2020-09-25"),
    salary: 65000.00
  } ),
  (reginald:Employee:Contractor {
    id: 2,
    first_name: "Reginald",
    last_name: "Cousins"
  } )-[:CONTRACTED_HOURLY_FROM {
    contract_number: 301,
    hourly_rate: 50.00
  } ]->(baltimore);
```

It is not possible to control the types of relationships that can be made between nodes

Integrity in the graph paradigm

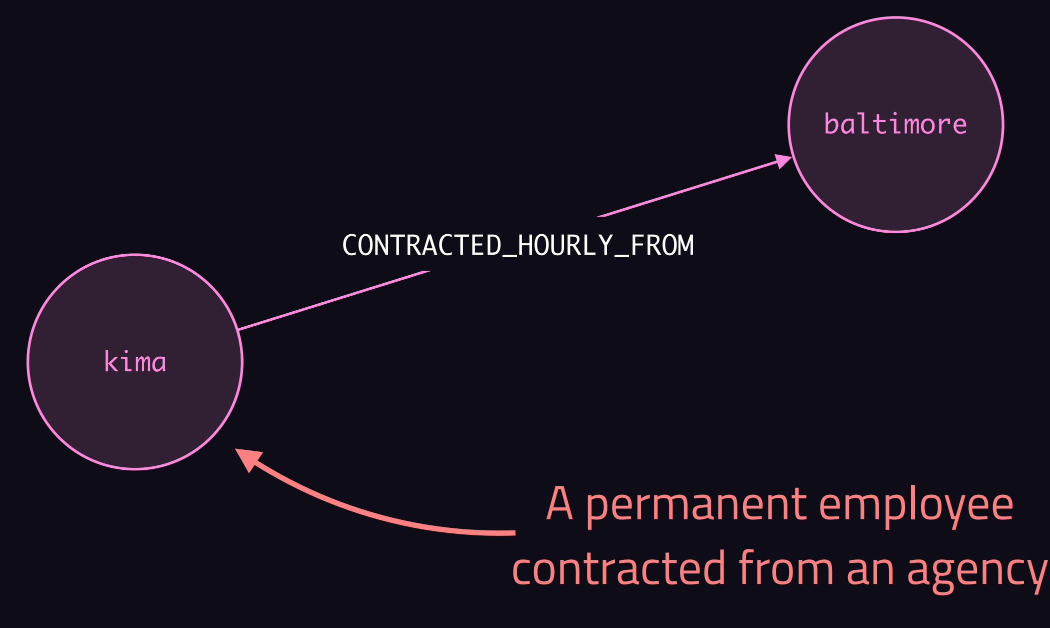
```
MATCH (baltimore:Agency {id: 8} )
CREATE
  (kima:Employee:PermanentEmployee {
    id: 1,
    first_name: "Kima",
    last_name: "Greggs",
    start_date: date("2020-09-25"),
    salary: 65000.00
  } ),
  (reginald:Employee:Contractor {
    id: 2,
    first_name: "Reginald",
    last_name: "Cousins"
  } );
```



It is not possible to control the types of relationships that can be made between nodes

Integrity in the graph paradigm

```
MATCH (baltimore:Agency {id: 8} )
CREATE
  (kima:Employee:PermanentEmployee {
    id: 1,
    first_name: "Kima",
    last_name: "Greggs",
    start_date: date("2020-09-25"),
    salary: 65000.00
  } )-[:CONTRACTED_HOURLY_FROM {
    contract_number: 301,
    hourly_rate: 50.00
  } ]->(baltimore),
  (reginald:Employee:Contractor {
    id: 2,
    first_name: "Reginald",
    last_name: "Cousins"
  } );
```



It is not possible to control the types of relationships that can be made between nodes

Integrity in the graph paradigm

```
MATCH (baltimore:Agency {id: 8} )
CREATE
  (kima:Employee:PermanentEmployee {
    id: 1,
    first_name: "Kima",
    last_name: "Greggs",
    start_date: date("2020-09-25"),
    salary: 65000.00
  } ),
  (reginald:Employee:Contractor {
    id: 2,
    first_name: "Reginald",
    last_name: "Cousins"
  } )-[:CONTRACTED_HOURLY_FROM {
    contract_number: 301,
    hourly_rate: 50.00
  } ]->(kima);
```

baltimore

kima

CONTRACTED_HOURLY_FROM

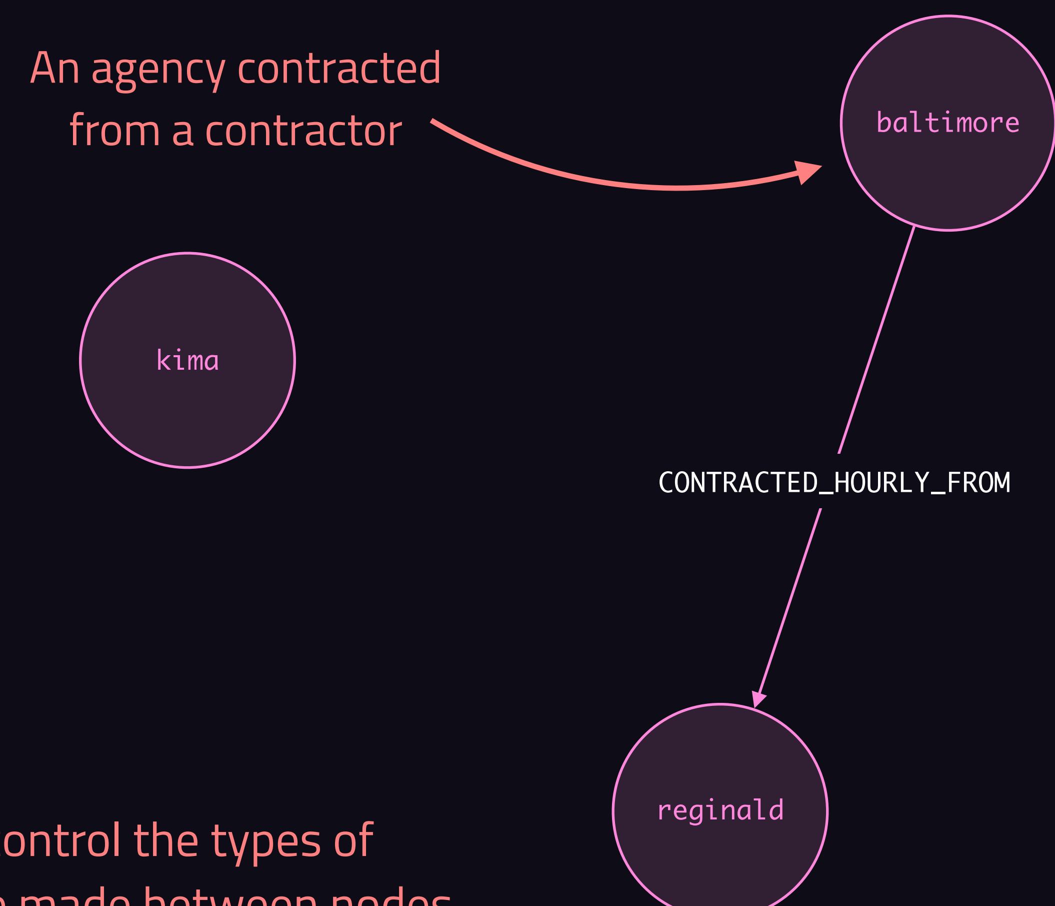
reginald

A contractor contracted from a permanent employee

It is not possible to control the types of relationships that can be made between nodes

Integrity in the graph paradigm

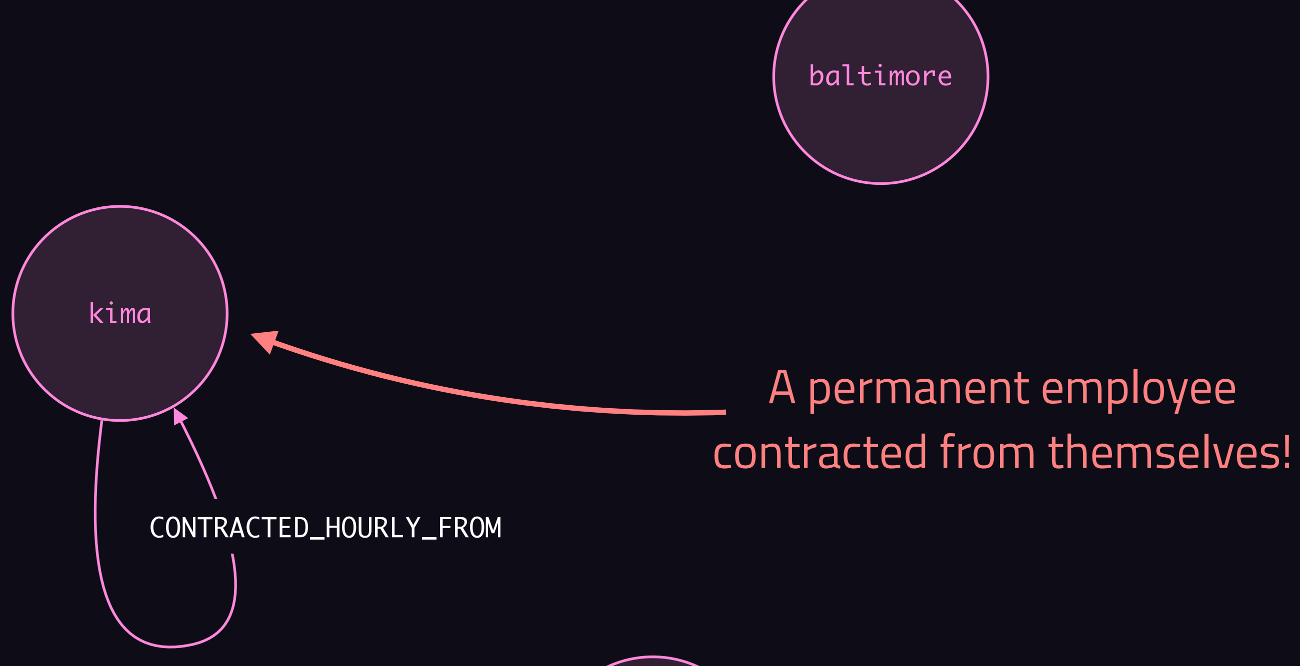
```
MATCH (baltimore:Agency {id: 8} )
CREATE
  (kima:Employee:PermanentEmployee {
    id: 1,
    first_name: "Kima",
    last_name: "Greggs",
    start_date: date("2020-09-25"),
    salary: 65000.00
  } ),
  (reginald:Employee:Contractor {
    id: 2,
    first_name: "Reginald",
    last_name: "Cousins"
  } )<-[:CONTRACTED_HOURLY_FROM {
    contract_number: 301,
    hourly_rate: 50.00
  } ]-(baltimore);
```



It is not possible to control the types of relationships that can be made between nodes

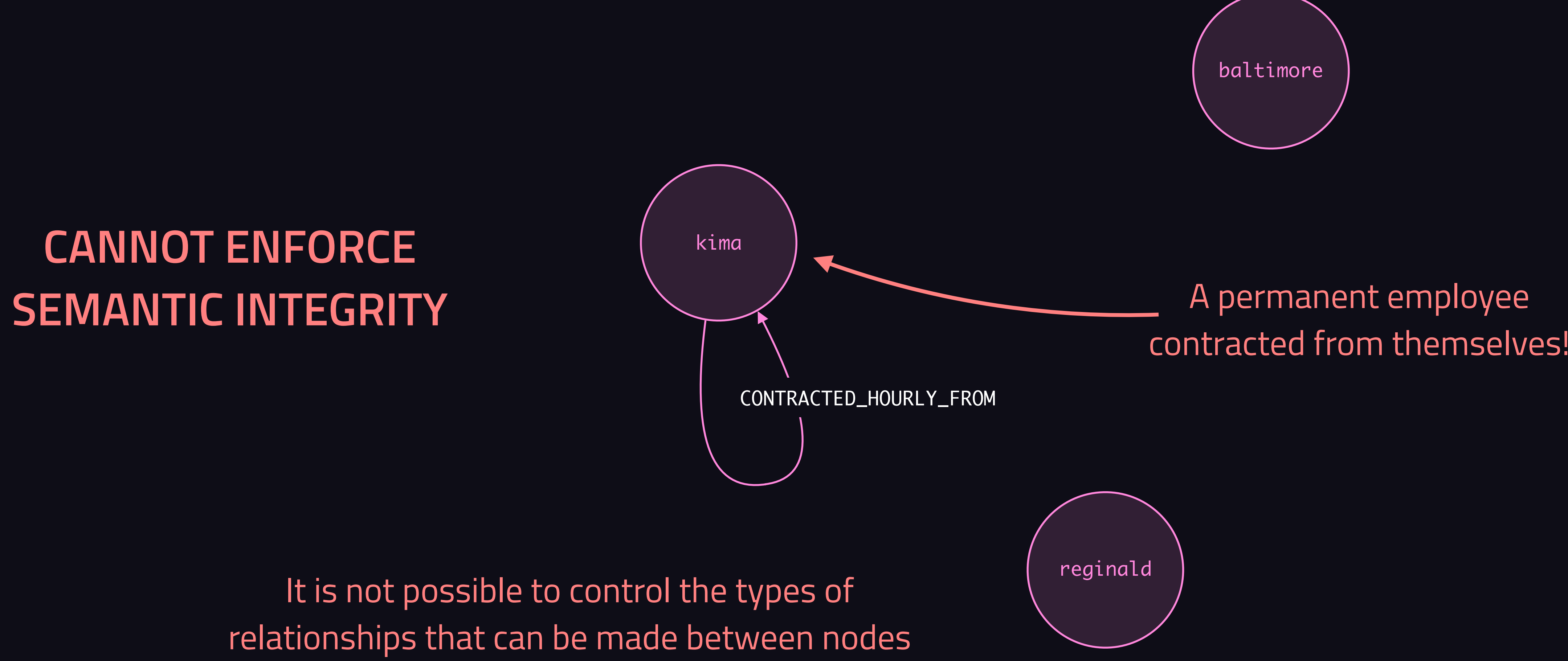
Integrity in the graph paradigm

```
MATCH (baltimore:Agency {id: 8} )
CREATE
  (kima:Employee:PermanentEmployee {
    id: 1,
    first_name: "Kima",
    last_name: "Greggs",
    start_date: date("2020-09-25"),
    salary: 65000.00
  } )-[:CONTRACTED_HOURLY_FROM {
    contract_number: 301,
    hourly_rate: 50.00
  } ]->(kima),
  (reginald:Employee:Contractor {
    id: 2,
    first_name: "Reginald",
    last_name: "Cousins"
  } );
```



It is not possible to control the types of relationships that can be made between nodes

Integrity in the graph paradigm



Summary

- Relational databases → Cannot enforce semantic integrity
- Document databases → Cannot enforce semantic integrity
- Graph databases → Cannot enforce semantic integrity

Polymorphic querying

How current databases lead to brittle architectures

An object model for a simple filesystem

Owner
Ownership
Admin
Owned
Email
Key
User
Owner
GroupOwnership
Name
Key
UserGroup
Owned
Path
Key
File
Owner
ResourceOwnership
Id
Key<ID>
Resource<ID>
Owned

https://typedb.com/lectures/why-polymorphic-database

An object model for a simple filesystem

Selected constructors and methods of UserGroup:

```
UserGroup(String name, GroupOwnership.Owner owner) → UserGroup // Instantiates a group with a new name and existing owner
UserGroup(Name name, GroupOwnership.Owner owner) → UserGroup   // Instantiates a group with an existing name and owner
UserGroup.getId() → Name                                       // Returns the group's name
UserGroup.getOwnership() → GroupOwnership                      // Returns the ownership of the group
UserGroup.getOwner() → GroupOwnership.Owner                    // Returns the group's owner
UserGroup.getOwned() → Ownership.Owned                         // Returns things the group owns
UserGroup.getOwnedResources() → ResourceOwnership.Owned        // Returns resources the group owns
```

An object model for a simple filesystem

Objects in the filesystem:

```
Admin cedric = new Admin("cedric@vaticle.com");

User jimmy = new User("jimmy@vaticle.com");

UserGroup engineers = new UserGroup("engineers", cedric);

File benchmark = new File("/jimmy/benchmark-results.xlsx", jimmy);

File roadmap = new File("/vaticle/feature-roadmap.pdf", engineers);
```

An object model for a simple filesystem

1. *"Retrieve the ID of a given resource's owner."*

```
// Retreive the ID of the feature roadmap's owner.
System.out.println(((Id.Key<?>) roadmap.getOwner()).getId().value);

// Output:
// engineers
```

2. *"Retrieve the IDs of all resources owned by a given user."*

```
// Retrieve the IDs of all resources owned by Jimmy.
System.out.println(jimmy.getOwnedResources().stream()
  .map(resource -> ((Id.Key<?>) resource).getId().value)
  .collect(Collectors.toSet())
);

// Output:
// [ /jimmy/benchmark-results.xlsx ]
```

An object model for a simple filesystem

3. *"Retrieve the types and IDs of all objects in the filesystem."*

```
// Make a collection of all filesystem objects.
Set<Id.Key<?>> fileSystemObjects = Set.of(
  cedric, jimmy, engineers, benchmark, roadmap
);

// Retrieve the type and ID of all objects in the filesystem.
fileSystemObjects.stream()
  .map(object -> Map.of(
    "object-type", object.getClass().getSimpleName(),
    "object-id", ((Id.Key<?>) object).getId().value
  ))
  .forEach(System.out::println);

// Output:
// { object-id=/vaticle/feature-roadmap.pdf, object-type=File }
// { object-id=/jimmy/benchmark-results.xlsx, object-type=File }
// { object-id=engineers, object-type=UserGroup }
// { object-id=cedric@vaticle.com, object-type=Admin }
// { object-id=jimmy@vaticle.com, object-type=User }
```

An object model for a simple filesystem

4. *"Retrieve the details of every ownership in the filesystem."*

```java
// Retrieve the details of every ownership in the filesystem.
fileSystemObjects.stream()
  .filter(object -> object instanceof Ownership.Owned)
  .map(object -> (Ownership.Owned) object)
  .map(owned -> Map.of(
    "owned-id", ((Id.Key<?>) owned).getId().value,
    "owned-type", owned.getClass().getSimpleName(),
    "owner-id", (((Id.Key<?>) owned.getOwner()).getId()).value,
    "owner-type", owned.getOwner().getClass().getSimpleName(),
    "ownership-type", owned.getOwnership().getClass().getSimpleName()
  ))
  .forEach(System.out::println);

// Output:
// {
//   owned-id=/vaticle/feature-roadmap.pdf,
//   ownership-type=ResourceOwnership,
//   owner-type=UserGroup,
//   owner-id=engineers,
//   owned-type=File
// }
// {
//   owned-id=/jimmy/benchmark-results.xlsx,
//   ownership-type=ResourceOwnership,
//   owner-type=User,
//   owner-id=jimmy@vaticle.com,
//   owned-type=File
// }
// {
//   owned-id=engineers,
//   ownership-type=GroupOwnership,
//   owner-type=Admin,
//   owner-id=cedric@vaticle.com,
//   owned-type=UserGroup
// }
```

The relational implementation

```
CREATE TABLE users (
  email TEXT PRIMARY KEY
);

CREATE TABLE admins (
  email TEXT PRIMARY KEY REFERENCES users(email)
);


CREATE TABLE user_groups (
  name TEXT PRIMARY KEY
);


CREATE TABLE resources (
  id TEXT PRIMARY KEY
);

CREATE TABLE files (
  path TEXT PRIMARY KEY REFERENCES resources(id)
);
```

Using Fowler's class-table inheritance pattern

The relational implementation

Abstract ownership tables

```
CREATE TABLE ownerships (
  id SERIAL PRIMARY KEY
);

CREATE TABLE group_ownerships (
  id INT PRIMARY KEY REFERENCES ownerships(id)
);

CREATE TABLE resource_ownerships (
  id INT PRIMARY KEY REFERENCES ownerships(id)
);
```

```
CREATE TABLE admin_of_group_ownerships (
  id INT PRIMARY KEY REFERENCES group_ownerships(id),
  admin_id TEXT REFERENCES admins(email),
  user_group_id TEXT REFERENCES user_groups(name)
);

CREATE TABLE user_of_resource_ownerships (
  id INT PRIMARY KEY REFERENCES resource_ownerships(id),
  user_id TEXT REFERENCES users(email),
  resource_id TEXT REFERENCES resources(id)
);

CREATE TABLE user_group_of_resource_ownerships (
  id INT PRIMARY KEY REFERENCES resource_ownerships(id),
  user_group_id TEXT REFERENCES user_groups(name),
  resource_id TEXT REFERENCES resources (id)
);
```

Concrete ownership tables

The relational implementation

```sql
DO $$
DECLARE
  ownership_id INT;
BEGIN
  INSERT INTO resources (id)
  VALUES ('/jimmy/benchmark-results.xlsx');

  INSERT INTO files (path)
  VALUES ('/jimmy/benchmark-results.xlsx');

  INSERT INTO ownerships (id)
  VALUES (DEFAULT)
  RETURNING id INTO ownership_id;

  INSERT INTO resource_ownerships (id)
  VALUES (ownership_id);

  INSERT INTO user_of_resource_ownerships (id, user_id, resource_id)
  VALUES (ownership_id, 'jimmy@vaticle.com', '/jimmy/benchmark-results.xlsx');
END $$;
```

Instantiate a file with the specified path, and assign the user Jimmy as its owner

The relational implementation

4. "Retrieve the details of every ownership in the filesystem."

```
SELECT ownership_type, owned_type, owned_id, owner_type, owner_id
FROM (
  SELECT
    'group_ownership' AS ownership_type,
    'user_group' AS owned_type,
    admin_of_group_ownerships.user_group_id AS owned_id,
    'admin' AS owner_type,
    admin_of_group_ownerships.admin_id AS owner_id
  FROM ownerships
  JOIN admin_of_group_ownerships USING (id)
  UNION
  SELECT
    'resource_ownership' AS ownership_type,
    'file' AS owned_type,
    files.path AS owned_id,
    'user' AS owner_type,
    user_of_resource_ownerships.user_id AS owner_id
  FROM ownerships
  JOIN user_of_resource_ownerships USING (id)
  JOIN files ON files.path = user_of_resource_ownerships.resource_id
  WHERE user_of_resource_ownerships.user_id NOT IN (
    SELECT admins.email
    FROM admins
  )
  ...
```

```
  UNION
  SELECT
    'resource_ownership' AS ownership_type,
    'file' AS owned_type,
    files.path AS owned_id,
    'admin' AS owner_type,
    admins.email AS owner_id
  FROM ownerships
  JOIN user_of_resource_ownerships USING (id)
  JOIN files ON files.path = user_of_resource_ownerships.resource_id
  JOIN admins ON admins.email = user_of_resource_ownerships.user_id
  UNION
  SELECT
    'resource_ownership' AS ownership_type,
    'file' AS owned_type,
    files.path AS owned_id,
    'user_group' AS owner_type,
    user_group_of_resource_ownerships.user_group_id AS owner_id
  FROM ownerships
  JOIN user_group_of_resource_ownerships USING (id)
  JOIN files ON files.path = user_group_of_resource_ownerships.resource_id
) AS ownerships;
```

The relational implementation

4. *"Retrieve the details of every ownership in the filesystem."*

```
SELECT ownership_type, owned_type, owned_id, owner_type, owner_id
FROM (
  SELECT
    'group_ownership' AS ownership_type,
    'user_group' AS owned_type,
    admin_of_group_ownerships.user_group_id AS owned_id,
    'admin' AS owner_type,
    admin_of_group_ownerships.admin_id AS owner_id
  FROM ownerships
  JOIN admin_of_group_ownerships USING (id)
  UNION
  SELECT
    'resource_ownership' AS ownership_type,
    'file' AS owned_type,
    files.path AS owned_id,
    'user' AS owner_type,
    user_of_resource_ownerships.user_id AS owner_id
  FROM ownerships
  JOIN user_of_resource_ownerships USING (id)
  JOIN files ON files.path = user_of_resource_ownerships.resource_id
  WHERE user_of_resource_ownerships.user_id NOT IN (
    SELECT admins.email
    FROM admins
  )
  ...
```

Return types are hardcoded with unions and joins

Type names hardcoded

Attribute interfaces hardcoded

Type inheritance hardcoded

The relational implementation

- Return types must be hardcoded with unions and joins.
- One union branch is needed for each combination of concrete classes that implement the `Owned` and `Owner` interfaces, either directly or by inheritance.

Ownership.Owned implementation	Ownership.Owner implementation
UserGroup implements GroupOwnership.Owned	Admin implements GroupOwnership.Owner
File implements ResourceOwnership.Owned	User implements ResourceOwnership.Owner
File implements ResourceOwnership.Owned	Admin implements ResourceOwnership.Owner
File implements ResourceOwnership.Owned	UserGroup implements ResourceOwnership.Owner

The document implementation

```
db.users.insert( {
  "email": "cedric@vaticle.com",
  "user_type": "admin"
} )

db.users.insert( {
  "email": "jimmy@vaticle.com",
  "user_type": "user"
} )

owner_id = db.users.find( {
  "email": "cedric@vaticle.com"
} ).next()["_id"]

db.groups.insert( {
  "name": "engineers",
  "group_type": "user_group",
  "owner": owner_id
} )
```

Using Coupal's polymorphic pattern

Using references for dependent data

```
owner_id = db.users.find( {
  "email": "jimmy@vaticle.com"
} ).next()["_id"]

db.resources.insert( {
  "path": "/jimmy/benchmark-results.xlsx",
  "resource_type": "file",
  "owner": owner_id
} )

owner_id = db.groups.find( {
  "name": "engineers"
} ).next()["_id"]

db.resources.insert( {
  "path": "/vaticle/feature-roadmap.pdf",
  "resource_type": "file",
  "owner": owner_id
} )
```

The document implementation

4. *"Retrieve the details of every ownership in the filesystem."*

```
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_owner"
  } },
  { "$lookup": {
    "from": "groups",
    "localField": "owner",
    "foreignField": "_id",
    "as": "user_group_owner"
  } },
  { "$addFields": { "owner": { "$concatArrays": [ "$user_owner", $user_group_owner" ] } } },
  { "$unwind": "$owner" },
  { "$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": { "$eq": [ "$owner.user_type", "user" ] }, "then": "user" },
      { "case": { "$eq": [ "$owner.user_type", "admin" ] }, "then": "admin" },
      { "case": { "$eq": [ "$owner.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": { "$eq": [ "$owner.user_type", "user" ] }, "then": "$owner.email" },
      { "case": { "$eq": [ "$owner.user_type", "admin" ] }, "then": "$owner.email" },
      { "case": { "$eq": [ "$owner.group_type", "user_group" ] }, "then": "$owner.name" }
    ] } }
  } }
] )
```

The document implementation

Attribute interfaces hardcoded

4. *"Retrieve the details of every ownership in the filesystem."*

```
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_owner"
  } },
  { "$lookup": {
    "from": "groups",
    "localField": "owner",
    "foreignField": "_id",
    "as": "user_group_owner"
  } },
  { "$addFields": { "owner": { "$concatArrays": [ "$user_owner", $user_group_owner" ] } } },
  { "$unwind": "$owner" },
  { "$addFields": {
    "owned_type": { "$switch": { "branches": [
      { "case": { "$eq": [ "$group_type", "user_group" ] }, "then": "user_group" },
      { "case": { "$eq": [ "$resource_type", "file" ] }, "then": "file" }
    ] } }
  } },
  ...
```

Return types are hardcoded with unions and lookups

```
  { "$addFields": {
    "owned_id": { "$switch": { "branches": [
      { "case": { "$eq": [ "$group_type", "user_group" ] }, "then": "$name" },
      { "case": { "$eq": [ "$resource_type", "file" ] }, "then": "$path" }
    ] } }
  } },
  { "$addFields": {
    "owner_type": { "$switch": { "branches": [
      { "case": { "$eq": [ "$owner.user_type", "user" ] }, "then": "user" },
      { "case": { "$eq": [ "$owner.user_type", "admin" ] }, "then": "admin" },
      { "case": { "$eq": [ "$owner.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": { "$eq": [ "$owner.user_type", "user" ] }, "then": "$owner.email" },
      { "case": { "$eq": [ "$owner.user_type", "admin" ] }, "then": "$owner.email" },
      { "case": { "$eq": [ "$owner.group_type", "user_group" ] }, "then": "$owner.name" }
    ] } }
  } }
] )
```

Type names hardcoded

The graph implementation

```
CREATE (cedric:User:Admin {email: "cedric@vaticle.com"});

CREATE (jimmy:User {email: "jimmy@vaticle.com"});

MATCH (cedric:Admin {email: "cedric@vaticle.com"})
CREATE
  (engineers:UserGroup {name: "engineers"}),
  (cedric)-[:OWNS {ownership_type: "GroupOwnership"}]->(engineers);

MATCH (jimmy:User {email: "jimmy@vaticle.com"})
CREATE
  (benchmark:Resource:File {path: "/jimmy/benchmark-results.xlsx"}),
  (jimmy)-[:OWNS {ownership_type: "ResourceOwnership"}]->(benchmark);

MATCH (engineers:UserGroup {name: "engineers"})
CREATE
  (roadmap:Resource:File {path: "/vaticle/feature-roadmap.pdf"}),
  (engineers)-[:OWNS {ownership_type: "ResourceOwnership"}]->(roadmap);
```

Multiple labels used to emulate node inheritance hierarchies

Properties must be used to emulate relationship inheritance hierarchies

The graph implementation

4. *"Retrieve the details of every ownership in the filesystem."*

```
MATCH (owner)-[ownership:OWNS]->(owned)
UNWIND labels(owned) AS owned_type
UNWIND labels(owner) AS owner_type
WITH ownership, owned, owned_type, owner, owner_type,
  {
    User: "email",
    Admin: "email",
    UserGroup: "name",
    File: "path"
  } AS id_type_map
WHERE owned_type IN keys(id_type_map)
AND id_type_map[owned_type] IN keys(owned)
AND owner_type IN keys(id_type_map)
AND id_type_map[owner_type] IN keys(owner)
AND NOT (
  owner_type = "User"
  AND "Admin" IN labels(owner)
)
RETURN
  ownership.ownership_type AS ownership_type,
  owned_type,
  owned[id_type_map[owned_type]] AS owned_id,
  owner_type,
  owner[id_type_map[owner_type]] AS owner_id;
```

The graph implementation

4. *"Retrieve the details of every ownership in the filesystem."*

```
MATCH (owner)-[ownership:OWNS]->(owned)
UNWIND labels(owned) AS owned_type
UNWIND labels(owner) AS owner_type
WITH ownership, owned, owned_type, owner, owner_type,
  {
    User: "email",
    Admin: "email",
    UserGroup: "name",
    File: "path"
  } AS id_type_map
WHERE owned_type IN keys(id_type_map)
AND id_type_map[owned_type] IN keys(owned)
AND owner_type IN keys(id_type_map)
AND id_type_map[owner_type] IN keys(owner)
AND NOT (
  owner_type = "User"
  AND "Admin" IN labels(owner)
)
RETURN
  ownership.ownership_type AS ownership_type,
  owned_type,
  owned[id_type_map[owned_type]] AS owned_id,
  owner_type,
  owner[id_type_map[owner_type]] AS owner_id;
```

Generic pattern matching allows multiple return types

Attribute interfaces hardcoded

Type inheritance hardcoded

An object model for a simple filesystem

- Relational, document, and graph databases all require metadata to be hardcoded into polymorphic queries.
- The queries do not actually express the natural-language questions being asked of the database.

An object model for a simple filesystem

The natural language question:

"Retrieve the details of every ownership in the file system, specifically: the type of ownership (group or resource), the type and ID of the owned object, and the type and ID of the object's owner."

An object model for a simple filesystem

The relational query:

```
SELECT ownership_type, owned_type, owned_id, owner_type, owner_id
FROM (
  SELECT
    'group_ownership' AS ownership_type,
    'user_group' AS owned_type,
    admin_of_group_ownerships.user_group_id AS owned_id,
    'admin' AS owner_type,
    admin_of_group_ownerships.admin_id AS owner_id
  FROM ownerships
  JOIN admin_of_group_ownerships USING (id)
  UNION
  SELECT
    'resource_ownership' AS ownership_type,
    'file' AS owned_type,
    files.path AS owned_id,
    'user' AS owner_type,
    user_of_resource_ownerships.user_id AS owner_id
  FROM ownerships
  JOIN user_of_resource_ownerships USING (id)
  JOIN files ON files.path = user_of_resource_ownerships.resource_id
  WHERE user_of_resource_ownerships.user_id NOT IN (
    SELECT admins.email
    FROM admins
  )
  ...
```

```
  UNION
  SELECT
    'resource_ownership' AS ownership_type,
    'file' AS owned_type,
    files.path AS owned_id,
    'admin' AS owner_type,
    admins.email AS owner_id
  FROM ownerships
  JOIN user_of_resource_ownerships USING (id)
  JOIN files ON files.path = user_of_resource_ownerships.resource_id
  JOIN admins ON admins.email = user_of_resource_ownerships.user_id
  UNION
  SELECT
    'resource_ownership' AS ownership_type,
    'file' AS owned_type,
    files.path AS owned_id,
    'user_group' AS owner_type,
    user_group_of_resource_ownerships.user_group_id AS owner_id
  FROM ownerships
  JOIN user_group_of_resource_ownerships USING (id)
  JOIN files ON files.path = user_group_of_resource_ownerships.resource_id
) AS ownerships;
```

An object model for a simple filesystem

The relational query:

"Retrieve the string `"group_ownership"`*, the string* `"user_group"`*, the user group ID, the string* `"admin"`*, and the admin ID for group ownerships by admins; retrieve the string* `"resource_ownership"`*, the string* `"file"`*, the file path, the string* `"user"`*, and the user ID for resource ownerships by users where the user is not an admin; retrieve the string* `"resource_ownership"`*, the string* `"file"`*, the file path, the string* `"admin"`*, and the admin email for resource ownerships by users where the user is an admin; and retrieve the string* `"resource ownership"`*, the string* `"file"`*, the file path, the string* `"user_group"`*, and the user group ID for resource ownerships by user groups."*

An object model for a simple filesystem

The application code:

```
fileSystemObjects.stream()
  .filter(object -> object instanceof Ownership.Owned)
  .map(object -> (Ownership.Owned) object)
  .map(owned -> Map.of(
    "owned-id", ((Id.Key<?>) owned).getId().value,
    "owned-type", owned.getClass().getSimpleName(),
    "owner-id", (((Id.Key<?>) owned.getOwner()).getId()).value,
    "owner-type", owned.getOwner().getClass().getSimpleName(),
    "ownership-type", owned.getOwnership().getClass().getSimpleName()
  ))
  .forEach(System.out::println);
```

An object model for a simple filesystem

The application code:

"Retrieve the details of every ownership in the file system, specifically: the type of ownership (group or resource), the type and ID of the owned object, and the type and ID of the object's owner."

An object model for a simple filesystem

Polymorphic queries in relational, document, and graph databases:

- Do not express the natural-language question.
- Are specific to data model implementations.
- Must be manually maintained when the data model changes.
- Lead to brittle application architectures.
- Are actually imperative in nature.

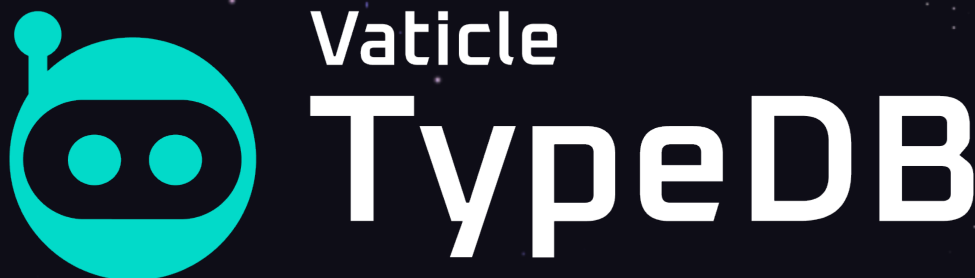
Summary

With contemporary database paradigms, it is not possible to do the following in the general case without external tools:

- Eliminate mismatch with object models in applications.
- Enforce semantic integrity of inserted polymorphic data.
- Write declarative polymorphic queries over inserted data.

If these things can be done natively in the application, why shouldn't they be natively possible in the database?

That's why we built...

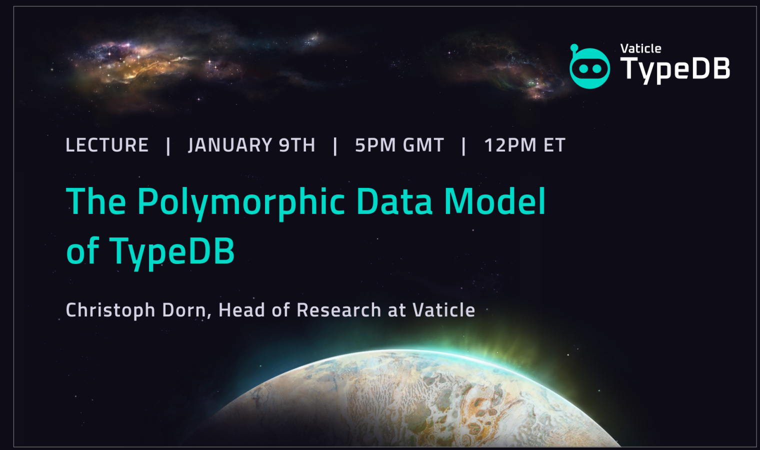


The world's first polymorphic database

There is much more to talk about, see these upcoming lectures:

Tuesday, Dec 12th

Tuesday, Jan 9th



Replay now available

Register or watch at TypeDB.com/lectures

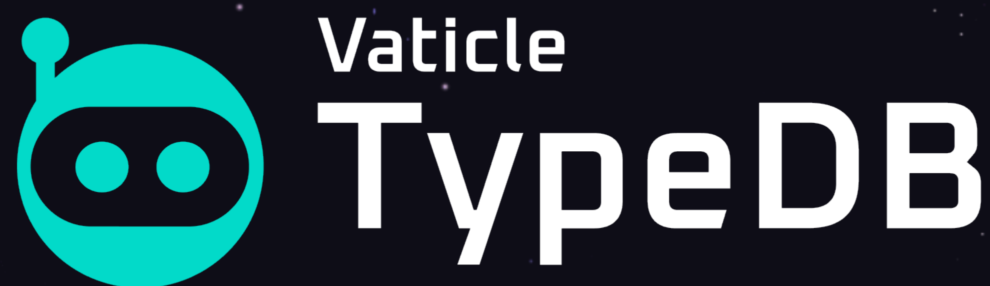
Q & A

More TypeDB Resources

TypeDB Learning Center - typedb.com/learn

Download TypeDB - typedb.com/deploy

TypeDB Cloud - cloud.typedb.com



Thank you!

Join us at typedb.com/discord