TypeDB vs RDF/OWL

For the Hypergraph Context Graph

Complete Technical Analysis with Code Examples

Core Question: Can TypeDB's PERA model be the right foundation for building the hypergraph-based decision context system described in the pitch deck?

Answer: Yes, and it's arguably the best existing database for this exact use case.

Generated: February 05, 2026

Part I: The Three Paradigms Compared

1.1 RDF/OWL: The Triple Paradigm

RDF is fundamentally built on triples: (subject, predicate, object)

RDF Triples:

```
:Acme :receivedDiscount :Deal_123 .
:Deal_123 :hasAmount "20%" .
:Deal_123 :approvedBy :VP_Sales .
:Deal_123 :underPolicy :RetentionPolicy .
```

The n-ary problem: When you need to represent a single decision involving multiple entities, RDF forces reification - creating an artificial intermediate node:

RDF Reification (to say 'VP approved 20% discount for Acme under retention policy'):

```
:Decision_001 rdf:type :DiscountApproval .
:Decision_001 :involvesCustomer :Acme .
:Decision_001 :involvesVP :VP_Sales .
:Decision_001 :involvesDeal :Deal_123 .
:Decision_001 :involvesPolicy :RetentionPolicy .
:Decision_001 :involvesIncidents :SEV1_List .
```

Issue	Description
Verbose	1 decision = 6+ triples
No native semantics	The "Decision_001" node is artificial scaffolding
Query complexity	SPARQL queries become nested and hard to optimize
Lost atomicity	Nothing in the data model says these triples form ONE event

1.2 Property Graphs (Neo4j): The Edge Paradigm

Property graphs allow properties on edges but are still fundamentally binary:

Cypher (Neo4j):

```
(Acme)-[:RECEIVED_DISCOUNT {amount: "20%"}]-&gt;(Deal_123)
(Deal_123)-[:APPROVED_BY]-&gt;(VP_Sales)
(Deal_123)-[:UNDER_POLICY]-&gt;(RetentionPolicy)
```

The n-ary problem: Same as RDF. You need to reify with an artificial node:

Cypher Reification:

```
CREATE (d:Decision {type: "DiscountApproval"})
CREATE (d)-[:INVOLVES]-&gt;(Acme)
CREATE (d)-[:APPROVED_BY]-&gt;(VP_Sales)
CREATE (d)-[:UNDER_POLICY]-&gt;(RetentionPolicy)
```

Problems: The Decision node is artificial, traversal must always go through it, no native concept of 'these entities participated in one atomic event'.

1.3 TypeDB PERA: The Relation Paradigm

TypeDB's key insight: Relations are first-class citizens that natively connect n entities.

TypeQL Schema Definition:

```
define
  relation discount-approval,
    relates customer,
    relates approver,
    relates deal,
    relates policy,
    relates justifying-incident;
```

```
  entity customer, plays discount-approval:customer;
  entity person, plays discount-approval:approver;
  entity deal, plays discount-approval:deal;
  entity policy, plays discount-approval:policy;
  entity incident, plays discount-approval:justifying-incident;
```

TypeQL Insert (a single decision):

```
insert
  $acme isa customer, has name "Acme";
  $vp isa person, has name "VP Sales";
  $deal isa deal, has id "Deal_123", has amount 0.20;
  $policy isa policy, has name "Retention";
  $incident isa incident, has severity "SEV-1";

  (customer: $acme, approver: $vp, deal: $deal,
   policy: $policy, justifying-incident: $incident) isa discount-approval;
```

This IS a hyperedge. The relation connects 5 entities in a single atomic structure.

Part II: Mapping TypeDB to Category Theory

2.1 The Pitch Deck's Mathematical Structure

From the pitch deck:

```
Hypergraph H = (V, E)
  V = entities (Customer, Deal, VP, Policy, ...)
  E = hyperedges (decisions connecting entities)

s-adjacency: Two hyperedges are s-adjacent iff they share &gt;= s nodes
s-path: Chain of hyperedges where consecutive pairs are s-adjacent
```

2.2 TypeDB's Native Mapping

Pitch Deck Concept	TypeDB Implementation
Vertex (node)	Entity or Attribute
Hyperedge	Relation instance
Hyperedge membership	Role playing (entity plays role in relation)
s-adjacency	Relations sharing >= s role players

Critical insight: TypeDB's relations ARE hyperedges. They natively connect arbitrary numbers of entities through typed roles.

2.3 Implementing IS >= 2 in TypeQL

Finding s-adjacent relations:

```
# Find all discount-approval decisions that share at least 2 entities
match
  $d1 isa discount-approval;
  $d2 isa discount-approval;
  $d1 != $d2;

  # Entity 1 shared
  $d1 (customer: $shared1);
  $d2 (customer: $shared1);

  # Entity 2 shared (could be any role)
  { $d1 (approver: $shared2); $d2 (approver: $shared2); } or
  { $d1 (policy: $shared2); $d2 (policy: $shared2); } or
  { $d1 (deal: $shared2); $d2 (deal: $shared2); };

fetch $d1, $d2;
```

With TypeDB 3.0 Functions:

```
define
fun shared_entities($r1: relation, $r2: relation) -&gt; integer:
  match
    $r1 ($role1: $entity);
    $r2 ($role2: $entity);
  reduce count($entity);

# Then query:
match
  $d1 isa discount-approval;
  $d2 isa discount-approval;
  $d1 != $d2;
  let $shared = shared_entities($d1, $d2);
  $shared &gt;= 2;
fetch $d1, $d2, $shared;
```

Part III: Nested Relations - The Killer Feature

3.1 The 2-Morphism Problem

The category theory document identifies a gap:

```
Current (IS &gt;= 2):     Structural similarity - "these edges share nodes"
Desired (2-morphisms): Reasoning dependency - "this decision REFERENCES that"
```

The pitch deck can find decisions that share entities. But it can't express:

- Decision B cited Decision A as **precedent**
- Decision B **overrides** Decision A
- Decision B **generalizes** from Decision A

3.2 TypeDB's Nested Relations = 2-Morphisms

TypeDB allows relations to play roles in other relations. This is exactly what you need for 2-morphisms:

Schema for 2-Morphisms:

```
define
  # The base decision relation (hyperedge)
  relation discount-approval,
    relates customer,
    relates approver,
    relates deal,
    relates policy,
    plays precedent-chain:precedent,      # Can BE a precedent
    plays precedent-chain:derived;         # Can HAVE a precedent

  # Meta-relation: relationship BETWEEN decisions (2-morphism!)
  relation precedent-chain,
    relates precedent,                     # The earlier decision
    relates derived,                       # The decision citing it
    owns precedent-type;                   # PRECEDENT, EXCEPTION, GENERALIZATION

  attribute precedent-type, value string;
```

Inserting a 2-Morphism:

```
insert
  # Decision 1: 20% discount for Acme (earlier)
  $d1 (customer: $acme, approver: $vp, deal: $deal1, policy: $retention)
      isa discount-approval;

  # Decision 2: 25% discount for Acme (later, citing d1)
  $d2 (customer: $acme, approver: $vp, deal: $deal2, policy: $retention)
      isa discount-approval;

  # The 2-morphism: d2 cites d1 as precedent
  (precedent: $d1, derived: $d2) isa precedent-chain,
      has precedent-type "PRECEDENT";
```

3.3 Coherence Checking in TypeQL

The coherence diagram from the category theory doc:

```
        e3 (Strategic Account)
       /  \
      B    Y
     /      \
   e1 &lt;--a-- e2
  (25%)    (20%)

Question: Does B = a . Y ? (Is the reasoning chain coherent?)
```

Query for coherence violations:

```
match
  # The three decisions
  $e1 isa discount-approval;
  $e2 isa discount-approval;
```

```
  $e3 isa account-status;  # "Strategic Account" designation

  # The 2-morphisms (meta-relations)
  (precedent: $e2, derived: $e1) isa precedent-chain;           # a: e2 -&gt; e1
  (justification: $e3, justified: $e1) isa justification-chain; # B: e3 -&gt; e1
  (justification: $e3, justified: $e2) isa justification-chain; # Y: e3 -&gt; e2

  # Check for inconsistency
  $e1 has discount-amount $amt1;
  $e2 has discount-amount $amt2;
  $amt1 &gt; $amt2;  # e1 gives more discount than its precedent

  # But e1 doesn't have additional justification beyond e2's
  not {
    (justification: $extra, justified: $e1) isa justification-chain;
    not { (justification: $extra, justified: $e2) isa justification-chain; };
  };

fetch $e1, $e2, $e3: name;
```

This is the 2-categorical coherence check that the category theory document said 'no implementations exist' for. TypeDB's nested relations make it possible.

Part IV: Feature Comparison

Feature	RDF/OWL	TypeDB PERA	Pitch Deck Need
N-ary relations	Reification (verbose)	Native	Native hyperedges
Typed roles	Properties only	First-class roles	Customer, Approver roles
Nested relations	Double reification	Native	2-morphisms
Schema validation	SHACL/OWL (complex)	Native strong typing	Structural constraints
Inheritance	RDFS subclassing	Native with interfaces	Polymorphic decisions
Query language	SPARQL (triples)	TypeQL (relations)	Pattern matching
Reasoning	OWL DL (complex)	Rule-based inference	Derived facts

4.1 The Verbosity Comparison

Representing a 5-entity decision:

RDF (reified) - 6 triples, 1 artificial node:

```
:Decision_001 rdf:type :DiscountApproval .
:Decision_001 :involvesCustomer :Acme .
:Decision_001 :involvesApprover :VP_Sales .
:Decision_001 :involvesDeal :Deal_123 .
:Decision_001 :involvesPolicy :RetentionPolicy .
:Decision_001 :involvesIncident :SEV1_001 .
```

TypeDB - 1 relation, no artificial nodes:

```
(customer: $acme, approver: $vp, deal: $deal,
 policy: $policy, incident: $incident) isa discount-approval;
```

Representing a 2-morphism (decision references decision):

RDF (double reification):

```
:PrecedentLink_001 rdf:type :PrecedentChain .
:PrecedentLink_001 :hasPrecedent :Decision_001 .
:PrecedentLink_001 :hasDerived :Decision_002 .
:PrecedentLink_001 :hasType "PRECEDENT" .
# And Decision_001 and Decision_002 are each 6+ triples...
```

TypeDB - 1 nested relation:

```
(precedent: $d1, derived: $d2) isa precedent-chain,
    has precedent-type "PRECEDENT";
```

Part V: Type-Theoretic Alignment

5.1 TypeDB's Dependent Types

The PERA model is grounded in dependent type theory. From TypeDB's documentation:

> *"The PERA model is a conceptual data model... based on the theory of dependent types, and interfaces serve as abstractions of the dependencies between data-storing types."*

When you define:

```
relation discount-approval,
  relates customer,
  relates approver;
```

You're saying: "A discount-approval **depends on** a customer and an approver." This is a dependent type: **DiscountApproval : Customer -> Approver -> Type**. The relation cannot exist without its role players.

5.2 Interface Polymorphism = Role Abstraction

Generic traversal across decision types:

```
define
  # Abstract role that any "participant" can play
  relation decision-event @abstract,
    relates participant;

  # Concrete decision types specialize the roles
  relation discount-approval sub decision-event,
    relates customer as participant,
    relates approver as participant,
    relates deal as participant;

  relation contract-signing sub decision-event,
    relates signatory as participant,
    relates witness as participant,
    relates contract as participant;
```

Polymorphic query across ALL decision types:

```
# Find any decisions sharing 2+ participants
match
  $d1 isa decision-event;
  $d2 isa decision-event;
  $d1 != $d2;
  $d1 (participant: $p1);
  $d2 (participant: $p1);
  $d1 (participant: $p2);
  $d2 (participant: $p2);
  $p1 != $p2;
fetch $d1, $d2;
```

Part VI: Why NOT RDF/OWL

6.1 OWL's Complexity Problem

OWL was designed for open-world reasoning on the Semantic Web. Its features include: Description Logic inference, transitive/symmetric/reflexive properties, cardinality restrictions, disjointness axioms. **Most of this is irrelevant or counterproductive for enterprise decision traces.**

6.2 The Open World Assumption

RDF/OWL uses the Open World Assumption: If something isn't stated, it's unknown (not false). **This is wrong for enterprise systems:**

- If a discount approval doesn't have VP sign-off recorded, it's **NOT approved**, not 'unknown'
- If a policy exception isn't documented, it **doesn't exist**, not 'might exist'

TypeDB uses Closed World Assumption with explicit schema constraints:

```
relation discount-approval,
  relates approver @card(1..);  # Must have at least one approver
```

6.3 The Query Language Gap

SPARQL (built for triple patterns):

```
SELECT ?decision WHERE {
  ?decision rdf:type :DiscountApproval .
  ?decision :involvesCustomer ?customer .
  ?decision :involvesApprover ?approver .
  # ...5 more lines for a single decision
}
```

TypeQL (built for relation patterns):

```
match
  (customer: $c, approver: $a, deal: $d, policy: $p) isa discount-approval;
fetch $c, $a, $d, $p;
```

Part VII: Implementation Architecture

7.1 Current vs Proposed Stack

```
Current:                          Proposed:
  HyperNetX (Python)                TypeDB
    -&gt; In-memory hypergraph           -&gt; Persistent hypergraph
  Custom BFS/Yen                    TypeQL
    -&gt; Traversal algorithms           -&gt; Declarative traversal
  LLM Agents                        LLM Agents (unchanged)
    -&gt; Interpretation                 -&gt; GraphAgent issues TypeQL
```

7.2 Complete Schema for the Pitch Deck

```
define
# === ENTITIES ===
entity customer,
  owns name,
  plays discount-approval:customer;

entity employee,
  owns name, owns title,
  plays discount-approval:approver,
  plays discount-approval:requester;

entity deal,
  owns deal-id @key, owns amount, owns discount-percentage,
  plays discount-approval:deal;

entity policy,
  owns policy-name, owns max-discount,
  plays discount-approval:governing-policy;

entity incident,
  owns incident-id @key, owns severity,
  plays discount-approval:justifying-incident;

# === DECISION HYPEREDGE ===
relation discount-approval,
  relates customer, relates approver, relates requester,
  relates deal, relates governing-policy,
  relates justifying-incident @card(0..),
  owns decision-timestamp, owns decision-rationale,
  plays precedent-chain:precedent,
  plays precedent-chain:derived,
  plays exception-override:base-decision,
  plays exception-override:exception-decision;

# === 2-MORPHISMS (Meta-relations) ===
relation precedent-chain,
  relates precedent, relates derived,
  owns precedent-type;

relation exception-override,
  relates base-decision, relates exception-decision,
  owns override-rationale;

# === FUNCTIONS FOR IS CONSTRAINT ===
fun count_shared_participants($r1: discount-approval,
                              $r2: discount-approval) -&gt; integer:
  match
    { $r1 (customer: $e); $r2 (customer: $e); } or
    { $r1 (approver: $e); $r2 (approver: $e); } or
    { $r1 (deal: $e); $r2 (deal: $e); } or
    { $r1 (governing-policy: $e); $r2 (governing-policy: $e); };
  reduce count($e);
```

Part VIII: Definitive Comparison

8.1 For Building the Pitch Deck's System

Requirement	RDF/OWL	Neo4j	TypeDB
Native hyperedges	No (Reify)	No (Reify)	YES
Typed roles	Properties	Labels	First-class
IS >= 2 queries	Complex	Complex	Functions
Nested relations	Double reify	Double reify	Native
Schema validation	SHACL	Weak	Strong
Polymorphic queries	RDFS	Manual	Native
Production ready	Yes	Yes	Yes

8.2 The Category-Theoretic View

Structure	Math Concept	RDF/OWL	TypeDB
Hyperedge	n-ary relation	Reified node	Native relation
s-adjacency	Line graph edge	SPARQL aggregation	TypeQL function
Path category	Morphisms = paths	Manual construction	Query composition
2-morphism	Morphism between morphisms	Double reification	Nested relation
Coherence	Diagram commutativity	Not expressible	Query constraints